

Massive Cross-Platform Simulations of Online Social Networks

Goran Murić, Alexey Tregubov, Jim Blythe, Andrés Abeliuk, Divya Choudhary, Kristina Lerman,
Emilio Ferrara

{gmuric,tregubov,blythe,aabeliuk,dchoudha,lerman,ferrarae}@isi.edu

USC Information Sciences Institute
Marina del Rey, CA, USA

ABSTRACT

As part of the DARPA SocialSim challenge, we address the problem of predicting behavioral phenomena including information spread involving hundreds of thousands of users across three major linked social networks: Twitter, Reddit and GitHub. Our approach develops a framework for data-driven agent simulation that begins with a discrete-event simulation of the environment populated with generic, flexible agents, then optimizes the decision model of the agents by combining a number of machine learning classification problems. The ML problems predict when an agent will take a certain action in its world and are designed to combine aspects of the agents, gathered from historical data, with dynamic aspects of the environment including the resources, such as tweets, that agents interact with at a given point in time. In this way, each of the agents makes individualized decisions based on their environment, neighbors and history during the simulation, although global simulation data is used to learn accurate generalizations. This approach showed the best performance of all participants in the DARPA challenge across a broad range of metrics. We describe the performance of models both with and without machine learning on measures of cross-platform information spread defined both at the level of the whole population and at the community level. The best-performing model overall combines learned agent behaviors with explicit modeling of bursts in global activity. Because of the general nature of our approach, it is applicable to a range of prediction problems that require modeling individualized, situational agent behavior from trace data that combines many agents.

KEYWORDS

massive scale simulations; collaborative platforms; agent based simulation; AI agents; online social networks

ACM Reference Format:

Goran Murić, Alexey Tregubov, Jim Blythe, Andrés Abeliuk, Divya Choudhary, Kristina Lerman, Emilio Ferrara. 2020. Massive Cross-Platform Simulations of Online Social Networks. In *Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), Auckland, New Zealand, May 9–13, 2020, IFAAMAS, 9 pages*.

1 INTRODUCTION

Agent-based simulations derived from observational data can be used as a powerful tool to model and predict how users react to different information to which they are exposed on social media platforms. However, a number of challenges present themselves in effectively using available data to build agent models that can be

used in high-fidelity simulations to make such predictions. First, the phenomena of interest may be the product of many agents following decision workflows that are not easily recoverable from the available data. Second, machine learning approaches have typically been used in this area to learn particular focused aspects of user behavior via classification methods (e.g. link-prediction, content suggestions), and have not been applied to larger questions of information flow and evolution. Third, ML models that do address global information flow tend to create a single global model, that is limited in its predictive power because it does not capture the differing reactions and activities of individual agents that might behave differently in the future because of variations to their individual environment and goals. Fourth, the data usually embodies observations of events that took place within a given set of social media platforms, meaning that positive examples of user activity abound while it is harder to find negative examples, for example of when and why a specific action was not taken. Finally, any learned or rule-based behavior must be designed to run very efficiently in order to scale simulations to hundreds of thousands or millions of users while modeling individual decisions taken in their dynamic contexts.

In this paper we present a framework for agent modeling of user behavior from observational data that addresses these challenges, and show its use in building an effective model of cross-platform behavior across GitHub, Twitter and Reddit. Some key contributions of this paper include:

- We develop a set of ML models that learn aspects of user behavior in Twitter and Reddit as classification problems, combining features of users and resources.
- We use a set of simple agent behavior models implemented in DASH framework [6, 9]. DASH allows developing cognitive agents and integrating our ML models along with simpler statistical and rule-based models of behavior.
- We develop features for ML learning that can be interpreted differently for different agents, such as ‘the volume of the Reddit channel I have most frequently contributed to’, to combine generalization from the data with individualized agent behavior.
- We present performance results of four model configurations that were rigorously evaluated during the DARPA Social-Sim Challenge event along with two baseline comparisons. Our models performed the best of all participants across a broad range of metrics defined for the event, indicating the promise of the approach we describe here. The best performing model overall combines learned agent behavior with statistical modeling of bursty behavior, indicating a beneficial interaction between this agent-centered learning and the architecture of the simulation.

The next section introduces the domain in which we modeled cross-platform information spread and the available data. We then introduce our agent model and the machine learning approaches used to define agent behavior. Next we present our experimental results and related work.

2 CHALLENGE PROBLEM DESCRIPTION

The DARPA SocialSim Challenge aims at simulating specific types of dynamics between users and content on three major online social platforms: GitHub, Twitter and Reddit. The goal of a challenge is to predict how a given unit of information will spread across a multi-platform online environment. In particular, it is designed to focus on the simulation of social structure and temporal dynamics with the focus on: 1) cross-platform information spread, 2) spread of specific units of information: Common vulnerabilities and exposures (CVE) IDs, URLs and malwares and 3) quantifying how the newly created information spreads. A unit of information is any trackable string that can be identified across platforms, including specific URLs, CVE IDs and malware-related keywords. For example, suppose that an important malware has been discussed in Twitter with hashtag `#malware_name`, and consequently the associated anti-malware software starts appearing in GitHub and is discussed in Reddit. The simulation should capture as precisely as possible the behaviors of all actors involved in the information spread.

All the teams participating in the challenge were provided with the *training data*: a set of relevant actions extracted from GitHub, Reddit and Twitter over the course of 32, 32 and 19 months respectively. The data covers the period before the end of August 2017, which corresponds to the end of the training data period in Figure 1. Some additional data has been provided such as additional information on users, the list of potential bot accounts or information on GitHub repositories. The teams had the freedom to use the provided data as needed to make the appropriate models. At the beginning of the challenge, the teams receive some additional data covering an unidentified period before the simulated time, which can be used to setup the *initial conditions* of the system (Figure 1). *Initial conditions* data cover period of 2 weeks right before test period (simulation time). A time *gap* between the training data and the initial conditions data could be introduced. The teams are requested to simulate a specific time interval after the initial conditions. Length of the *Test period* or *simulation time* is 6 weeks. The “accuracy” of a simulation has been assessed by comparing the *simulation output* to the *ground truth* using the set of measures as explained in Section 4.2.

At the beginning of the simulation, the initial condition data contains a set of actions and content which could contain the relevant keywords or no keywords at all. The users can interact with one or many pieces of content in any time during the simulation.

The simulation output is the list of all the actions performed by the relevant actors in all three platforms. For **GitHub**, we simulate ten actions that users can make: create or delete either a repository, a tag, or a branch (respectively *Create* and *Delete*), create or comment a pull request (respectively *PullRequest* and *PullRequestReviewComment*), create an issue (*Issues*, *IssueComment*), and push (*Push*, *CommitComment*). Moreover, a user can *watch* and *fork* existing repositories. For **Reddit** we simulate two user actions: *post* – creating the original content; and *comment* – commenting on a

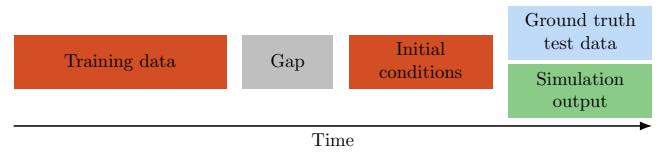


Figure 1: Simulation timeline. *Training data* has been used to train the models. *Initial conditions* data has been provided at the day of the challenge. It is used to set the initial state of the system before the simulation. After the simulation, the *simulation output* is then compared to *ground truth* to assess the accuracy of the simulation.

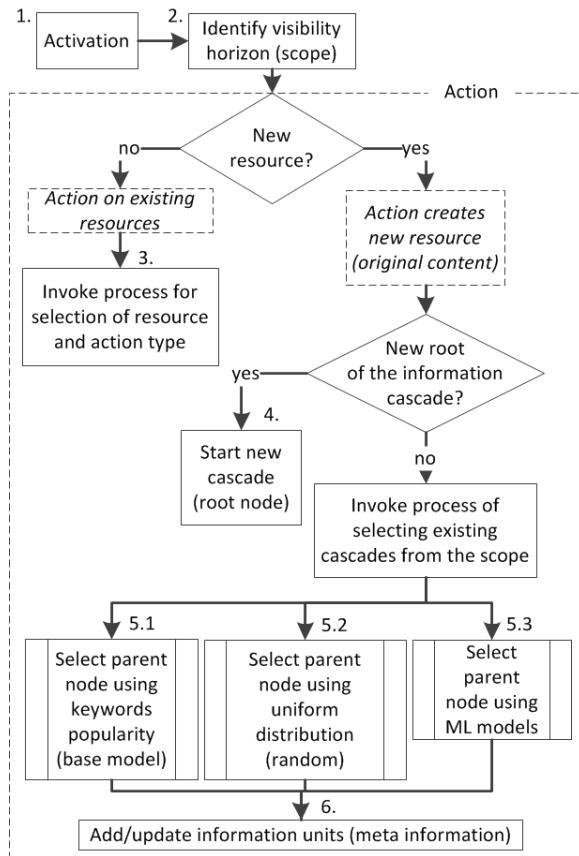
post or on another comment. For **Twitter** we focus on two user actions: *tweet* – creating an original tweet; and *retweet* – broadcasting the other user’s tweet.

3 AGENT FRAMEWORK AND MODELS

To simulate multiple social platforms with millions of users interacting with hundreds of thousands of pieces of content, we use the DASH agent framework implemented in the FARM simulation platform [9], implemented in Python. At the core of the simulation framework is discrete event simulation where the event queue contains two different kinds of events. First, individual agents appear on the queue with specific time-points. When these reach the front of the queue they begin a decision process to determine what action will be taken, e.g. a comment on a Reddit post. On completing an action the agent is placed back on the queue at a time point that implements a basic rate of activity for the agent. The second kind of object on the event queue is a trigger, which on reaching the front of the queue may conditionally initiate new events based on the state of the simulation. As we describe below, in this case we use triggers to model bursty behavior, or rapid increases in the rate of activity perhaps triggered by some event that is external to the social media environment. We call this technique a dual event queue approach. While it is simple, it combines flexibility in modeling global behavior with the efficiency of the discrete event model.

3.1 ML ready agents and simulation components

We describe our general approach to agent modeling of multiple social media platforms. Some researchers have modeled cross-platform information spread at the global level or with very simple agents [13, 18]. In contrast, we use a high-fidelity multi-agent model that allows agents to make independent choices using potentially complex decision algorithms. In reality, users of a social network have their personal preferences and habits. For example, one Twitter user may be interested in UK politics, while the other follows sports, and both of them are also interested in crypto coins. The first one might be very active and often engage in discussions, while the other one rarely replies. Users can vary widely across a number of different dimensions. In order to create such a model, however, we must specify this variety of behaviors for possibly millions of agents in a way that matches our observable data, and for this we turn to machine learning approaches as described below.

**Figure 2: Agents' workflow**

Agents in our simulation interact with each other by interacting with the social media resources they produce or are associated with. The nature of interactions can be various and multiple. For example, in Twitter we define agents as users and resources as tweets. Users can interact with tweets by retweeting them, liking them or replying to them.

The environment ϵ , then, consists of the agents $\{\alpha_1, \alpha_2, \dots, \alpha_n\} \in A$ and resources $\{r_1, r_2, \dots, r_m\} \in R$. Agents and resources are defined by their state. States include two sets of features X_{α_i} and X_{r_j} for agents and resources respectively.

All models used agents' decision process shown in Figure 2. This process is implemented as a DASH agent and consists of modular components that can have multiple implementations (marked 1-5). Experiments can also be configured to have heterogeneous agents using different modules implementations (algorithms).

The major components of the agents' decision process are the following:

- (1) *Activation*. Agents become active and ready to take an action in one of two ways: internal self-activation by coming to the front of the event queue as an agent or external activation via a trigger that comes to the front of the queue. These methods can be used simultaneously or separately. This dual queue approach for incorporating external signal into simulations described in Section 3.4
- (2) *Identification of the visible horizon*. It is both computationally and cognitively infeasible for the active agent to consider all possible resources in making an action decision. The agent's visible horizon h is defined by a set of rules that limit the resources the agent can possibly interact in a given time step. The users of a social network are presented with a very small subset of the resources they could in theory interact with, as illustrated in Figure 3a. For example, a Twitter user can interact only with tweets that are displayed on their device. Our attempt is to emulate this limited visibility of the available resources by allowing agents to interact only with a specific fraction of resources. The models for calculating the visibility horizon h are described more in details in Section 3.5
- (3) *Invoking the process of selecting a resource and action type*. If an agent makes a decision to take an action on an already existing resource (e.g. edit or delete its tweet) then it needs to select resources to interact with. This decision can be translated into the following set of problems: given the agent-resource pair with the corresponding sets of features, whether the action will be selected. These are classification problems, where a combined set of features $x = x_{\alpha_i} \parallel x_{r_j}$ needs to be classified into positive or negative class. Models developed for this problem are described in Section 3.2
- (4) *Starting a new information cascade*. Starting a information cascade means creating an original resource rather than acting on an already existing resource. Our simulation models use training data to sample the roots of information cascades. Information units (keywords, hashtags, URLs) are also sampled from the training data.
- (5) *Reacting to an existing information cascade*. This process is similar to (3). This decision can be translated into the following question: given the agent-resource pair with the corresponding sets of features, whether the action will be selected. Twitter and Reddit models developed for this problem are described in Section 3.2
- (6) *Adding/Updating information units*. Agents can add and update meta information of the resource (e.g. keywords, hashtags, URLs). Information spread is measured per each information unit and discussed in section 4.

Each of the agents in this framework are cross-platform agents in the sense that they can take actions on different platforms. The social platform, where the action happens, is specified when the agents' activation is scheduled on the queue. Parameterization and training required for each component (1-5) in Figure 2 is done for each social platform.

During the simulation, this process is repeated for each agent whenever it is activated as illustrated in Figure 3b. To optimize the use of the resources, we sometimes pre-compute, parallelize or bundle some of the steps when possible.

3.2 Twitter and Reddit ML models

ML models predict the probability of interaction between an agent and a resource. In the “real world” the users are presented with a set of objects they can interact with. For instance, a user on Twitter can see a few dozen tweets in a given moment. For each of the

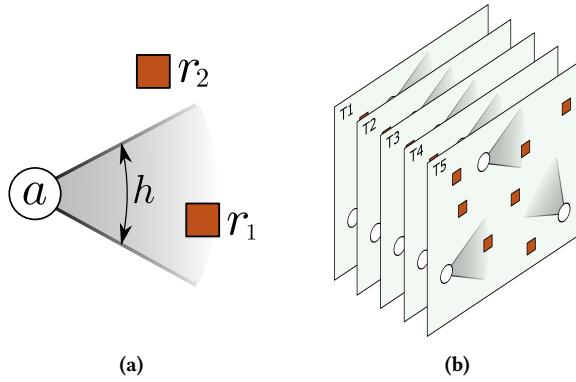


Figure 3: Agents, visibility horizon and simulation in time: a) An agent can interact with any resource which falls within their visibility horizon h . The decision if a particular agent will interact with a particular resource is left to other decision models; b) Multiple agents are active within the simulated environment, and they repeat their decision process over time

tweets, a user makes a decision either to like a tweet, retweet, reply or ignore it. In our simulation, for each of the actions the decision process is translated into a binary classification problem for a given feature vector $X_k = X_i^\alpha \parallel X_j^r \parallel X_k^c$. The feature vector is made up of feature vector X_i^α for an agent i , a feature vector X_j^r for a resource j and a combined feature vector X_k^c which are concatenated into a single vector X_k . This way, for each combination of agent-resource pairs we can construct the unique feature vector that will be used to make a decision on an action. During the simulation, as each agent reaches the front of the event queue, it solves a binary classification problem on the set of resources that are within agent's horizon h . For a given agent α_i , a vector X_k is constructed for all resources that are within the horizon h .

Reddit training. In the case of Reddit, we solve two classification problems: (1) *post*→*comment*: whether the user will make a comment on a given post and (2) *comment*→*comment*: whether the user will make a sub-comment on a given comment. For each of the two problems we build separate training datasets.

For the *post*→*comment* problem, we first match all the users with posts they commented on. Then, for each user-post pair we create a unified feature vector by concatenating user features, post features and combined features in a single vector as illustrated in Table 1. All the vectors created this way are assigned a positive label as they represent the instances of users commenting on the posts. The initial dataset consists only of positive examples. This problem is different from the standard supervised classification problem by the lack of negative examples in the training set. Usually, to properly train a binary classifier, one needs the negative examples. The data on “real” negatives could be harvested by carefully observing user’s behavior within the Reddit interface and select the posts user saw but decided to ignore. Such information is not available to us and we approach this task as the one-class classification problem.

Often referred to as *PU learning*, this problem is solved by building a binary classifier from a training set consisted of positive P

and unlabeled U data. To create the U part of the dataset, we match users with posts they did not comment to and create the feature vector from such user-post pairs. A fairly naive interpretation of the *PU learning* process is that unlabeled vectors that are similar to true positives will be labeled as positive and unlabeled vectors which are different from true positive will be labeled as negative. It is out of the scope of this paper to discuss the multiple nuances of *PU learning*. To build our training sets, we use an approach similar to one proposed in [20].

Finally, with the training set ready, we build a classifier able to predict if a given *user-post* pair will result in a user commenting on a post. After comparing multiple classification algorithms and parameter optimization, we decided to use Multi-layer Perceptron classifier with 15 hidden layers, which yielded the highest AUC score.

Table 1: To build the agent-based ML models, the training set has been build by assembling *user features*, *post features* and *combined features*.

	User features			Post features			Combined features			y
	U_0	...	U_n	P_0	...	P_m	C_0	...	C_z	
X_0	$x_{0,0}^\alpha$...	$x_{0,n}^\alpha$	$x_{0,0}^r$...	$x_{0,m}^r$	$x_{0,0}^c$...	$x_{0,m}^c$	1
X_1	$x_{1,0}^\alpha$...	$x_{1,n}^\alpha$	$x_{1,0}^r$...	$x_{1,m}^r$	$x_{1,0}^c$...	$x_{1,m}^c$	1
\vdots										
X_i	$x_{i,0}^\alpha$...	$x_{i,n}^\alpha$	$x_{i,0}^r$...	$x_{i,m}^r$	$x_{i,0}^c$...	$x_{i,m}^c$	0
X_k	$x_{-,0}^\alpha$...	$x_{-,n}^\alpha$	$x_{-,0}^r$...	$x_{-,m}^r$	$x_{k,0}^c$...	$x_{k,m}^c$	0

We approach the *comment*→*comment* problem in a similar way. The main difference is in the set of features used to predict the child comment. For the *post*→*comment* problem we match user features with the *post* features, while for the *comment*→*comment* problem we match user features with the *comment* features. This way, we allow the agents to apply the different models when deciding to comment, depending on the resource they are commenting on.

Reddit features. For the *post*→*comment* problem the final feature vector X_k consists of ≈ 200 features made from user features X_i^α , post features X_j^r and combined features X_k^c .

- User features X_i^α consist of preference features, sentiment features and frequency features.

Preference features are multidimensional vectors which represent the user’s preference towards *keywords*, *subreddits* and *domains*. The main idea behind building the preference vectors is to quantify the user’s interests and leaning towards certain topics. The preference vectors are built using FastText word embedding technique [10]. First we train a set of FastText models using the corpus from the relevant content on Reddit. Then, we assign the embedded vectors to each keyword, subreddit name and domain which appear in the training data. During the simulation, there is a possibility of encountering the keywords or subreddit names which do not exist in the training data. FastText can embed words which were never seen before but appear in a similar context or have a similar lexical features as the words from the training corpus. For each user, the preference vector is computed as the separate weighted averaged vector of all the

keywords, subreddits and domains they used. For example, the keyword preference vector for the user i is calculated as $Pk_i = \sum_{j=1}^n f_j K_j$, where K_j is an embedded vector of a keyword j used by the user, and f_j is the number of instances the keyword has been used. This way, we are able to identify the direction of the user interests. The preference vectors for the subreddits and domains are calculated in a similar fashion.

Sentiment features quantify the two dimensions of user's writing on Reddit, namely *polarity* and *subjectivity*. We calculate the means and the standard deviations of polarity and subjectivity of all user's posts and comments separately.

Frequency features quantify the user's activity by measuring the number of their posts, comments and subreddits together with the additional temporal measures of user's activity.

- *Post features* X_f^r , similarly to user features, consist of embedded vectors of keywords, subreddits and domains related to the post. Additionally, we use a single measure of sentiment polarity and subjectivity extracted from the post content. Furthermore, we include the info about the author of the post.
- *Combined features* X_k^c are the cosine and euclidean distances of user preference vectors and post-related embedded vectors. This way we quantify the similarity between the given post and user's interests in the past.

For the *comment*→*comment* problem we use a similar approach. Instead of the *post* features we calculate the similar set of *comment* features and concatenate it together with the user features and combined features. We add an additional feature which represents the position of a comment in a discussion tree.

Twitter Model For Twitter, we focus on modeling the 'retweet' action of a user. At any instance, a user has access to multitude of tweets on the platform. In our simulation, we develop a classification model for 'retweet' action *tweet, user*→*retweet*: whether a tweet will be retweeted for a given (tweet,user) combination. This model is applied to all tweets that an agent has access to whenever it reaches the front of the event queue. The simulation selects one of the tweets with highest probability according to the model and creates a retweet event for this agent and tweet. The final model used for the task was a 'Word2Vec' model combined with a 'random forest' model.

Twitter Feature Engineering In order to build a predictive retweet model, we need true indicative features from textual data to help the model learn. There are two main types of retweet activity by users on the platform- 'normal retweet' and 'quoted retweet'. A normal retweet is when a user retweets the original tweet without any additions, while the quoted retweet has additions to the original tweet. So a quoted retweet can give increased information of overlap of the tweet with the user profile as compared to the normal tweet. This understanding is important to decipher the learning of the model. The feature space used for the model consists of 3 types of features: *user features*, *tweet features* and *combined features*.

- *User features*: We create user profile in terms of their historical tweet activity on the platform. The intent is to model the user's persona on the platform based on their past activities. There are 84 features describing a user. One set of features is

focused on *user's profile* including user description, number of followers, number of friends etc. The other set is focused on the *tweet activity of a user* such as their average number of tweets per week, maximum number of user mentions per tweet, average retweet period etc. We also exploit hashtags used by users to model their interest in multiple topics floating on the platform in the form of tweets at any given time. The user description field has an interesting potential to define users' interests at a broader level.

A retweet score is separately assigned to each of the connections of a user. Every tweet has an author and the authors' retweet score for the user will be added to the final set of features. This is to capture the hypothesis that a user is more likely to retweet a tweet from one of their connections with the highest retweet score.

- *Tweet features*: Tweets are described with features including time of the day, day of the week, hashtags used, user mentions, number of characters and several others.
- *Combined features*: User historical features are adjoined with features of the entire set of tweets the user can take an action on at a given time. 'Word2Vec' model (W) is used to find out the similarity of the user's profile with the tweet features. It is trained on all hashtags of the corpus along with user descriptions. A representing feature vector V_k is obtained for all the hashtags. We create an average hashtag feature vector for a user H_{u_i} based on the most frequently used hashtags of the user. The user description vector D_{u_i} is created for each user. H_{u_i} and D_{u_i} are then used to find similarity of the user features with tweets. For each hashtag (V_k) of each tweet(t), cosine similarity is calculated with the H_{u_i} and D_{u_i} .

Feature engineering equipped the data with multiple defining features as mentioned above. For the current model, we use hashtags as major input for Word2Vec features for both normal retweet and quoted retweet. The model is then trained with the given data of feature set, retweet labels. A separate development set is used to tune parameters to eventually obtain a model accuracy of 92% on the test data.

3.3 GitHub Inverse Tree (ITree) model

Consider the following task: given a user u_i and the repository r_j , predict the number of actions $y_{i,j}$ the user u_i will perform on a repository r_j . This translates to a typical regression problem and could be approached in multiple ways. However, in GitHub the target repository is often uncertain while the rate of user's actions is typically already known, since the users activity rate varies slowly, and it is easily predicted just by observing the rates in the past. The active users remain active and the new users usually start with a slow rate of actions. Thus we solve the inverse problem: given a user u_i and the number of actions y_i , predict the repository r_i that user will most likely perform an action on. To predict the repository, we build a model that narrows down our choice to the set of repositories that share the same set of features.

For the given vectors of input and target variables, the regression tree finds a mapping from the input variables to a finite number of groups of output variables such that the average within group squared error of the target variables is minimized. The IT traverses

the trained regression tree backwards and for a given target variable and set of known features, extracts the boundaries of the unknown features. A regression model relates y to a function of X , so that $y \approx f(X)$. In our case, $X = X^u \parallel X^r \parallel X^c$, and we need to solve $X^r = f(y, X^u, X^c)$.

The problem of predicting the set of target variables resembles the problem known as a multi-output regression [5, 11], known also as the multi-target [1, 2] or multi-response regression [21]. Multi-output regression models are built to simultaneously predict multiple real-valued target variables. The result is usually a vector of values. However, instead of predicting a single vector, the ITree algorithm outputs two vectors: the first one L_i with lower bounds and the second one U_i with upper bounds for each feature we want to predict. Finally the output is processed as a set of tuples where $Y_i = \{(l_1, u_1), \dots, (l_m, u_m)\}$.

The ITree model works in the opposite direction compared to the “traditional” regression tree model. First, we build a training data set D of N instances such that each instance is characterized by an input vector of m real variables $X_k = (x_k^1, \dots, x_k^m)$ and a single target variable y_k . The input vectors are built by concatenating the user features X_i^u , repository features X_j^r and combined features X_k^c so that $X_k = X_i^u \parallel X_j^r \parallel X_k^c$. Then, we build a regression tree able to predict the number of user–repository actions, which is the target variable y . During the simulation, we do not predict y , as it is already estimated based on the previous user’s activity. However, we use the target variable y_i together with known features X_i^u and X_k^c and back-traverse the previously trained regression tree. By traversing the regression tree backwards, we extract the boundaries of unknown features X_j^r which describe the repository. Based on the lower and upper feature bounds, we select the set of repositories to choose from. The initial version of the model picks a single repository r_j from the set of chosen repositories uniformly at random and assigns the y_i actions from a user u_i to a repository r_j . For instance, assume it is known that user u_i will make z *push* actions. The ITree model will first calculate the feature boundaries of the repositories the user is most likely to *push*. This way we narrow down our selection to multiple repositories as the potential targets. Then, ITree model randomly chose z repositories from the narrow selection and simulates the *push* action of user u_i to chosen repositories.

3.4 Dual queue event scheduling and burstiness

The framework has two queues: one for scheduling agents own activation time (self-activation) and one for external signal or triggers. Events that are scheduled in the main event queue are executed as a discrete event schedule. Events from external signal queue represent triggers that activate events on the main queue. In this paper external queue was used for burst events. Use of the external signal queue is optional because agents can schedule their next events on the main queue using self-activation rate.

In this paper we discuss only one type of external triggers - burst events. In our experiments we observed that using only agent self-activation rate does not simulate burstiness of events well. For this reason we used the external signal queue to trigger bursts of events. Bursts were measured in training data. Sequence of bursts observed in training data was sampled replayed in simulation. Each burst

event was scheduled in the external signal queue. When a burst event is handled by the simulation controller, it creates multiple events associated with this burst and schedules them in the main queue. Size and length of each burst as well as participating users were sampled from bursts observed in training data. Kleinberg’s algorithm [17] was used to detect bursts.

3.5 Horizon and limited visibility models

Limited visibility models determine what active discussions and events are visible to agents to react to. Formally we define the agent’s horizon at a time step as all resources that are visible to an agent at the step. Depending on the platform it may be a list of repositories for GitHub, tweets and retweets for Twitter, posts and comments for Reddit that are visible to an agent. In our experiments we used resource age (time between when it was posted and when is viewed) to determine if it should be visible to an agent.

Resources from different platforms have different life spans. For GitHub we used all repositories observed during the training period (2–6 weeks of training data) and kept them during the test period (6 weeks). Twitter, Reddit and Telegram are more fast-paced social platforms. For example, the majority of Twitter discussions have a life span of 2 days.

We used the following algorithm to determine visible resources for each agent. Each information cascade (resource tree) stays visible x number of days where x is minimum between a fixed threshold (2 days) and estimated lifespan of the information cascade. Lifespan of the information cascade is estimated using training data. It is done as follows. When new information cascade is created (agent creates a root resource, e.g. original tweet), its metadata (e.g. features) is sampled from training data. These features include lifespan, which becomes an estimation for a lifespan of a new cascade.

4 RESULTS

4.1 Experiments and models

Our simulation produces a sequence of events that represent users’ activity on Reddit, Twitter, and GitHub. The resulting sequence of events is then compared to the ground truth according to a set of metrics described below. The ground truth is the sequence of events recorded from the actual social networks for the simulated period, as illustrated in Fig 1. In this paper we discuss the following model configurations:

- Random null model - this does not use the simulation framework but instead replays the training data in the test time period, randomly shuffling the order of events.
- Base model - a model where each step in agent’s decision workflow is executed using simple probabilistic model. Probability of each decision is based on frequency of that decision in the training data. For example, steps such as selecting action type or selecting a resource are separately determined using this approach.
- ML model (Twitter and Reddit) - an extension of the base model where for Reddit and Twitter platforms we use the ML model to determine how cascades grow (what tweets are retweeted and what posts are commented on). This is step 5 in Figure 2.

- Bursts model - an extension of the base model where in addition to self-activation agents are activated by burst events using the external signal queue as described in section 3.4. The approach samples from a set of bursts detected in the training data using Kleinberg’s algorithm [17]. For each burst, we sample from a subset of agents that are activated during a very short period of time (within 30 minutes) after the burst.
- Combined model - a combination of the Bursts and ML models from above.
- ITree for GitHub model - an extension of the base model where GitHub agents use ITree models to determine action type and resource (step 3 in Figure 2)

The ML model, Bursts model, Combined model and ITree for GitHub model are all built on top of the base model. The ML and ITree models augment the agent’s decision flow, and the Bursts model adds bursts in the event queue.

4.2 Evaluation metrics

In order to capture information spread across platforms we track *information units*, which are keywords, hashtags or associated URLs, depending on the platform. These are typically gathered from metadata, and one action or resource may be associated with multiple information units. In order to evaluate the predicted information spread across platforms, we measure the number of shares (e.g. number of retweets in Twitter, or comments in Reddit), the size of the audience (the number of unique users who shared), the speed of information spread, and its lifetime.

For model performance evaluation we use the following metrics:

- Temporal correlation of shares (audience sizes) - the Pearson correlation between the number of shares (audience sizes) time series is computed between all pairs of platforms. This distribution is computed for both simulation and the ground truth. Then the Kolmogorov-Smirnov test and Jensen-Shannon divergence are computed to measure similarity.
- Time delta - determines the number of hours it takes for a piece of information to appear on another platform. The Kolmogorov-Smirnov test and Jensen-Shannon divergence are computed to measure similarity.
- Correlation of shares (audiences) - the Pearson correlation between the activity (audience size) time series between all pairs of platforms. Measured as *RMSE*.
- Correlation of lifetimes - the Pearson correlation of the average lifetime for each platform, then correlation between simulation and the ground truth is computed. Measured as *RMSE*.
- Correlation of speeds - the Pearson correlation between speeds of information across platforms for each information unit, then correlation between simulation and the ground truth is measured. Measured as *RMSE*.

For burst measurements we use Kleinberg’s algorithm [17] to detect bursts on all platforms (population level measurement). Each metric related to bursts is computed as the *RMSE* of the correlation between simulation and the ground truth. Figure 4 shows *RMSE* of correlation of the following distributions:

- Distribution of the number of bursts

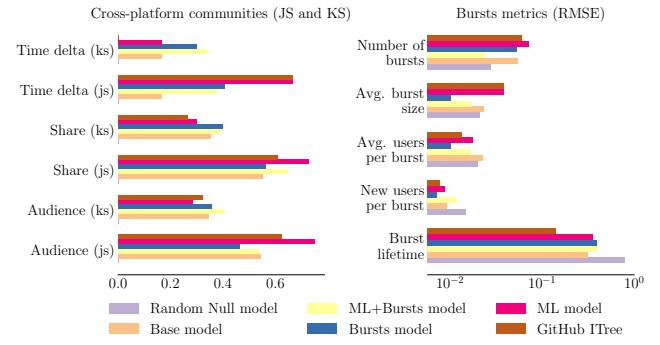


Figure 4: Cross-platform community and burst metrics. Our models outperform the base model in the majority of metrics.

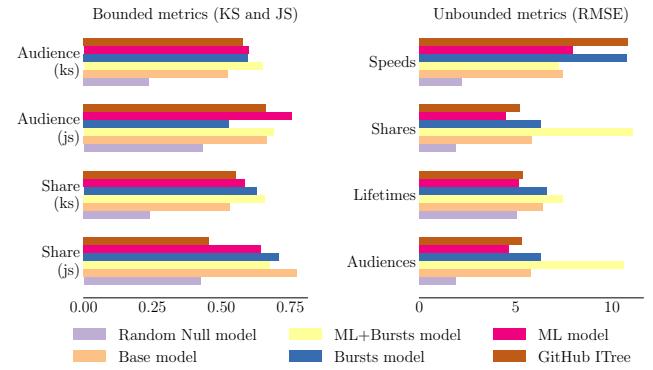


Figure 5: Cross-platform population metrics. Our models outperform both the base model and the random model in the majority of metrics.

- Distribution of the average burst size
- Distribution of the average number of users per burst
- Distribution of the number of new users per burst
- Distribution of bursts lifetime

In this paper we present measurements for two scales: population level and community level. Community is a subset of users grouped by a certain criteria (e.g. active on the same sub-reddit, use the same repositories or use similar information units, etc.). We identified several dozens of communities. Community level metrics were computed separately for each community and average is shown in Figure 4. Results for population level metrics (burst and cross-platform) are shown in Figure 5. Higher value are better in Figure 4 and 5 as we show $1/RMSE$.

We discuss these results in greater detail in below in Section 6.

5 RELATED WORK

There were several approaches proposed to study structure and user behavior on social media networks. In [4] authors explore various models to simulate a static structure of the social networks enabled by social media. The authors introduced several metrics for

network characterization generated by several agent-based simulations. Similar metrics were used in the DARPA SocialSim Challenge. Models discussed there focus on a static structure of the network and often limited to simulation of one social platform.

In recent years there has been evolving interest in studying cross-platform information spread (e.g. [13, 15, 18]). For example, in [18] authors studied a cross-platform spillover effects of viral videos. They found that when a video content appears on a lag platform it doubles its subsequent view growth on the original platform.

Simulation of information spread across-multiple platforms on the scale of tens of thousands and hundreds of thousands of agents can be computationally intensive. Previous work on scalability of agent-based simulation frameworks was done in [7, 8].

Applications of machine learning methods were also widely discussed for link prediction in social networks. There has been an extensive research done in the area of network embedding and it's application in link prediction [14, 16, 19, 22]. Wang et al [23] shows a link prediction problem as a two class discrimination problem and used machine learning approaches to learn prediction models. Link prediction is often used for recommending links on social networks [3].

Content-based link prediction methods have been developed and gained popularity among researchers. The large set of those models focus on predicting the activity in social networks based on combination of numerous content and user features [12, 25] Link-prediction and information spread prediction on social networks is often challenging because available training data can only provide positive feedback. Authors in [24] use retweet networks to address this problem (retweet network was used to reconstruct negative feedback).

6 CONCLUSION AND DISCUSSION

In this paper we present several agent-base simulation models that utilize machine learning in the agent's decision workflow. We also introduce a dual queue technique for discrete event scheduling that allows incorporating external event signals such as bursts. Our experiments on cross-platform information spread show that machine learning methods can be successfully used in multi-platform agent-based simulations, and they improve performance metrics compared to simple probabilistic models (base model and random null model).

In our experiments, all models perform significantly better in terms of the temporal correlation of cross-platform information spread than the random null model (see Figure 4 and Figure 5.). In fact, the random null model scores zeros across all community level metrics.

Both ML models and ITree model score on average 23% higher in Jensen-Shannon divergence of the temporal correlation of audiences. The results are similar for population and community level measurements, showing that the Reddit and Twitter ML models help agents to make a better (relative to base model) decisions about what tweets and posts should be retweeted. Also the GitHub ITree model helps make better decisions about which repositories agents interact with.

All models presented in the paper are data-driven, which allows us to build large-scale high-fidelity simulations. The results also

suggest that predicting bursts of events is hard without external signals. Reproducing bursts using burst samples taken from training data can improve temporal metrics.

The base model and random null model perform relatively well in burst metrics. In most burst metrics the bursts model does not perform well compared to all other models. This suggest that re-playing sequence of bursts from the initial conditions data-set does not improve the metrics that measure burst properties and distributions. However, the combined model (ML + Bursts) scores the best across almost all population level cross-platform information spread metrics while the bursts model and ML model separately come second and third respectively in these metrics. This can be explained by the fact that explicit modeling of bursts (bursts model) affects all time-sensitive metrics. Introducing bursts alone did not improve the burstiness metrics but improved temporal correlations of content shares and audience sizes.

The results presented in this paper lay ground for the future research and extensions of the proposed models. This includes incorporating external signals that drive activation of the agents. It can potentially improve prediction of event bursts (reaction of a subset of agents on external signal). Machine learning methods can also be applied to other aspects of agent's decision workflow. For example, when agents generate new information cascades they have to choose information units. This choice can also be learned from observational data.

Finally, there are multiple ways how event visibility horizon can be implemented. In this paper we used a naive approach with an estimation of the resource life span (which determines how long it will stay within the horizon). What rules define visible events can affect agents decisions. The visibility horizon model can be also used as a tool for modeling social network users' cognitive load and limitations.

7 ACKNOWLEDGEMENTS

The authors are grateful to the Defense Advanced Research Projects Agency (DARPA), contract W911NF-17-C-0094, for their support.

REFERENCES

- [1] Timo Aho, Bernard Ženko, Sašo Džeroski, and Tapio Elomaa. 2012. Multi-Target Regression with Rule Ensembles. *Journal of Machine Learning Research* 13, Aug (2012), 2367–2407.
- [2] Annalisa Appice and Saso Džeroski. 2007. Stepwise Induction of Multi-target Model Trees. In *Machine Learning: ECML 2007*. Springer Berlin Heidelberg, Berlin, Heidelberg, 502–509. https://doi.org/10.1007/978-3-540-74958-5_46
- [3] Lars Backstrom and Jure Leskovec. 2011. Supervised Random Walks: Predicting and Recommending Links in Social Networks. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining (WSDM '11)*. ACM, New York, NY, USA, 635–644. <https://doi.org/10.1145/1935826.1935914>
- [4] Federico Bergenti, Enrico Franchi, and Agostino Poggi. 2011. Selected models for agent-based simulation of social networks. In *3rd Symposium on Social Networks and Multiagent Systems (SNAMAS 2011)*, 27–32.
- [5] Hendrik Blockeel, Luc De Raedt, and Jan Ramon. 2000. Top-down induction of clustering trees. (nov 2000). arXiv:cs/0011032
- [6] Jim Blythe. 2012. A dual-process cognitive model for testing resilient control systems. In *2012 5th International Symposium on Resilient Control Systems*. IEEE, 8–12. <https://doi.org/10.1109/ISRCS.2012.6309285>
- [7] James Blythe, Emilia Ferrara, Di Huang, Kristina Lerman, Goran Muric, Anna Sapienza, Alexey Tregubov, Diogo Pacheco, John Bollenbacher, Alessandro Flammini, et al. 2019. The DARPA SocialSim Challenge: Massive Multi-Agent Simulations of the Github Ecosystem. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 1835–1837.
- [8] J. Blythe and A. Tregubov. 2018. FARM: Architecture for Distributed Agent-Based Social Simulations. In *Massively Multi-Agent Systems II*, Donghui Lin, Toru Ishida, Franco Zambonelli, and Itsuki Noda (Eds.). Springer International Publishing, 96–107.
- [9] Jim Blythe and Alexey Tregubov. 2019. FARM: Architecture for Distributed Agent-Based Social Simulations. Springer, Cham, 96–107. https://doi.org/10.1007/978-3-030-20937-7_7
- [10] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. (jul 2016). arXiv:1607.04606
- [11] Hanen Borchani, Gherardo Varando, Concha Bielza, and Pedro Larrañaga. 2015. A survey on multi-output regression. *WIREs Data Mining Knowl Discov* 5 (2015), 216–233. <https://doi.org/10.1002/widm.1157>
- [12] Ethem F. Can, Hüseyin Oktay, and R. Manmatha. 2013. Predicting retweet count using visual cues. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management - CIKM '13*. ACM Press, New York, New York, USA, 1481–1484. <https://doi.org/10.1145/2505515.2507824>
- [13] Joseph D O'Brien, Ioannis K Dassios, and James P Gleeson. 2019. Spreading of memes on multiplex networks. *New Journal of Physics* 21, 2 (2019), 025001.
- [14] Palash Goyal and Emilio Ferrara. 2018. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems* (2018). <https://doi.org/10.1016/j.knosys.2018.03.022> arXiv:1705.02801
- [15] Anna-Katharina Jung, Milad Mirbabaei, Björn Ross, Stefan Stieglitz, Christoph Neuberger, and Sanja Kapidzic. 2018. Information Diffusion between Twitter and Online Media. (2018).
- [16] Seyed Mehran Kazemi and David Poole. 2018. Simple embedding for link prediction in knowledge graphs. In *Advances in Neural Information Processing Systems*, Vol. 2018–December. Neural information processing systems foundation, 4284–4295. arXiv:1802.04868
- [17] Jon Kleinberg. 2003. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery* 7, 4 (2003), 373–397.
- [18] Haris Kriještorac, Rajiv Garg, and Vijay Mahajan. 2019. Cross-Platform Spillover Effects in Consumption of Viral Content: A Quasi-Experimental Analysis Using Synthetic Controls. *Available at SSRN 3011533* (2019).
- [19] L. Linyuan and Tao Zhou. 2011. Link prediction in complex networks: A survey. (mar 2011), 1150–1170 pages. <https://doi.org/10.1016/j.physa.2010.11.027> arXiv:1010.0725
- [20] F. Mordelet and J.-P. Vert. 2014. A bagging SVM to learn from positive and unlabeled examples. *Pattern Recognition Letters* 37 (feb 2014), 201–209. <https://doi.org/10.1016/j.patrec.2013.06.010>
- [21] Timo Similä and Jarkko Tikka. 2007. Input selection and shrinkage in multiresponse linear regression. *Computational Statistics & Data Analysis* 52 (2007), 406–422.
- [22] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale information network embedding. In *WWW 2015 - Proceedings of the 24th International Conference on World Wide Web*. Association for Computing Machinery, Inc, 1067–1077. <https://doi.org/10.1145/2736277.2741093> arXiv:1503.03578
- [23] Peng Wang, BaoWen Xu, YuRong Wu, and XiaoYu Zhou. 2015. Link prediction in social networks: the state-of-the-art. *Science China Information Sciences* 58, 1 (01 Jan 2015), 1–38. <https://doi.org/10.1007/s11432-014-5237-y>
- [24] Tauhid R Zaman, Ralf Herbrich, Jürgen Van Gael, and David Stern. 2010. Predicting information spreading in twitter. In *Workshop on computational social science and the wisdom of crowds, nips*, Vol. 104. Citeseer, 17599–601.
- [25] Qi Zhang, Yeyun Gong, Jindou Wu, Haoran Huang, and Xuanjing Huang. 2016. Retweet Prediction with Attention-based Deep Neural Network. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management - CIKM '16*. ACM Press, New York, New York, USA, 75–84. <https://doi.org/10.1145/2983323.2983809>



“Although I do firmly
believe that the
brain is a machine,
whether this
machine is a computer
is another question”

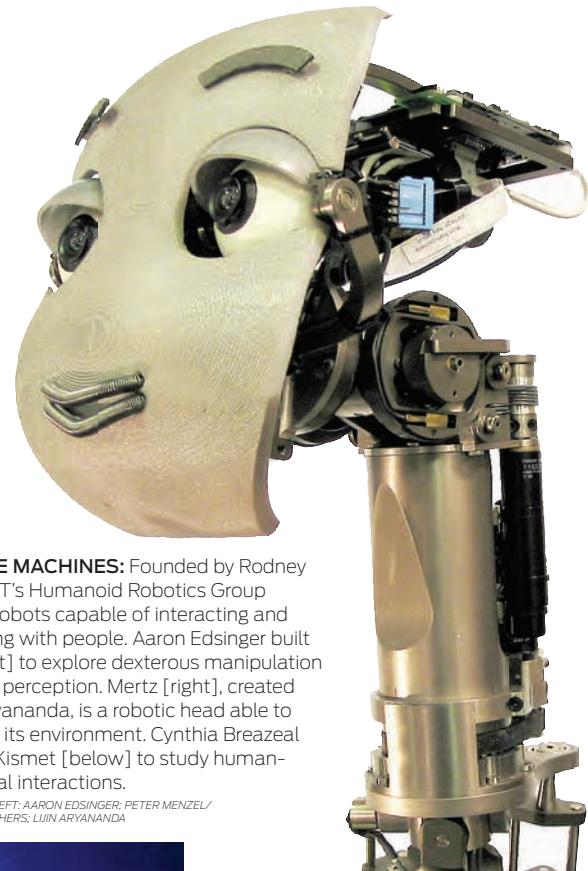
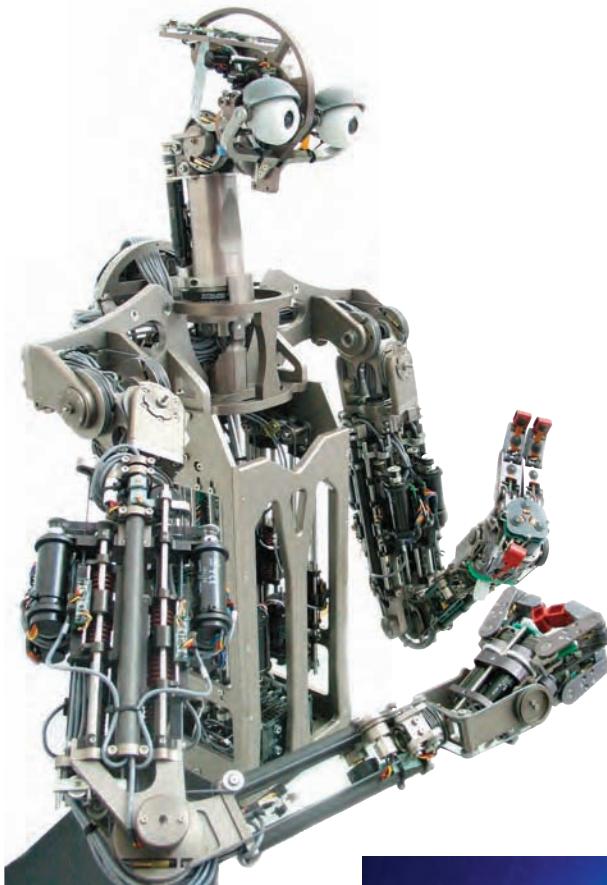
—RODNEY BROOKS



A POWERFUL ARTIFICIAL INTELLIGENCE WON'T SPRING FROM A SUDDEN TECHNOLOGICAL "BIG BANG"—IT'S ALREADY EVOLVING SYMBIOTICALLY WITH US
BY RODNEY BROOKS

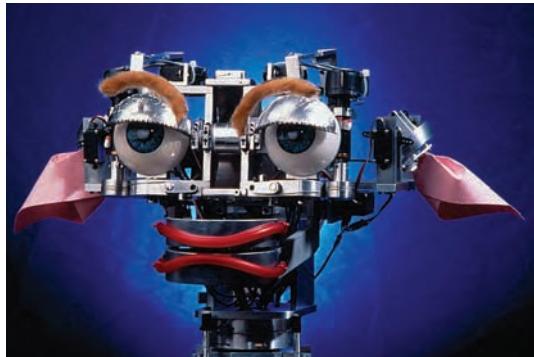
I, Rodney Brooks, Am a Robot

AM A MACHINE. So are you. ¶ Of all the hypotheses I've held during my 30-year career, this one in particular has been central to my research in robotics and artificial intelligence. I, you, our family, friends, and dogs—we all are machines. We are really sophisticated machines made up of billions and billions of biomolecules that interact according to well-defined, though not completely known, rules deriving from physics and chemistry. The biomolecular interactions taking place inside our heads give rise to our intellect, our feelings, our sense of self. ¶ Accepting this hypothesis opens up a remarkable possibility. If we really are machines and if—this is a big *if*—we learn the rules governing our brains, then in principle there's no reason why we shouldn't be able to replicate those rules in, say, silicon and steel. I believe our creation would exhibit genuine human-level intelligence, emotions, and even consciousness.



SOCIAL MACHINES: Founded by Rodney Brooks, MIT's Humanoid Robotics Group develops robots capable of interacting and cooperating with people. Aaron Edsinger built Domo [left] to explore dexterous manipulation and visual perception. Mertz [right], created by Lijin Aryananda, is a robotic head able to learn from its environment. Cynthia Breazeal designed Kismet [below] to study human-robot social interactions.

PHOTOS, FROM LEFT: AARON EDSINGER; PETER MENZEL/
PHOTO RESEARCHERS; LIJIN ARYANANDA



I'm far from alone in my conviction that one day we will create a human-level artificial intelligence, often called an artificial general intelligence, or AGI. But how and when we will get there, and what will happen after we do, are now the subjects of fierce debate in my circles. Some researchers believe that AGIs will undergo a positive-feedback self-enhancement until their comprehension of the universe far surpasses our own. Our world, those individuals say, will change in unfathomable ways after such superhuman intelligence comes into existence, an event they refer to as the singularity.

Perhaps the best known of the people proselytizing for this singularity—let's call them singularitarians—are acolytes of Raymond Kurzweil, author of *The Singularity Is Near: When Humans Transcend Biology* (Viking, 2005) and board member of the Singularity Institute for Artificial Intelligence, in Palo Alto, Calif. Kurzweil and his colleagues believe that this super AGI will be created either through ever-faster advances in artificial intelligence or by more biological means—"direct brain-computer interfaces, biologi-

cal augmentation of the brain, genetic engineering, [and] ultrahigh-resolution scans of the brain followed by computer emulation" are some of their ideas. They don't believe this is centuries away; they think it will happen sometime in the next two or three decades.

What will the world look like then? Some singularitarians believe our world will become a kind of techno-utopia, with humans downloading their consciousnesses into machines to live a disembodied, after-death life. Others, however, anticipate a kind of technodamnation in which intelligent machines will be in conflict with humans, maybe waging war against us. The proponents of the singularity are technologically astute and as a rule do not appeal to technologies that would violate the laws of physics. They well understand

the rates of progress in various technologies and how and why those rates of progress are changing. Their arguments are plausible, but plausibility is by no means certainty.

My own view is that things will unfold very differently. I do not claim that any specific assumption or extrapolation of theirs is faulty. Rather, I argue that an artificial intelligence could evolve in a much different way. In particular, I don't think there is going to be one single sudden technological "big bang" that springs an AGI into "life." Starting with the mildly intelligent systems we have today, machines will become gradually more intelligent, generation by generation. The singularity will be a period, not an event.

This period will encompass a time when we will invent, perfect, and deploy, in fits and starts, ever more capable systems, driven not by the imperative of the singularity itself but by the usual economic and sociological forces. Eventually, we will create truly artificial intelligences, with cognition and consciousness recognizably similar to our own. I have no idea how, exactly, this creation will come about. I also don't know when it will happen, although I strongly sus-

pect it won't happen before 2030, the year that some singularitarians predict.

But I expect the AGIs of the future—embodied, for example, as robots that will roam our homes and workplaces—to emerge gradually and symbiotically with our society. At the same time, we humans will transform ourselves. We will incorporate a wide range of advanced sensory devices and prosthetics to enhance our bodies. As our machines become more like us, we will become more like them.

And I'm an optimist. I believe we will all get along.

LIKE MANY AI researchers, I've always dreamed of building the ultimate intelligence. As a longtime fan of *Star Trek*, I have wanted to build Commander

Data, a fully autonomous robot that we could work with as equals. Over the past 50 years, the field of artificial intelligence has made tremendous progress. Today you can find AI-based capabilities in things as varied as Internet search engines, voice-recognition software, adaptive fuel-injection modules, and stock-trading applications. But you can't engage in an interesting heart-to-power-source talk with any of them.

We have many very hard problems to solve before we can build anything that might qualify as an AGI. Many problems have become easier as computer power has reliably increased on its exponential and seemingly inexorable merry way. But we also need fundamental breakthroughs, which don't follow a schedule.

To appreciate the challenges ahead of us, first consider four basic capabilities that any true AGI would have to possess. I believe such capabilities are fundamental to our future work toward an AGI because they might have been the foundation for the emergence, through an evolutionary process, of higher levels of intelligence in human beings. I'll describe them in terms of what children can do.

■ The object-recognition capabilities of a 2-year-old child. A 2-year-old can observe a variety of objects of some type—different kinds of shoes, say—and successfully categorize them as shoes, even if he or she has never seen soccer cleats or suede oxfords. Today's best computer vision systems still make mistakes—both false positives and false negatives—that no child makes.

■ The language capabilities of a 4-year-old child. By age 4, children can

engage in a dialogue using complete clauses and can handle irregularities, idiomatic expressions, a vast array of accents, noisy environments, incomplete utterances, and interjections, and they can even correct nonnative speakers, inferring what was really meant in an ungrammatical utterance and reformatting it. Most of these capabilities are still hard or impossible for computers.

■ The manual dexterity of a 6-year-old child. At 6 years old, children can grasp objects they have not seen before; manipulate flexible objects in tasks like tying shoelaces; pick up flat, thin objects like playing cards or pieces of paper from a tabletop; and manipulate unknown objects in their pockets or in a bag into which they can't see. Today's robots can at most do any one of these things for some very particular object.

■ The social understanding of an 8-year-old child. By the age of 8, a child can understand the difference between what he or she knows about a situation and what another person could have observed and therefore could know. The child has what is called a "theory of the mind" of the other person. For example, suppose a child sees her mother placing a chocolate bar inside a drawer. The mother walks away, and the child's brother comes and takes the chocolate. The child knows that in her mother's mind the chocolate is still in the drawer. This ability requires a level of perception across many domains that no AI system has at the moment.

But even if we solve these four problems using computers, I can't help wondering: What if there are some essential aspects of intelligence that we still do not understand and that do not lend themselves to computation? To a large extent we have all become computational bigots, believers that any problem can be solved with enough computing power. Although I do firmly believe that the brain is a machine, whether this machine is a computer is another question.

I recall that in centuries past the brain was considered a hydrodynamic machine. René Descartes could not believe that flowing liquids could produce thought, so he came up with a mind-body dualism, insisting that mental phenomena were nonphysical. When I was a child, the prevailing view was that the brain was a kind of telephone-switching network. When I was a teenager, it became an electronic computer, and later, a massively parallel digital computer. A few

years ago someone asked me at a talk I was giving, "Isn't the brain just like the World Wide Web?"

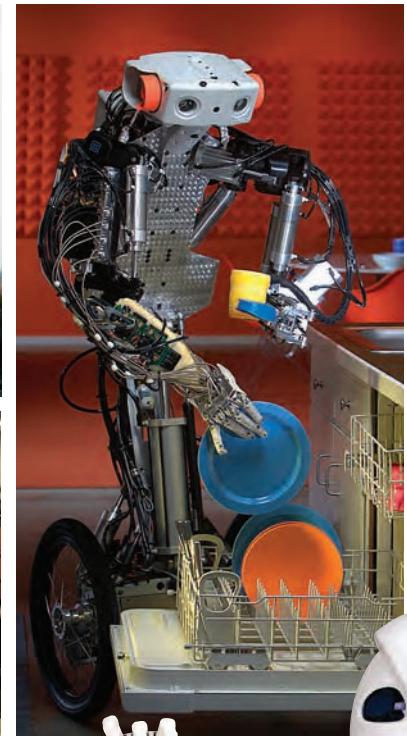
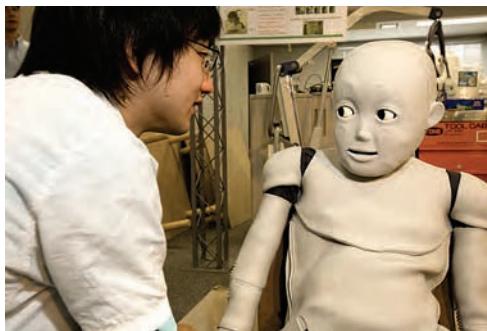
We use these metaphors as the basis for our philosophical thinking and even let them pervade our understanding of what the brain truly does. None of our past metaphors for the brain has stood the test of time, and there is no reason to expect that the equivalence of current digital computing and the brain will survive. What we might need is a new conceptual framework: new ways of sorting out and piecing together the bits of knowledge we have about the brain.

Creating a machine capable of effectively performing the four capabilities above may take 10 years, or it may take 100. I really don't know. In 1966, some AI pioneers at MIT thought it would take three months—basically an undergraduate student working during the summer—to completely solve the problem of object recognition. The student failed. So did I in my Ph.D. project 15 years later. Maybe the field of AI will need several Einsteins to bring us closer to ultraintelligent machines. If you are one, get to work on your doctorate now.

GREW UP IN a town in South Australia without much technology. In the late 1960s, as a teenager, I saw *2001: A Space Odyssey*, and it was a revelation. Like millions of others, I was enthralled by the soft-spoken computer villain HAL 9000 and wondered if we could one day get to that level of artificial intelligence. Today I believe the answer is yes. Nevertheless, in hindsight, I believe that HAL was missing a fundamental component: a body.

My early work on robotic insects showed me the importance of coupling AI systems to bodies. I spent a lot of time observing how those creatures crawled their way through complex obstacle courses, their gaits emerging from the interaction of their simple leg-control programs and the environment itself. After a decade building such insectoids, I decided to skip robotic lizards and cats and monkeys and jump straight to humanoids, to see what I could do there.

My students and I have learned a lot simply by putting people in front of a robot and asking them to talk to the machine. One of the most surprising things we've observed is that if a robot has a humanlike body, people will interact with it in a humanlike way. That's one of the reasons I came to believe that to



ANDROIDS ARISING: Clockwise from bottom left: Honda's Asimo can dance and climb stairs and has even conducted an orchestra; the Actroid female humanoid was developed by the Japanese firm Kokoro and Osaka University; Chinese roboticist Zou Ren Ti, of the Xi'an Chaoren Sculpture Research Institute, sits next to his android twin [right]; Anybots, in Mountain View, Calif., is developing tele-operated mechanical servants; Toyota's Partner robots include little droids that play the



trumpet and the violin; Osaka University researchers built the CB2 robot to mimic the appearance and behavior of a toddler.

PHOTOS, CLOCKWISE FROM BOTTOM LEFT: HONDA; YVES GELLIE/CORBIS; INDIANA UNIVERSITY; ANYBOTS; TOYOTA; ANDRONIKI CHRISTODOULOU/WPN

build an AGI—and its predecessors—we'll need to give them a physical constitution.

At this point, I can guess what you're wondering. What will AGIs look like and when will they be here? What will it be like to interact with them? Will they be sociable, fun to be around?

I believe robots will have myriad sizes and shapes. Many will continue to be simply boxes on wheels. But I don't see why, by the middle of this century, we shouldn't have humanoid robots with agile legs and dexterous arms and hands. You won't have to read a manual or enter commands in C++ to operate them. You will just speak to them, tell them what to do. They will wander around our homes, offices, and factories, performing certain tasks as if they were people. Our environments were designed and built for our bodies, so it will be natural to have these human-shaped robots around to

perform chores like taking out the garbage, cleaning the bathtub, and carrying groceries.

Will they have complex emotions, personalities, desires, and dreams? Some will, some won't. Emotions wouldn't be much of an asset for a bathtub-cleaning robot. But if the robot is reminding me to take my meds or helping me put the groceries away, I will want a little more personal interaction, with the sort of feedback that lets me know not just whether it's understanding me but how it's understanding me. So I believe the AGIs of the future will not only be able to act intelligently but also convey emotions, intentions, and free will.

So now the big question is: Will those emotions be real or just a very sophisticated simulation? Will they be the same kind of stuff as our own emotions? All I can give you is my hypothesis: the robot's

emotional behavior can be seen as the real thing. We are made of biomolecules; the robots will be made of something else. Ultimately, the emotions created in each medium will be indistinguishable. In fact, one of my dreams is to develop a robot that people feel bad about switching off, as if they were extinguishing a life. As I wrote in my book *Flesh and Machines* (Pantheon, 2002), "We had better be careful just what we build, because we might end up liking them, and then we will be morally responsible for their well-being. Sort of like children."

MANY OF THE advocates of the singularity appear to the more sober observers of technology to have a messianic fervor about their predictions, an unshakable faith in the certainty of their predicted future.

To an outsider, a lot of their convictions seem to have many commonalities with religious beliefs. Many singularitarians believe people will conquer death by downloading their consciousnesses into machines before their bodies give out. They expect this option will become available, conveniently enough, within their own lifetimes.

But for the sake of argument, let's accept all the wildest hopes of the singularitarians and accept that we will somehow construct an AGI in the next three or four decades. My view is that we will not live in the techno-utopia future that is so fervently hoped for. There are many possible alternative futures that fit within the themes of the singularity but are very different in their outcomes.

One scenario often considered by singularitarians, and Hollywood, too, is that an AGI might emerge spontaneously on a large computer network. But perhaps such an AGI won't have quite the relationship with humans that the singularitarians expect. The AGI may not know about us, and we may not know about it.

In fact, maybe some kind of AGI already exists on the Google servers, probably the single biggest network of computers on our planet, and we aren't aware of it. So at the 2007 Singularity Summit, I asked Peter Norvig, Google's chief scientist, if the company had noticed any unexpected emergent properties in its network—not full-blown intelligence, but any unexpected emergent property. He replied that they had not seen anything like that. I suspect we are a long, long way from consciousness unexpectedly showing up in the Google network. (Unless it is already there and cleverly concealing its tracks!)

Here's another scenario: the AGI might know about us and we know about it, but it might not care about us at all. Think of chipmunks. You see them wandering around your garden as you look out the window at breakfast, but you certainly do not know them as individuals and probably do not give much thought to which ones survive the winter. To an AGI, we may be nothing more than chipmunks.

From there it's only a short step to the question I'm asked over and over again: Will machines become smarter than us and decide to take over?

I don't think so. To begin with, there will be no "us" for them to take over from. We, human beings, are already starting

to change ourselves from purely biological entities into mixtures of biology and technology. My prediction is that we are more likely to see a merger of ourselves and our robots before we see a stand-alone superhuman intelligence.

Our merger with machines is already happening. We replace hips and other parts of our bodies with titanium and steel parts. More than 50 000 people have tiny computers surgically implanted in their heads with direct neural connections to their cochleas to enable them to hear. In the testing stage, there are retina microchips to restore vision and motor implants to give quadriplegics the ability to control computers with thought. Robotic prosthetic legs, arms, and hands are becoming more sophisticated. I don't think I'll live long enough to get a wireless Internet brain implant, but my kids or their kids might.

And then there are other things still further out, such as drugs and genetic and neural therapies to enhance our senses and strength. While we become more robotic, our robots will become more biological, with parts made of artificial and yet organic materials. In the future, we might share some parts with our robots.

We need not fear our machines because we, as human-machines, will always be a step ahead of them, the machine-machines, because we will adopt the new technologies used to build those machines right into our own heads and bodies. We're going to build our robots incrementally, one after the other, and we're going to decide the things we like having in our robots—humility, empathy, and patience—and things we don't, like megalomania, unrestrained ambition, and arrogance. By being careful about what we instill in our machines, we simply won't create the specific conditions necessary for a runaway, self-perpetuating artificial-intelligence explosion that runs beyond our control and leaves us in the dust.

When we look back at what we are calling the singularity, we will see it not as a singular event but as an extended transformation. The singularity will be a period in which a collection of technologies will emerge, mature, and enter our environments and bodies. There will be a brave new world of augmented people, which will help us prepare for a brave new world of AGIs. We will still have our emotions, intelligence, and consciousness.

And the machines will have them too. □



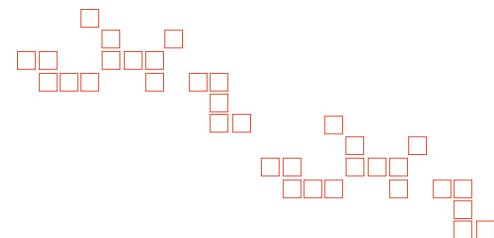
EXPERT VIEW: Esther Dyson

WHO SHE IS:

Commentator and evangelist for emerging technologies, investor and board member for start-ups; currently focused on health care, genetics, private aviation, and commercial space. Ran PC Forum conference until 2007; currently hosts the annual Flight School conference.

THOUGHTS

"The singularity I'm interested in will come from biology rather than machines. We won't be building things; we'll be growing and cultivating them, and then they will grow on their own."



review articles

DOI:10.1145/2629559

HACs offer a new science for exploring the computational and human aspects of society.

BY N.R. JENNINGS, L. MOREAU, D. NICHOLSON, S. RAMCHURN, S. ROBERTS, T. RODDEN, AND A. ROGERS

Human-Agent Collectives

THE COMPUTER HAS come a long way from its initial role as a scientific tool in the research lab. We live in a world where a host of computer systems, distributed throughout our physical and information environments, are increasingly implicated in our everyday actions. Computer technologies impact all aspects of our lives and our relationship with the digital has fundamentally altered as computers have moved out of the workplace and away from the desktop. Networked computers, tablets, phones and personal devices are now commonplace, as are an increasingly diverse set of digital devices built into the world around us. Data and information is generated at unprecedented speeds and volumes from an increasingly diverse range of sources and via ever more sensor types. It is then combined in unforeseen ways, limited only by human imagination. People's activities and collaborations are becoming ever more dependent upon and intertwined with this ubiquitous information substrate.

As these trends continue apace, it is becoming apparent that many endeavors involve the symbiotic interleaving of humans and computers. Moreover,

the emergence of these close-knit partnerships is inducing profound change. The ability of computer systems to sense and respond to our ongoing activities in the real world is transforming our daily lives and shaping the emergence of a new digital society for the 21st century. More specifically, rather than issuing instructions to passive machines that wait until they are asked before doing anything, we are now starting to work in tandem with highly interconnected computational components that act autonomously and intelligently (aka agents⁴²). This shift is needed to cope with the volume, variety, and pace of the information and services that are available.

It is simply infeasible to expect individuals to be aware of the full range of potentially relevant possibilities and be able to pull them together manually. Computers need to do more to proactively guide users' interactions based on their preferences and constraints. In so doing, greater attention must be given to the balance of control between people and machines. In many situations, humans are in charge and agents predominantly act in a supporting role, providing advice and suggesting options. In other cases, however, agents are in control and humans play the supporting role (for example, automatic parking systems on cars and algorithm-

» key insights

- HACs are a new class of socio-technical systems in which humans and smart software (agents) engage in flexible relationships in order to achieve both their individual and collective goals. Sometimes the humans take the lead, sometimes the computer does and this relationship can vary dynamically.
- There are major scientific challenges that must be addressed in developing systems that interact and motivate humans to work alongside agents in large, dynamic, and uncertain environments where privacy and ethical concerns may arise.
- Key research challenges in HACs include achieving flexible autonomy between humans and the software, and constructing agile teams that conform and coordinate their activities.



HCI: human-computer interaction
CSCW: Computer-supported Cooperative Work

mic trading on stock markets). Moreover, these relationships may change during the course of an activity (for example, a human may be interrupted by a more pressing request and so takes a less hands-on approach to the current task or an agent may encounter an unexpected situation and have to ask for human assistance for a task it was planning to complete autonomously).

We call this emerging class of systems *human-agent collectives* (HACs) to reflect the close partnership and the flexible social interactions between the humans and the computers. As well as exhibiting increased autonomy, such systems are inherently open and social. This openness means participants need to continually and flexibly establish and manage a range of social relationships. Thus, depending on the task at hand,

different constellations of people, resources, and information must come together, operate in a coordinated fashion, and then disband. The openness and presence of many distinct stakeholders, each with their own resources and objectives, means participation is motivated by a broad range of incentives—extrinsic (for example, money or tax-benefit), social or image motivation (for example, public accreditation or leader-board position) or intrinsic (for example, personal interest in a social cause, altruism, or hobby²)—rather than diktat. Moreover, once presented with such incentives, the stakeholders need to be evaluated and rewarded in ways that ensure they sustain behaviors beneficial to the system they partially form.³³

Embryonic examples of future HAC

systems where people routinely and synergistically interact and collaborate with autonomous software are starting to emerge. For example, as we travel, increasingly interconnected transport management systems cooperate to aid our journey. Systems such as Waze (<http://www.waze.com/>) blend citizen and (electronic-) sensor generated content to aid the user. Furthermore, software agents can proactively interact to arrange a place to stay and somewhere to eat in accordance with the traveler's preferences and current circumstances. However, despite relevant work on parts of the problem in the AI, HCI, CSCW and UbiComp communities, it is apparent that developing a comprehensive and principled science for HACs is a major research challenge, as is the process by which such systems can be

designed and built, and the means by which HACs will be accepted and deployed in the wild.

What Is Different about Human-Agent Collectives?

HAC systems exhibit a number of distinctive features that make it particularly challenging to engineer and predict their behavior. Their open nature means control and information is widely dispersed among a large number of potentially self-interested people and agents with different aims and objectives. The various system elements exhibit a range of availabilities; some are persistent and others are transient. The independent actors need to coordinate flexibly with people and agents that are themselves adapting their behaviors and actions to the prevailing circumstances to best achieve their goals. The real-world context means uncertainty, ambiguity, and bias are endemic and so the agents need to handle information of varying quality, trustworthiness, and provenance. Thus, techniques are required to provide an auditable information trail from the point of capture (a sensor or a human participant), through the fusion and decision processes, to the point of action, and the agents will have to reason about the trust and reputation of their collaborators to take the best course of action. Finally, in many cases, it is important that the collective action of the volitionally participating actors results in acceptable social outcomes (such as fairness, efficiency, or stability). When taken together, these HACs features require us to:

- 1 ▶ Understand how to provide flexible autonomy that allows agents to sometimes take actions in a completely autonomous way without reference to humans, while at other times being guided by much closer human involvement.
- 2 ▶ Discover the means by which groups of agents and humans can exhibit agile teaming and come together on an ad hoc basis to achieve joint goals and then disband once the cooperative action has been successful.
- 3 ▶ Elaborate the principles of incentive engineering in which the actors' rewards are designed so the actions the participants are encouraged to take generate socially desirable outcomes.
- 4 ▶ Design an accountable information infrastructure that allows the veracity

and accuracy of seamlessly blended human and agent decisions, sensor data, and crowd-generated content to be confirmed and audited.

A number of research domains are beginning to explore fragments of this overarching vision. However, none of them is dealing with the totality, nor the associated system-level challenges. For example, interacting intelligent agents are becoming a common means of designing and building systems that have many autonomous stakeholders, each with their own aims and resources.⁹ To date, much of this work has focused on systems where all the agents are either software or hardware (for example, robots or unmanned autonomous systems (UAS)). However, it is increasingly being recognized that it is both necessary and beneficial to involve humans, working as active information gatherers and information processors, in concert with autonomous software agents, within such systems.^{11,38} For example, systems have been demonstrated where humans gather real-world information and pass it to an autonomous agent that performs some basic aggregation before presenting it online.³⁰ Such approaches are often termed participatory sensing²⁶ or citizen-sensing.¹⁵ Likewise, a number of systems have been demonstrated in which autonomous agents pass information-processing tasks to the human participants, then collect and aggregate the results.⁴⁰ However, these broad strands of work typically assume the authority relations between humans and agents are fixed and that there exists a largely static set of skilled human participants who participate on a voluntary basis.^a This contrasts with the HAC view of agents operating within a dynamic environment in which the flexible autonomy varies the human-agent authority relationships in a context-dependent manner and in which these actors individually make decisions based on their preferences and the properties of their owners.

In the areas of HCI and CSCW, research has increasingly turned to the crowd and how to exploit computer

systems to harness and coordinate the endeavors of people.⁴¹ Essentially, the task has been to manage the means through which people are instructed and to coordinate their responses in a manner that makes sense. This large-scale, networked collaboration is typically achieved using software systems to coordinate and analyze these human endeavors. Moreover, software agents have emerged as a key technology for observing and reacting to human activities.¹⁷ This approach has also gained popularity with mixed-initiative systems⁸ and the development of context-aware computing approaches within ubicomp.¹ However, in most of this work, the software agents are a tool to aid in understanding and managing user interactions. Users are in the foreground and the agents are in the background. The challenge of HACs is moving from the presumption of a dominant relationship, to consider how users and agents coexist on a common footing and are considered in a manner that allows flexible relationships to emerge.

The role of people within HACs also brings the incentives of the participants to the fore. Most current systems largely assume altruistic and benevolent behavior, and have not dealt with the need to provide incentives to potentially self-interested participants, nor have they explicitly handled the inherent uncertainty of participatory content in a consistent manner (see Rahwan et al.²⁸ for examples of such behavior and Naroditskiy et al.²³ for the design of incentive structures to combat it). Similarly, current approaches to accountable information infrastructures have focused on augmenting specific systems, such as databases³ or computational workflows,⁷ with the ability to track information provenance. Now, emerging efforts, such as the W3C PROV Recommendation,²¹ are starting to allow for the tracking of provenance across multiple systems and to systems where confidentiality of data needs to be preserved.¹⁴ However, no work deals with the simultaneous challenges of humans in the loop and long-term and online operation.

Human-Agent Collectives in Action

Consider the aftermath of a major natural disaster. A number of organizations

a Amazon Mechanical Turk (AMT), and other similar systems, is an exception to this in that it allows software systems to automatically generate Human Intelligence Tasks and make payments to a large pool of human participants who complete them.

i.e. w/o humans. the system
fully autonomous mode

are in the area, including first responders (FRs), humanitarian aid organizations, and news reporters, as well as locals. A key aim for many of these actors is to assess the situation to determine the areas to focus on in the coming days and weeks. To assist in this task, a number of the organizations have UASs that can be used for aerial exploration and a number of locals have installed sensors to monitor the environment (for example, two weeks after the Fukushima incident, locals had built and deployed over 500 Geiger counter sensors and were uploading their readings; see <http://jncm.ecs.soton.ac.uk/>), in addition to many locals using social media platforms such as Ushahidi or Google Crisis Response to record requests for help and complete maps of the stricken area. A representative system architecture for this HAC is shown in the accompanying figure and an associated video is available at <http://vimeo.com/76205207>.

As can be seen, the information infrastructure contains a wide variety of content (for example, maps of roads and key amenities, weather forecasts, and social media reports from locals in the affected areas), from many sources. Some of these sources provide higher quality, more trustworthy information than others (for example, international aid organizations versus locally built environmental sensor readings). To help account for and justify the decisions that are made, the provenance of information is stored wherever possible. Moreover, the decisions made by both responders and autonomous agents (including UASs) are tracked to ensure all members of the HAC are accountable for their actions and the successes and failures of the rescue effort can be better understood when such data is reviewed at a later stage.

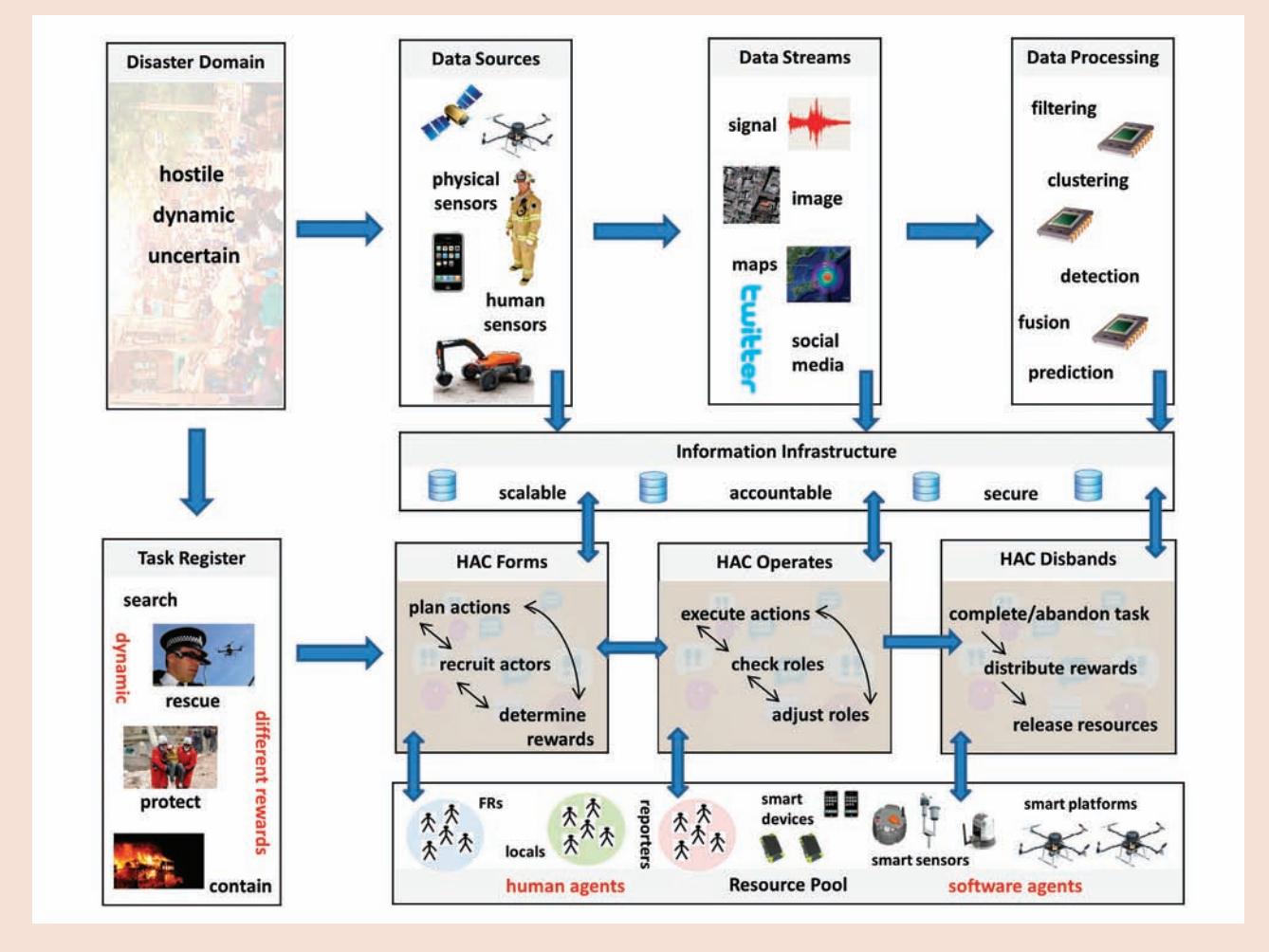
At the start of a day, the various actors (for example, FRs or local volunteers)

register their availability and relevant resources (for example, UAS, ground transport vehicles or medical supplies) and indicate specific tasks they would like to perform (for example, search the area near the school or determine if there is running water in a particular district). These tasks will be informed by their current assessment of the situation and may be influenced by particular requests from locals for assistance.

As a first step, some actor constructs a plan to achieve one or more of the tasks^b (that is, the HAC forms). This plan is likely to involve constructing teams of people, agents, and resources to work together on a variety of subtasks because many activities are likely to be beyond the capability of just one team

^b It may be a human or a software agent that proposes the initial plan. Moreover, multiple actors may attempt to construct plans simultaneously, some of which may not come to fruition.

HAC system for disaster response.



member. As they join, the various responders may accept the plan as is and be ready to start enacting it. However they may wish to make minor modifications (for example, putting in waypoints en route to the chosen area to maximize the value of the information obtained or requesting help from volunteers for some parts of the plan). Some may even desire to make more major modifications (for example, indicating that a particular subtask that has initially been excluded is more important than one of the suggested ones or that significant extra resources are needed for the plan to be successful). This plan co-creation iterates until agreement is reached, with various cycles of the humans taking charge and the agents replanning to account for the responders' preferences.

Due to the nature of the problem and environment, the HAC's plan execution (operation) phase may not go smoothly. New higher-priority tasks may appear, planned ones may turn out to be unnecessary, new actors and resources may become available, or committed ones may disappear (for example, due to FRs being exhausted or UASs running out of power). All of these will involve the agents and the humans in an ongoing monitoring and replanning endeavor, potentially involving the disbanding of existing teams and the coming together of new ones with different combinations of capabilities that are a better fit for certain types of rescue missions. Moreover, the autonomy relationship between the humans and the agents may change during the course of plan execution. For example, a team of UASs may initially be instructed to gather imagery from a particular area in an entirely autonomous fashion and not to disturb the FRs until the whole task is complete. However, in performing this task the UASs might run into difficulty and request assistance with a complex operation (for example, maneuvering in a tight space to get a particular view of a building) or the UASs might discover something important they believe is worth interrupting the responders for or worth requesting help from an online crowd to analyze the images collected. As unforeseen events happen and the complexity of the effort grows to involve hundreds of responders, volunteers, and UASs, the response coordinators

Different constellations of people, resources, and information must come together, operate in a coordinated fashion, and then disband.

may need to rely more on autonomous agents to compute plans and allocate resources and actions to individuals and UASs. Moreover, when the coordinators detect the agents are less able to judge human abilities (for example, because fatigue reduces human performance) or the nature of a task (for example, complex digging operations or surveillance), the coordinators may choose to step in to change the plan or input more information.

Over a number of days, responders may notice that locals are less willing to support the rescue effort and that those who do only work on tasks in their own areas. This means some areas do not receive sufficient attention. To help fill this gap, a number of additional incentive schemes could be put in place. First, volunteers could be rewarded if they sign up friends and family to assist with the rescue effort, such rewards could be financial (for example, payment for hours worked or vouchers for fuel) or nonfinancial (for example, promise of quicker return of their amenities). Second, individuals who complete tasks outside their local area and on high-priority tasks could receive additional credit, such as community service awards or double time pay. Finally, to encourage accurate reporting of the importance and urgency of tasks, an incentive structure that increases the reputation of individuals who make assessments that accord well with those of the professional responders may be introduced and bonus payments made to recruiters who hire the top task performers.

Key Research Challenges

HACs present research challenges in, among other things, how we balance control between users and agents (flexible autonomy), dynamically (dis)assembling collectives (agile teaming), motivating actors (incentive engineering) and how we provide an information infrastructure to underpin these endeavors. While none of these are entirely new areas, the HAC system context introduces additional complexity and brings new elements to the fore.

Flexible autonomy. HACs provoke fundamental questions about the relationship between people and digital systems that exhibit some form of autonomy. Specifically, the emergence of HACs highlights the growing extent to

which computer systems can no longer be thought of as entirely subservient. We routinely obey navigation systems without question or follow computer-generated instructions delivered on our phones. As such autonomous systems increasingly instruct us, new forms of relationship are emerging. This shift not only raises issues about how we might design for interaction within these systems, it also brings into focus larger social and ethical issues of responsibility and accountability.

HACs are fundamentally socio-technical systems. Relationships between users and autonomous software systems will be driven as much by user-focused issues (such as responsibility, trust, and social acceptability), as technical ones (such as planning or coordination algorithms). Consequently, we need to uncover the interactive principles of flexible autonomy that shape these systems. A critical issue here is the balance of control between the agents and the users involved in the collective. In particular, when do users need to be in control and when should software systems override them?

Given this, a key challenge is how to ensure a positive sense of control within HACs. Core to this issue is the sense of social accountability and responsibility inherent in our everyday activities. Our professional and personal actions are routinely available to others and we are held accountable for them. In fact, it could be argued that this shapes many of our broader societal relations and understandings. But how will we feel when we are sharing our world with computational elements that exert as much control over the environment as us? Will the relationship be an easy or a tense one and how might we manage that relationship? What checks and balances are needed to allow a fruitful relationship to mature between software agents and humans in HACs as the balance of control shifts between them?

Tackling these questions requires us to think about how software agents might reveal their work to users. At present, such software often operates “behind the scenes” with limited visibility to users. They might make a recommendation on our behalf or schedule activities with the result of their endeavors presented to users. However, little if anything is conveyed of the ratio-

nale leading to this result. In contrast, computational agents within HACs will need to make their actions and rationale available so they can be socially accountable.

Revealing the role and actions of software agents to users will bring into focus a raft of questions that require us to consider the broader social and ethical issues, potentially prompting significant reflection on the legal and policy frameworks within which these systems operate. For example, given the collective nature of the endeavor, it is important to determine who or what will ultimately be responsible for a particular outcome and what this might mean for the application of this approach. More routinely, to what extent will people allow the software agents in these collectives the trust and latitude they might give to other trained and qualified professionals? Will it be acceptable for software agents to make mistakes as they learn how to do a job as part of a collective?

A critical issue here is how to represent both human and agent endeavors in HACs, at multiple levels of scale and aggregation. Along with promoting a sense of social accountability, the ability to recognize and understand the activities of others and to flexibly respond to these actions is necessary to enable cooperation and to coordinate actions as part of broader social endeavors. Thus, the provision of mechanisms to make users aware of the actions of others is central to the design of many cooperative systems. In particular, the following are central issues: What mechanisms are needed to allow us to do this with users and autonomous software agents alike? How might we sense human actions and recognize the various activities a user is involved in and how might these be conveyed to software agents? What are the most appropriate techniques for presenting agents’ actions and ongoing progress to users?

Agile teaming. Humans and agents will form short-lived teams in HACs and coordinate their activities to achieve the various individual and joint goals present in the system before disbanding. This will be a continual process as new goals, opportunities and actors arrive. To date, research within the multi-agent systems community has generated a significant number of algorithms to

form and coordinate teams; specifically those algorithms found within the areas of coalition formation and decentralized coordination.^{27,32} However, many of these approaches focus on interactions between software agents alone and do not consider the temporal aspects of agile teaming.³⁶

In HAC settings, these assumptions are challenged. Centralized control is simply not possible for large scale dynamic HACs. Moreover, approaches must be developed to consider not just what the optimal coalition looks like, but how the individual humans and agents, each with their own limited communication and computation resources, can negotiate with one another to form a coalition, without having explicit knowledge of the utilities and constraints of all the other actors within the system.

Addressing the decentralization issues is likely to involve local message passing approaches that draw on insights from the fields of probabilistic inference, graphical models, and game theory. These allow coordination and coalition formation problems to be efficiently represented as a graph by exploiting the typically sparse interaction of the agents (that is, not every agent has a direct interaction with every other). To date, however, these approaches have only addressed teams with tens of actors, while HACs will scale to hundreds or possibly thousands. Similarly, extant approaches have been developed with the explicit assumption that all of the actors engaged in the coordination have similar computational and communication resources; an assumption that will almost certainly not be valid within most HACs. Addressing the challenge of scaling up these approaches, such that they can deal with large numbers of actors with heterogeneous computational and communication resources, is likely to require principled approximations to be made (see Rahwan et al.²⁹ for promising work in this vein that uses well founded network flow optimization algorithms to address large-scale coalition formation problems).

Furthermore, previous approaches to forming teams and coordinating actors have typically assumed that complete and accurate knowledge regarding the utilities and constraints of the system is available to all. Although such

assumptions may be valid within the small-scale systems studied to date, they will not apply to larger HACs operating within dynamic environments where the sensing and communication capabilities of the actors are unknown. Previous work in this domain has addressed such uncertainty through the frameworks of Markov decision processes (MDPs) and partially observable MDPs. Now, while the Bayesian framework implicit in these approaches is well founded and principled, again it does not currently scale sufficiently to allow its use in large systems where time-critical decisions must be made. As in the previous case, this requires novel computational and approximation approaches to be devised.

Most importantly, the novel approaches to HAC formation and operation must also address the needs of the humans within the system. Users will have to negotiate with software agents regarding the structure of the short-lived coalitions they will collectively form, and then coordinate their activities within the resulting coalition. This represents a major departure for the mechanisms and techniques that have predominantly focused on computational entities with little regard for the ways in which users might form teams or consideration of their relationship to groupings such as teams. Consequently, the computational exploration must be balanced by a focus on the persuasion and engagement of human participants within collectives. For example, how might we understand and manage the conflict between the need to break up and reform teams with natural human preferences for stability and trust? How might users feel about the possibility of working across multiple teams simultaneously, and how might they feel about taking instructions from software agents? Initial work has begun to explore these questions within the context of mixed reality games²⁰ and social robotics.⁶ However, to fully understand how these dynamics impact the requirements of the underlying team formation and coordination algorithms, we need to build, deploy, and evaluate prototypical HACs in realistic settings.

Incentive engineering. What will cause HACs to form and what will motivate them to work well together? How do we align the incentives of a set of actors, ei-

ther individually or as a group, with the goals of the system designers to generate particular outcomes? Both of these endeavors are challenging whenever the behavior of the actors is guided by individual and potentially conflicting motives.⁴ Now, while it is acknowledged that such actors may be influenced by incentives of many different types, most research to date has drawn upon microeconomics, focusing on monetary incentives and assuming the actors' utility functions are well defined and linear. Moreover, actors are typically considered to be rational in that they carry out complex computations to deduce their best action in equilibrium. Unfortunately, these assumptions often result in incentive mechanisms that are centralized and brittle in the context of open systems such as HACs.

HACs require us to reconsider many of the presumptions central to most current approaches to incentive engineering. They will involve actors that are boundedly rational¹⁰ and whose behavior cannot always be controlled. For example, in disaster response settings, local volunteers (of different reliabilities) can be co-opted by their family members and friends (through their social network) and need to be coordinated to work alongside emergency responders from different agencies (with different capabilities). Moreover, humans and software agents may not always be receptive to monetary payments and may react better to social or intrinsic incentives. For example, in the DARPA Red Balloon Challenge, the incentive mechanism of the winning team was successful because it aligned individual financial incentives to find balloons with that of recruiting members from their social network and beyond.²⁸ In contrast, essentially the same scheme attracted very few volunteers on the MyHeartMap Challenge (<http://www.med.upenn.edu/myheartmap/>), which appealed mostly to altruistic motivations to save heart attack patients. Recently, Scekic et al.³³ have surveyed and categorized such early examples of incentive mechanisms for social computing platforms. They note the vast majority of current systems employ a simple contest between workers, from which a subjective assessment of the "winner" is made. This winner is then typically rewarded financially for the work they have done. Far fewer

systems make use of non-monetary incentives. Those that do, often focus on reputational issues. For example, they cite avvo.com, which attracts large numbers of lawyers in the U.S. to provide free response and advice to people visiting the website, and generates a reputation ranking of these lawyers based on the quality and timeliness of responses they provide. Now, the online ranking can clearly impact the chances of attracting customers to their private practices, but the effect is indirect, and admits a greater range of personal motivation, compared to a direct monetary payment. To date, however, comparatively little work has attempted to formally define this style of incentive mechanism. This is an important omission because we believe that bringing insights from behavioral economics and "nudge" approaches to behavior change,^{10,38} into the formal descriptions of mechanism design provided by game theory, is a promising point of departure to engineer the types of incentives HACs require.

Now, even if the right incentives can be ascertained, the actors in the system may not perform to their best, either because they have limited capabilities or are inherently byzantine. For example, in systems such as AMT, where thousands of workers attempt micro-tasks for a few cents, strategic workers attempt to complete as many tasks as possible with minimum effort. Similarly, in citizen science projects such as Zooniverse, amateur scientists often select the tasks they most enjoy rather than those needed most for the project.³⁴ Moreover, in long-lived interactions, human actors may suffer from fatigue and therefore their performance may degrade over time. However, naïvely filtering out the actors that cannot perform some tasks may mean their ability to perform other tasks properly is wasted. Hence, it is crucial to design mechanisms to ensure the incentives given to the actors to perform the tasks assigned to them are aligned with their capabilities and reliabilities. Against this background, initial work in crowdsourcing, and citizen science has, for example, demonstrated how to set the price paid for micro-tasks or how many tasks, of a particular type, each actor should be allocated to incentivize them to perform well.³⁹ Gamification approaches have also been successful in incentivizing

human participants to spend hours doing what would typically be viewed as boring tasks.⁴¹ Nevertheless, more work is needed to generalize these approaches and prove their efficacy in different application areas.

While these challenges relate to how incentives are chosen and presented, the computation of these incentives in the context of HACs is also a major challenge. Indeed, the fact that HACs involve large numbers of actors means computationally efficient algorithms need to be designed to enumerate and optimize the, possibly combinatorial, incentives to be given to a crowd, coalition, or individual within a HAC. When the operation of such HACs unfolds over a long period of time, possibly involving many repeated interactions and negotiations, the schedule of incentives to be offered is an even greater computational challenge. In most cases, optimality will not be achievable and, hence, the goal should be to seek approximate solutions. Some relevant examples include algorithms to incentivize large numbers of electric vehicle owners to schedule charging their cars at different times to avoid overloading a local transformer³¹ and algorithms to approximate fair rewards for the participants of large teams.¹⁹

Accountable information infrastructure. HACs will have a significant impact on the ways in which we think about the digital infrastructure that supports them. Specifically, we need to consider how the data underpinning can be shared. The provenance of this information is particularly critical. Here, provenance describes which information data is derived from (what), the humans or agents responsible for it (who), and the methods involved in deriving it (how). In turn, the infrastructure processes provenance to assess information quality, to allow users to understand and audit the past behavior of HACs, and to help humans decide whether a HAC's decisions can be trusted.

The ways in which HACs operate requires us to reconsider some of the prevailing assumptions of provenance work. Provenance is generally thought of as being fine-grained, deterministic, and accurately and completely describing executions.²¹ This assumption is not valid in HACs, since human activities are both difficult to capture and unre-

A key challenge is how to ensure a positive sense of control within HACs. Core to this issue is the sense of social accountability and responsibility inherent in our everyday activities.

liable. Moreover, asynchronous communications may make provenance incomplete. Finally, the fine-grained nature of provenance makes it difficult for humans to understand. Addressing these challenges is crucial, and a variety of techniques are needed. For instance, probabilistic models built on provenance may help capture the uncertainty associated with what happened and abstraction techniques may allow common patterns to be collapsed, and thus, large graphs to be more manageable. These promising directions require the meaning of such provenance descriptions, and the kind of reasoning they enable, to be investigated. Given the potential size of HACs in terms of agents and humans, and also in terms of duration of execution, the scalability of reasoning algorithms is also an important issue that requires further work.

The vision for an accountable information infrastructure is to help both humans and agents understand the decisions made and determine whether they can be trusted. Indeed, it is folklore that provenance can help derive trust and assess quality, but no principled approach, readily applicable to HACs, is currently available. In this context, the ability to learn from provenance is important as it has the potential to make provenance a rich source of information to establish trust, and also guide decision-making in HACs. In particular, given that provenance information typically takes the form of a graph, some of the methods developed for graphs in general may be customizable, and potentially be executable efficiently. An example of such a solution is network metrics that summarize complex situations and behaviors in a convenient and compact way. Specifically, network metrics can be specialized to provenance graphs, helping characterize HACs' past behavior, in an application-agnostic manner.⁵ Then, by applying machine learning techniques to provenance-oriented network metrics, we can label graphs and nodes to derive trust about agents or quality of data.

To date, existing infrastructure mechanisms tend to embody a "middleware" perspective, formalizing data models, developing algorithms, and engineering the integration of facilities such as provenance with applications. However, HACs need to understand and

respond to the behavior of people and how this human activity is captured, processed, and managed raises significant ethical and privacy concerns. Often at the core of these concerns is the manner in which people are separated from data collected about them. Specifically, in current infrastructures people are often unaware of the digital information they bleed, how this information is processed, and the consequential effects of the analytical inferences drawn from this data. Consequently, people are at an ethical disadvantage in managing their relationship with the infrastructure as they are largely unaware of the digital consequences of their actions and have no effective means of control or withdrawal. A HAC infrastructure will need to be accountable to people, allowing them to develop a richer and more bidirectional relationship with their data.

Developing an accountable infrastructure also responds to the call from privacy researchers such as Nissenbaum²⁴ to understand and support the relationship between users and their data. Indeed, her Contextual Integrity theory frames privacy as a dialectic process between different social agents. Others have built upon this point, suggesting a bidirectional relationship needs to be embedded into the design of services so they are recognized as inherently social.³⁵ This suggests users should have a significant element of awareness and control in the disclosure of their data to others²⁵ and the use of this data by software agents. Establishing such bi-directional relationships also requires us to reframe our existing approaches to the governance and management of human data.

Perhaps the most critical issues in this regard relate to seeking permission for the use of personal data within information systems. Current approaches adopt a transactional model where users are asked at a single moment to agree to an often quite complex set of terms of conditions. This transactional model is already being questioned in the world of bio-ethics, with Manson and O'Neill¹⁸ arguing for the need to consider consent as much broader than its current contractual conception. We suggest that HACs will similarly need to revisit the design principles of consent and redress the balance of agency toward the users.¹⁶

References

1. Abowd, G.D., Ebling, M., Hung, G., Lei, H., and Gellersen, H.W. Context-aware computing. *IEEE Pervasive Computing* 1, 3 (2002) 22–23.
2. Ariely, D., Bracha, A. and Meier, S. Doing good or doing well? Image motivation and monetary incentives in behaving prosocially. *American Economic Review* 99, 1 (2007), 544–55.
3. Buneman, P., Cheney, J. and Vansumeren, S. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Systems* 33, 4 (2008), 1–47.
4. Dash, R.K., Parkes, D.C. and Jennings, N.R. Computational mechanism design: A call to arms. *IEEE Intelligent Systems* 18, 6 (2003) 40–47.
5. Ebdon, M., Huynh, T.D., Moreau, L., Ramchurn, S.D. and Roberts, S.J. Network analysis on provenance graphs from a crowdsourcing application. In *Proc. 4th Int. Conf. on Provenance and Annotation of Data and Processes*. (Santa Barbara, CA, 2012), 168–182.
6. Fong, T., Nourbakhsh, I. and Dautenhahn, K. A survey of socially interactive robots. *Robots and Autonomous Systems* 42 (2003), 143–166.
7. Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L. and Myers, J. Examining the challenges of scientific workflows. *IEEE Computer* 40, 12 (2007), 26–34.
8. Horvitz, E. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems* (New York, NY, 1999), 159–166.
9. Jennings, N.R. An agent-based approach for building complex software systems. *Commun. ACM* 44, 4 (Apr. 2001) 35–41.
10. Kahneman, D. Maps of bounded rationality: Psychology for behavioral economics. *American Economic Review* 93, 5 (2003) 1449–1475.
11. Kamar, E., Gal, Y. and Grosz, B. Modeling information exchange opportunities for effective human-computer teamwork. *Artificial Intelligence Journal* 195, 1 (2013) 528–555.
12. Kifor, T. et al. Provenance in agent-mediated healthcare systems. *IEEE Intelligent Systems* 21, 6, (2006) 38–46.
13. Krause, A., Horvitz, E., Kansal, A. and Zhao, F. Toward community sensing. In *Proc. Int. Conf. on Information Processing in Sensor Networks* (St Louis, MO, 2008), 481–492.
14. Luger, E. and Rodden, T. An informed view on consent for UbiComp. In *Proc. Int. Joint Conf. on Pervasive and Ubiquitous Computing* (2013), 529–538.
15. Maes, P. Agents that reduce work and information overload. *Commun. ACM* 37, 7 (1994), 31–40.
16. Manson, N.C. and O'Neill, O. Rethinking informed consent in bioethics. *CUP*, 2007.
17. Michalak, T., Aaditha, K.V., Szczepanski, P., Ravindran, B. and Jennings, N.R. Efficient computation of the Shapley value for game-theoretic network centrality. *J. AI Research* 46 (2013), 607–650.
18. Moran, S., Pantidi, N., Bachour, K., Fischer, J.E., Flintham, M. and Rodden, T. Team reactions to voiced agent instructions in a pervasive game. In *Proc. Int. Conf. on Intelligent User Interfaces*, (Santa Monica, CA, 2013), 371–382.
19. Moreau, L. The foundations for provenance on the Web. *Foundations and Trends in Web Science* 2, 2–3 (2010) 99–241.
20. Moreau, L. et al. Prov-dm: The prov data model. W3C Recommendation REC-prov-dm-20130430, World Wide Web Consortium, 2013.
21. Naroditsky, V., Rahwan, I., Cebrian, M. and Jennings, N.R. Verification in referral-based crowdsourcing. *PLoS ONE* 7, 10 (2012) e45924.
22. Nissenbaum, H. Privacy as contextual integrity. *Washington Law Review* 79, 1 (2004), 119–158.
23. Palen, L. and Dourish, P. Unpacking privacy for a networked world. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2003), 129–136.
24. Paxton, M. and Benford, S. Experiences of participatory sensing in the wild. In *Proc. 11th Int. Conf. on Ubiquitous Computing*, (Orlando, FL, 2009), 265–274.
25. Rahwan, T., Ramchurn, S.D., Jennings, N.R. and Giovannucci, A. An anytime algorithm for optimal coalition structure generation. *J. Artificial Intelligence Research* 34 (2009), 521–567.
26. Rahwan, T. et al. Global manhunt pushes the limits of social mobilization. *IEEE Computer* 46, 4 (2013a) 68–75.
27. Rahwan, T., Nguyen, T-D, Michalak, T., Polukarov, M., Croitoru, M., Jennings, N.R. Coalitional games via network flows. In *Proc. 23rd Int. Joint Conf. on Artificial Intelligence*, (Beijing, China, 2013b), 324–331.
28. Reddy, S., Parker, A., Hyman, J., Burke, J., Estrin, D. and Hansen, M. Image browsing, processing, and clustering for participatory sensing. In *Proc. 4th Workshop on Embedded Networked Sensors*, (Cork, Ireland, 2007), 13–17.
29. Robu, V., Gerding, E.H., Stein, S., Parkes, D.C., Rogers, A. and Jennings, N.R. An online mechanism for multi-unit demand and its application to plug-in hybrid electric vehicle charging. *J. Artificial Intelligence Research* (2013).
30. Rogers, A., Farinelli, A., Stranders, R. and Jennings, N.R. Bounded approximate decentralised coordination via the max-sum algorithm. *Artificial Intelligence* 175, 2 (2011), 730–759.
31. Scelik, O., Truong, H.-L. and Dustdar, S. Incentives and rewarding in social computing. *Commun. ACM* 56, 6 (2013), 72–82.
32. Simpson, E., Roberts, S.J., Smith, A. and Lintott, C. Bayesian combination of multiple, imperfect classifiers. In *Proc. 25th Conf. on Neural Information Processing Systems*, (Granada, Spain, 2011).
33. Steeves, V. Reclaiming the social value of privacy. *Lessons from the Identity Trail*. I. Kerr, V. Steeves and C. Luccock, eds. Oxford University Press, 2009, 193–208.
34. Stone, P., Kaminka, G.A., Kraus, S. and Rosenschein, J.S. Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *Proc. 24th Conference on Artificial Intelligence* (2010).
35. Tambe, M. et al. Conflicts in teamwork: Hybrids to the rescue" In *Proc. 4th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, (Utrecht, The Netherlands, 2005), 3–10.
36. Thaler, R.H. and Cass, R.S. *Nudge: Improving Decisions About Health, Wealth, and Happiness*. Yale University Press, 2008.
37. Tran-Thanh, L., Venanzi, M., Rogers, A. and Jennings, N.R. Efficient budget allocation with accuracy guarantees for crowdsourcing classification tasks. In *Proc. 12th Int. Conf. on Autonomous Agents and Multi-Agent Systems* (St Paul, MN, 2013), 901–908.
38. von Ahn, L. et al. recaptcha: Human-based character recognition via web security measures. *Science* 321, 5895 (2008), 1465–1468.
39. von Ahn, L. and Dabbish, L. Labeling images with a computer game. In *Proc. STGCHI Conf. on Human Factors in Computing Systems*, (Vienna, Austria, 2004), 319–326.
40. Wooldridge, M.J. and Jennings, N.R. Intelligent agents: Theory and practice. *Knowledge Engineering Review* 10, 2 (1995) 115–152.

N.R. Jennings (nrj@ecs.soton.ac.uk) is the Regius Professor of Computer Science in Electronics and Computer Science at the University of Southampton University, U.K., and a chief scientific adviser to the U.K. government.

L. Moreau (lavr@ecs.soton.ac.uk) is a professor of computer science, head of the Web and Internet Science group (WAIS), and deputy head (Research and Enterprise) of Electronics and Computer Science at the University of Southampton, U.K.

D. Nicholson (dn4@ecs.soton.ac.uk) is a senior industrial scientist with BAE and a knowledge transfer officer for the EPSRC-funded Human Agent Collectives project.

S. Ramchurn (sdr@ecs.soton.ac.uk) is a lecturer in the Agents, Interaction, and Complexity Group (AIC), Electronics and Computer Science at the University of Southampton, U.K.

S. Roberts (sjrob@robots.ox.ac.uk) leads the Machine Learning Research Group at Oxford, U.K. He is also a Professorial Fellow of Somerville College and a faculty member of the Oxford-Man Institute.

T. Rodden (tar@cs.nott.ac.uk) is a professor of computing at the University of Nottingham and co-director of the Mixed Reality Laboratory.

A. Rogers (acr@ecs.soton.ac.uk) is a professor of computer science in the Agents, Interaction and Complexity Research Group in the School of Electronics and Computer Science at the University of Southampton, U.K.

Copyright held by owners/authors. Publication rights licensed to ACM. \$15.00

Offline Planning with Hierarchical Task Networks in Video Games

John-Paul Kelly

Department of Engineering
Australian National University
Canberra, ACT

Adi Botea

NICTA and
Australian National University
Canberra, ACT

Sven Koenig

Computer Science Department
University of Southern California
Los Angeles, CA

Abstract

Artificial intelligence (AI) technology can have a dramatic impact on the quality of video games. AI planning techniques are useful in a wide range of game components, including modules that control the behavior of fully autonomous units. However, planning is computationally expensive, and the CPU and memory resources available to game AI modules at runtime are scarce. Offline planning can be a good strategy to avoid runtime performance bottlenecks.

In this work, we apply hierarchical task network (HTN) planning to video games. We describe a system that computes plans offline and then represents them as game scripts. This can be seen as a form of generating game scripts automatically, replacing the traditional approach of composing them by hand. We apply our ideas to the commercial game THE ELDER SCROLLS IV: OBLIVION, with encouraging results. Our system generates scripts automatically at a level of complexity that would require a great human effort to achieve.

Introduction

Artificial intelligence (AI) can have a dramatic impact on the quality of video games. Non-playing characters (NPCs) are examples of entities that are under the control of the computer. Traditionally, NPCs were used with the only goal to populate the background of game worlds, and their behavior was limited to basic animations, with little or no interaction with the rest of the game world. In contrast, intelligent NPCs should act according to meaningful plans and interact with each other. AI techniques, such as search and planning, can make the behavior of NPCs look intelligent. However, planning is computationally expensive. Plans have to be available in real time, and the CPU and memory resources available to game AI modules at runtime are limited. A typical approach for achieving intelligent NPC behavior is therefore to use scripts, that is, plans that are composed offline by hand and cached to be used at runtime. A significant advantage of scripts is that they can then be used at runtime with little CPU and memory overhead. However, many scripts need to be generated by hand to cover a reasonable range of situations but they require lots of human effort to generate and the process is error-prone.

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

We address this issue by presenting a system that uses an offline planning approach for the automated generation of scripts. Our system guides planning with hierarchical task networks (HTNs) (Sacerdoti 1975), hand-coded structures that encode knowledge about the domain. While scripts are plans, HTNs correspond to many plans because they abstract details of specific planning problems away. Planning takes as input an HTN and information about the specific planning problem and outputs a plan. Designing either scripts or HTNs can be a time-consuming task. Fortunately, HTNs can be reused to generate many scripts within one game and potentially also across games. Our system can be integrated into games that implement scripting functionality. Our system converts plans from the plan-representation language, such as PDDL, into the scripting language accepted by the game. The resulting scripts are then handled by the game engine just like regular scripts. Our system currently handles only linear scripts (as opposed to conditional tree-shaped scripts) due in part to the availability of more advanced planning technology for sequential planning as compared to contingency planning. We apply our ideas to Bethesda Softworks' THE ELDER SCROLLS IV: OBLIVION, a popular commercial game, with encouraging results. Our system generates scripts automatically at a level of complexity that would require a great human effort to achieve.

Related Work

Scripts and finite state machines are traditional approaches for controlling the behavior of NPCs in video games. Finite state machines have been used from first-person shooters, such as ID Software's QUAKE series, to real-time strategy games, such as Blizzard Software's WARCRAFT III. Some recent games, such as Bungie's HALO 2, use hierarchical finite state machines (Isla 2005). However, the complexity of using finite state machines grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows. Their use is therefore considered problematic in current generation games (Orkin 2003). Scripts have been used from role-playing games, such as Bioware's NEVERWINTER NIGHTS, to first-person shooters, such as Epic Games' UNREAL TOURNAMENT. Scripts are composed offline in a high-level game-specific language and have similar properties as finite state machines. In particular, the complexity of using scripts grows quickly when

the behavior of NPCs and the world become more complicated or the number of NPCs grows, which makes it attractive to generate them semi-automatically. SCRIPTEASE, for example, aims at simplifying the process of writing scripts through pattern templates (McNaughton *et al.* 2004). It allows game developers to create relatively complicated behavior via a graphical user interface (GUI), without having to program them explicitly. The game developers must still manually choose for each NPC which behavior to initiate and when to initiate them, which means that the complexity of using SCRIPTEASE still grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows.

AI planning techniques have recently been used in games to control NPC opponents in first-person shooters, such as Epic Games' UNREAL TOURNAMENT or Monolith's F.E.A.R. Simple behavior (which does not need planning) was acceptable in early games, such as moving between ammunition caches and chasing all encountered opponents. More complicated behavior (which needs planning) is expected in modern games, such as moving as coordinated squads and performing tactics (including flanking or sending for backup) (Orkin 2006). While on-line planning might be able to handle dynamic environments better than off-line planning, it suffers from the problem that planning is computationally expensive. Plans have to be available in real time, and the CPU and memory resources available to game AI modules at runtime are limited. Off-line planning generates plans off-line and then uses them at runtime with little CPU and memory overhead. It is easier to implement than online planning and requires no changes to the game software. Off-line plans are typically generated by hand and then represented as game scripts. For example, an NPC might go every day for lunch at 12pm and then work until 4pm, resulting in realistic behavior. However, many scripts need to be generated by hand to cover a reasonable range of situations, they require lots of human effort to generate, the process is error-prone, and it is difficult to integrate new NPCs into a game. For example, if an NPC needs to buy merchandise from a store, then the NPC owner of the store needs to be there. Thus, game designers need to study the scripts for all NPCs to make sure that a change does not cause conflicts. The complexity of manual planning therefore grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows. We address this issue in the following by describing a system that uses SHOP2 (Nau *et al.* 2003), a modern HTN planner, to automatically compute plans offline and then represents them as game scripts. HTN planners were devised in the 1970s (Sacerdoti 1975) and have been advocated as knowledge-based planning approach for complex applications (Wilkins and desJardins 2001), such as production line scheduling (Wilkins 1988) and the game of bridge (Smith *et al.* 1998).

Our Testbed

We use Bethesda Softworks' THE ELDER SCROLLS IV: OBLIVION, a role-playing game, as testbed for our system. In this game, each NPC can be assigned a set of AI pack-

ages and a script. AI packages implement atomic behavior, such as eating, working and sleeping, and have activation preconditions that can include time ranges. For example, a sleep package could be active from 10pm to 8am. Setting the activation preconditions of appropriate AI packages corresponds to manual planning and does not require scripts, which are repeatedly executed in a loop and mostly used for dialogue and quest-related activities.

Our System

In our system, the game developers design an HTN that encodes knowledge of the game world, create a set of AI packages for all actions of the HTN, and specify several planning problems together with their initial world states. Each planning problem addresses the daily planning objectives of all NPCs. The planner is run on each planning problem. It takes as input the HTN, the initial world state and the planning problem and outputs a plan for all NPCs. The planner cannot plan for each NPC separately since NPCs can interact. The script generator converts the plan, whose actions correspond to AI packages, into a script for each NPC. Thus, we extend the use of scripts to encode daily plans of NPCs. The resulting scripts are then handled by the game engine just like regular scripts. We now describe the various components in turn.

The Hierarchical Task Network

The planning problem is encoded as an abstract task in HTN planning. Plans are action sequences that are refinements of the abstract task. HTN methods encode how abstract tasks (ovals) can be repeatedly refined into subtasks and eventually into actions (rectangles). When an abstract task is refined, the available methods are tried from left to right until a method is found whose preconditions match the current world state. A world state contains information such as a list of all locations, a list of all stores, a list of items traded in each store, the owner of each store, the number of deer that populate the game, and the NPC state of each NPC. An NPC state contains information such as the wealth, hunger and tiredness level of the NPC, its current activity and location, its inventory, its house, and a list of actions that it can perform.

Part of our HTN for THE ELDER SCROLLS IV: OBLIVION is illustrated in Figure 1. It encodes plans for a game day, divided into 24 game hours. If desired, plans that cover several game days can be obtained by chaining plans together that correspond to one game day each, by enforcing that the final world state of a plan is equal to the initial world state of the next plan. The picture on top illustrates four alternative methods that refine the abstract task of spending one game hour. If the preconditions of the first three methods are not satisfied, then an action is chosen at random, to avoid NPCs being idle. The picture at the bottom illustrates four alternative methods that refine the abstract task of obtaining food, namely directly by hunting or shopping or indirectly by learning how to hunt or making money, which will help one to obtain food in the future. The precondition of shopping is that the shop is open. After the NPC has waited

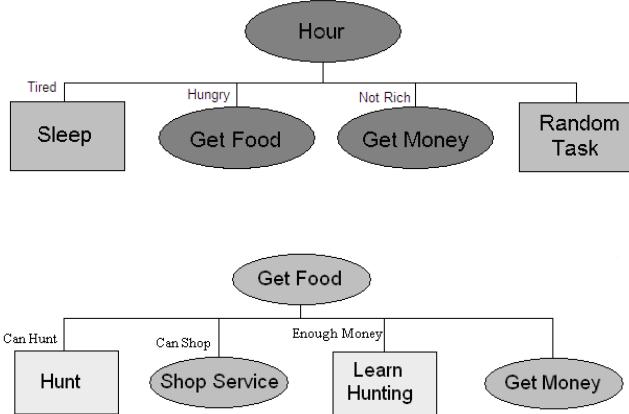


Figure 1: Part of the HTN.

for a sufficiently long time at the store, the precondition is considered to be unsatisfied and the next method is tried.

The Planner

We use JSHOP2 (Nau *et al.* 2003), an off-the-shelf Java implementation of SHOP2 that is publicly available and can handle numerical variables, which is important to model aspects of world states such as hunger levels.

The Script Generator

The script generator converts the linear plans (action sequences) from the plan-representation language PDDL into the scripting language used by THE ELDER SCROLLS IV: OBLIVION. Scripts check that the initial world state of a game day matches the initial world state of their plan. Scripts activate AI packages to implement actions. The AI packages thus have no preconditions. Each script uses conditional statements to ensure that the actions of the NPC are performed to completion, in the correct game hour and in the correct order since the game repeatedly executes each script in a loop. These conditional statements use variables that keep track of the number of actions performed during the current game hour as well as the last game hour that had all of its actions completed. Each script uses flag-based message passing to encode the exchanges of goods that occur between NPCs. The script of an NPC sends a request message to the script of another NPC to start an exchange. The script of the second NPC performs its part of the transaction and then sends a confirmation message. The script of the first NPC then performs its part of the transaction.

Example

We use a small example with three NPCs named Bob, Stan and Fred to illustrate parts of our system. Stan and Fred own one store each and thus interact with Bob via the exchanges of goods. For simplicity, we focus our attention on Bob for a few game hours and do not present all details.

```
(cost food 5.0)          (cost skins -4.0)
(cost goods -50.0)        (cost learn-hunt 30.0)
(cost rent 1.0)           (shop skin-shop skins Stan)
(shop skin-shop goods Stan) (shop food-shop food Fred)
(place Bobs-house)        (place forest)
(place skin-shop)         (place food-shop)
(deer 4.0)                (money Bob 7.0)
(asleep Bob)               (hungry Bob 7.0)
(hungry Bob 7.0)           (sleepy Bob 4.0)
(have-goods Bob)          (at Bob Bobs-house)
(lives Bob Bobs-house)    (available-task Bob purchase-food sedate-hunger)
(available-task Bob wander wander)
(available-task Bob skin get-money)
(available-task Bob sleep sleep)
(available-task Bob sell-goods get-money)
```

Figure 2: Part of the initial world state.

```
(!sleep Bob)
(!increment-person Bob)
(!wake Bob)
(!move Bob food-shop)
(!wait Bob food-shop)
(!increment-person Bob)
(!transaction Fred Bob Food)
(!eat Bob)
(!increment-person Bob)
(!move Bob forest)
(!skin Bob)
(!move Bob skin-shop)
(!transaction Stan Bob Skins)
```

Figure 3: Part of a plan.

```
if (theTime >= 10 && theTime < 11 && finished != 10)
  if current == 0
    AddScriptPackage TravelDeerForest
    set current to current + 1
  if current == 1
    if bobVar != 60
      set bobVar to 60
      set skinning to 1
      set skinDeer to 0
      AddScriptPackage Skin
  if current == 2
    AddScriptPackage TravelSkinShop
    set current to current + 1
  if current == 3
    if bobVar != 70
      set bobVar to 70
      set selling-skin to 1
      AddScriptPackage SellSkins
  if current == 4
    set current to 0
    set finished to 10
    set bobVar to 0
```

Figure 4: Part of a script.

Figure 2 shows part of the initial world state but skips the NPC states of Fred and Stan. The planning problem is $((day\ Fred\ Stan\ Bob))$. Figure 3 shows a few steps of the resulting plan for all three NPCs over a game day but skips actions that do not involve Bob. Bob sleeps, then wakes up and goes to buy food from Fred’s store. (He can satisfy his hunger this way since he has money.) Bob waits for the store to open. Fred shows up before Bob gives up, allowing Bob to buy the food and eat. Bob walks to forest to skin a deer and then goes to sell its pelt at Stan’s store (to make some money). The repeated call `!increment-person` makes Bob more tired and hungry over time.

Figure 4 shows part of the script that corresponds to the last four actions of the plan in Figure 3 although a detailed explanation of the code is beyond the scope of this paper. The first line ensures that each action of the plan is executed at the appropriate time. The variable `current` ensures that the actions are performed in the correct order by keeping track of how many actions have been completed in the current game hour. The variable `bobVar` ensures that each action is performed only once (although we skip its implementation details). The AI package `Skin` implements the action of killing and skinning a deer. The script of Bob sets the variable `skinning` to one. The script of the deer makes a pelt appear and increments the variable `current`, which allows Bob to execute the next action. Bob cannot increment the variable himself since he does not know exactly when his action completes. The AI package `SellSkins` initiates the sale of the pelt. The script of Bob sets the variable `selling-skin` to one. The script of Stan then performs the transaction and increments the variable `current`, which allows Bob to execute the next action. The last section of the script ensures that Bob does not execute the actions again once he has completed them.

Discussion

We now discuss the advantages and disadvantages of our system.

First, automated planning has advantages over manual planning. We have already argued that the complexity of manual planning grows quickly when the behavior of NPCs and the world become more complicated or the number of NPCs grows. Automated planning increases the game development speed and thus allows our system to compute more complex plans, which cover longer game periods and have more interactions between NPCs, resulting in a better gaming experience. Automated planning requires game developers to encode knowledge of the game worlds, in our case in form of HTNs. This should not be a problem for game developers since HTNs are similar to scripts. It is an open question whether game users, a community which is very large and heterogeneous in terms of programming skills, would welcome our system to add user-created content to off-the-shelf games. Scripting has gotten end users accustomed to using structured languages, and HTNs are not harder to understand than current sophisticated forms of scripts, such as conditional scripts. If desired, GUIs could be designed to help game developers and end users generate HTNs. Eventually, a standardized system similar to ours might get used

across games, similar to current graphics engines, networking code or physics simulators.

Second, offline automated planning has advantages over online automated planning. We have already argued that online planning is computationally expensive, and the CPU and memory resources available to game AI modules at runtime are scarce because sophisticated graphics and sound algorithms need them. Offline planning avoids this overhead and thus allows our system to compute more complex plans, which cover longer game periods and have more interactions between NPCs. Offline planning is less versatile than online planning since offline plans need to account for many contingencies and could thus be both difficult to compute and of large size in game worlds with incomplete or uncertain information (such as dynamic or partially observable domains). In contrast, online planning can make assumptions and replan when the current plan can no longer be executed. Our game world, however, can be modeled as completely observable and static by giving up only a small amount of realism, since the human player is the only unknown. For example, one can put virtual laws in place that prevent the human player from interfering with the NPCs to make their plans unexecutable, for example, by preventing the human player from killing NPCs that are vital for plan execution. Thus, offline planning is often sufficiently powerful, which is one of the main reasons why scripting, a form of offline planning traditionally performed by hand, is so popular in modern games.

Third, interfacing an offline automated planner via a scripting interface to a game has advantages. We have already argued that this makes offline planning easier to integrate into games than online planning since it requires no changes to the game software. This is an important issue in an industry with tight deadlines, where game companies tend to ignore new ideas unless they can easily be added to the game currently under development (Hopson 2006).

Comparison with SCRIPEASE

To the best of our knowledge, the work on SCRIPEASE (McNaughton *et al.* 2004) is the only research on automating script generation previously reported in the games literature. SCRIPEASE is a tool that allows a user to generate scripting code through a graphical front end. Scripting can be performed even by users with no programming skills. The process is faster and safer, since manual coding introduces bugs that can be hard to fix.

As an important difference from our work, SCRIPEASE makes no attempt at automating the planning side of scripting. The user is responsible to select all actions that compose a script, possibly starting from a predefined pattern. Arguably, SCRIPEASE could be seen as focused on the software engineering side of scripting rather than the AI side. The authors state that AI capabilities, such as learning, would be a good addition to the SCRIPEASE tool (McNaughton *et al.* 2004). In addition, we believe that our work and SCRIPEASE can be combined into a tool that would provide access to a planning engine through a sophisticated and intuitive GUI that would not require users to have deep planning knowledge. The GUI and all the machinery behind

NPCs	Time (sec.)	Nodes	Plan Length
3	6.12	1108	293
4	6.49	1452	381
5	7.07	1798	469
6	7.56	2142	557
7	7.94	2486	645
15	10.83	6332	1351
20	12.47	8062	1797
40	28.57	14982	3566

Table 1: Summary of results for 24 game hours when the problem size varies from 3 to 40 NPCs.

Game Hours	Time (sec.)	Nodes	Plan Length
4	12.31	2634	700
8	15.80	4570	1416
12	17.49	6566	2134
16	20.45	9086	2857
20	24.01	11126	3591
24	25.32	13614	4308

Table 2: Summary of results for 40 NPCs when the covered time period varies from 4 to 24 game hours.

it could be used to design HTNs and express the initial world state and the abstract task to be refined into a plan. Based on such input data, the planning engine could compute plans and translate them to the scripting language.

Evaluation

We ran two experiments with the HTN described earlier, which consists of 12 abstract tasks, 19 actions and 40 methods and requires 180 lines of code. The initial world state for 3 NPCs over 24 game hours uses 49 boolean atoms.

In our first experiment, we varied the problem size by increasing the number of NPCs from 3 to 40. Each plan covered a period of 24 game hours. In the second experiment, we varied the covered time period by increasing the number of game hours from 4 to 24. Each plan was for 40 NPCs. The two experiments were run on a PC with an AMD Athlon 64 X2 4200+ processor and 1 GB RAM. The initial NPC states were generated randomly. Tables 1 and 2 summarize the results of both experiments. They show the planning effort (measured in both CPU time and number of nodes) and the length of the produced plans.

The data in Tables 1 and 2 show that the planning effort grows at most linearly in both the number of NPCs and the number of game hours, which is evidence that the HTN guides planning well. Note that the numbers reported in the last row of each table are different, even though the number of NPCs and the number of game hours are the same in each case. This is because the problem sets for the two experiments were generated independently of each other. The NPCs tend to sleep slightly more in the plans computed in the first experiment, and agents that are asleep execute only one sleep action during several game hours while agents

that are awake typically execute several actions during each game hour. As a result, plans where NPCs sleep more tend to be shorter.

The size of each script for 40 NPCs varies between 80 and 500 lines when the number of game hours varies from 4 to 24. The plans are quite long but contain adjustment steps for state variables that can be compiled out by the script generator. For example, a script does not need to represent how hungry an NPC is. It needs to execute only the eating action. The runtime of the script generator is less than 30 seconds for 40 NPCs over 24 game hours. Manual planning and writing scripts of 500 lines for all 40 NPCs, in comparison, would be extremely tedious and time-consuming.

We have observed that automated planning makes the NPCs behave more realistically because there is more variation in their daily routine and both more and more meaningful interactions among them. Manual planning, for example, makes it attractive for the NPC owner of a store to keep its store open exactly from 9am to 5pm each day since such rigid rules make manual planning easier. Automated planning, in contrast, makes it easy for the NPC owner of a store to leave early if it is tired. Even though this action can influence the actions of other NPCs in complicated ways, the planner can compute a plan that will handle all details correctly. In addition, automated planning chains the actions of NPCs in meaningful ways. For example, an NPC attempts to buy food only if it has enough money available. In contrast, NPCs in the original game get food regardless of the amount of money they have.

Adding a new NPC to a game requires only slight modifications to the HTN and the initial world state. For example, one needs to add the NPC state for the new NPC to the initial world state. Similarly, changing the behavior of an NPC requires only slight modifications to the initial world state. For example, one needs to change the list of actions that the NPC can perform in its NPC state. In our experience, the editing takes only a couple of minutes in both cases before one can run the planner and script generator again.

Conclusion

Planning is a powerful but potentially resource-intensive technology to add intelligent behavior to NPCs in video games. In this paper, we have introduced an approach to offline HTN planning in the domain of video games. Our implementation generates scripts automatically, replacing the traditional approach of creating scripts manually. We applied our ideas to *THE ELDER SCROLLS IV: OBLIVION*, a popular role-playing game from Bethesda Softworks. In experiments, scripts are generated at a level of complexity that would require a great human effort to compose and debug. Future work includes applying offline conformant planning and offline contingent planning to games.

Acknowledgments

John-Paul Kelly was a summer student when this research was performed, and Sven Koenig was on sabbatical at NICTA. NICTA is funded by the Australian Government's Department of Communications, Information Technology,

and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

References

- R. P. Goldman and M. S. Boddy. Expressive Planning and Explicit Knowledge. In *Proceedings of AIPS-96*, pages 110–117, 1996.
- J. Hopson. We're Not Listening: An Open Letter to Academic Game Researchers, 2006. gamasutra.com/features/20061110/hopson_01.html.
- D. Isla. Handling Complexity in the Halo 2 AI. In *Proceedings of GDC-05*, 2005.
- M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. ScriptEase: Generative Design Patterns for Computer Role-Playing Games. In *Proceedings of IEEE ASE-04*, pages 88–99, 2004.
- D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *JAIR*, 20:379–404, 2003.
- J. Orkin. Constraining Autonomous Character Behavior with Human Concepts. In *AI Game Programming Wisdom 2*, pages 189–198, 2003.
- J. Orkin. Three States and a Plan: The A.I. of F.E.A.R. In *Proceedings of GDC-06*, 2006.
- M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proceedings of AIPS-92*, pages 189–197, 1992.
- E. Sacerdoti. The Nonlinear Nature of Plans. In *Proceedings of IJCAI-75*, pages 206–214, 1975.
- S. Smith, D. Nau, and T. Throop. Computer Bridge: A Big Win for AI Planning. *AI Magazine*, 19(2):93–105, 1998.
- D. Wilkins and M. desJardins. A Call for Knowledge-Based Planning. *AI Magazine*, 22(1):99–115, 2001.
- D. Wilkins. *Practical Planning: Extending the Classical Planning Paradigm*. Morgan Kauffman, 1988.

Current Trends in Automated Planning

Dana S. Nau

■ Automated planning technology has become mature enough to be useful in applications that range from game playing to control of space vehicles. In this article, Dana Nau discusses where automated-planning research has been, where it is likely to go, where he thinks it should go, and some major challenges in getting there. The article is an updated version of Nau's invited talk at AAAI-05 in Pittsburgh, Pennsylvania.

In ordinary English, *plans* can be many different kinds of things, such as project plans, pension plans, urban plans, and floor plans. In automated-planning research, the word refers specifically to *plans of action*:

representations of future behavior ... usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents.¹

One motivation for automated-planning research is theoretical: planning is an important component of rational behavior—so if one objective of artificial intelligence is to grasp the computational aspects of intelligence, then certainly planning plays a critical role. Another motivation is very practical: plans are needed in many different fields of human endeavor, and in some cases it is desirable to create these plans automatically. In this regard, automated-planning research has recently achieved several notable successes such as the Mars Rovers, software to plan sheet-metal bending operations, and Bridge Baron. The Mars Rovers (see figure 1) were controlled by planning and

scheduling software that was developed jointly by NASA's Jet Propulsion Laboratory and Ames Research Center (Estlin et al. 2003). Software to plan sheet-metal bending operations (Gupta et al. 1998) is bundled with Amada Corporation's sheet-metal bending machines such as the one shown in figure 2. Finally, software to plan declarer play in the game of bridge helped Bridge Baron to win the 1997 world championship of computer bridge (Smith, Nau, and Throop 1998).

The purpose of this article is to summarize the current status of automated-planning research, and some important trends and future directions. The next section includes a conceptual model for automated planning, classifies planning systems into several different types, and compares their capabilities and limitations; and the Trends section discusses directions and trends.

Conceptual Model for Planning

A conceptual model is a simple theoretical device for describing the main elements of a problem. It may fail to address several of the practical details but still can be very useful for getting a basic understanding of the problem. In this article, I'll use a conceptual model for planning that includes three primary parts (see figure 4a and 4b, which are discussed in the following sections: a *state-transition system*, which is a formal model of the real-world system for which we want to create plans; a *controller*, which performs actions that change the state of the system; and a *planner*, which produces the plans or policies that drive the controller.



Figure 1. One of the Mars Rovers.

State-Transition Systems

Formally, a *state-transition system* (also called a *discrete-event system*) is a 4-tuple $\Sigma = (S, A, E, \gamma)$, where

$S = \{s_0, s_1, s_2, \dots\}$ is a set of states;

$A = \{a_1, a_2, \dots\}$ is a set of *actions*, that is, state transitions whose occurrence is controlled by the plan executor;

$E = \{e_1, e_2, \dots\}$ is a set of *events*, that is, state transitions whose occurrence is not controlled by the plan executor;

$\gamma: S \times (A \cup E) \rightarrow 2^S$ is a state-transition function.

A state-transition system may be represented by a directed graph whose nodes are the states in S (for example, see figure 5). If $s' \in \gamma(s, e)$, where $e \in A \cup E$ is an action or event, then the graph contains a *state transition* (that is, an arc)

from s to s' that is labeled with the action or event e .²

If a is an action and $\gamma(s, a)$ is not empty, then action a is *applicable* to state s : if the plan executor executes a in state s , this will take the system to some state in $\gamma(s, a)$.

If e is an event and $\gamma(s, e)$ is not empty, then e may *possibly* occur when the system is in state s . This event corresponds to the internal dynamics of the system, and cannot be chosen or triggered by the plan executor. Its occurrence in state s will bring the system to some state in $\gamma(s, e)$.

Given a state-transition system Σ , the purpose of planning is to find which actions to apply to which states in order to achieve some objective, when starting from some given situation. A *plan* is a structure that gives the appropriate actions. The objective can be specified in

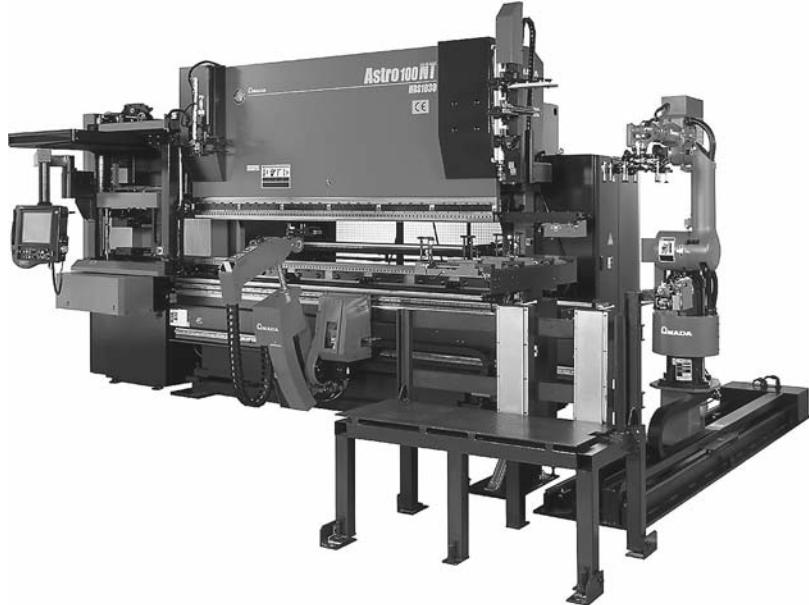


Figure 2. A Sheet-Metal Bending Machine.

(Figure used with permission of Amada Corporation).

several different ways. The simplest specification consists of a *goal state* s_g or a set of goal states S_g . For example, if the objective in figure 5 is to have the container loaded onto the robot cart, then the set of goal states is $S_g = \{s_4, s_5\}$. In this case, the objective is achieved by any sequence of state transitions that ends at one of the goal states. More generally, the objective might be to get the system into certain states, to keep the system away from certain other states, to optimize some utility function, or to perform some collection of tasks.

Planners

The planner's input is a *planning problem*, which includes a description of the system Σ , an initial situation and some objective. For example, in figure 5, a planning problem P might consist of a description of Σ , the initial state s_0 , and a single goal state s_5 .

The planner's output is a plan or policy that solves the planning problem. A *plan* is a sequence of actions such as

<take, move1, load, move2>.

A *policy* is a partial function from states into actions, such as

$\{(s_0, \text{take}), (s_1, \text{move1}), (s_3, \text{load}), (s_4, \text{move2})\}^3$

The aforementioned plan and policy both solve the planning problem P . Either of them, if executed starting at the initial state s_0 , will take Σ through the sequence of states $\langle s_1, s_2, s_3, s_4, s_5 \rangle$.⁴

In general, the planner will produce actions that are described at an abstract level. Hence it may be impossible to perform these actions without first deciding some of the details. In many planning problems, some of these details include what resources to use and what time to do the action.

What Resources to Use. Exactly what is meant by a *resource* depends on how the problem is specified. For example, if Σ contained more than one robot, then one approach would be to require the robot's name as part of the action (for example, move1(robot) and move1(robot2)), and another approach would be to consider the robot to be a resource whose identity will be determined later.

What Time to Do the Action. For example, in

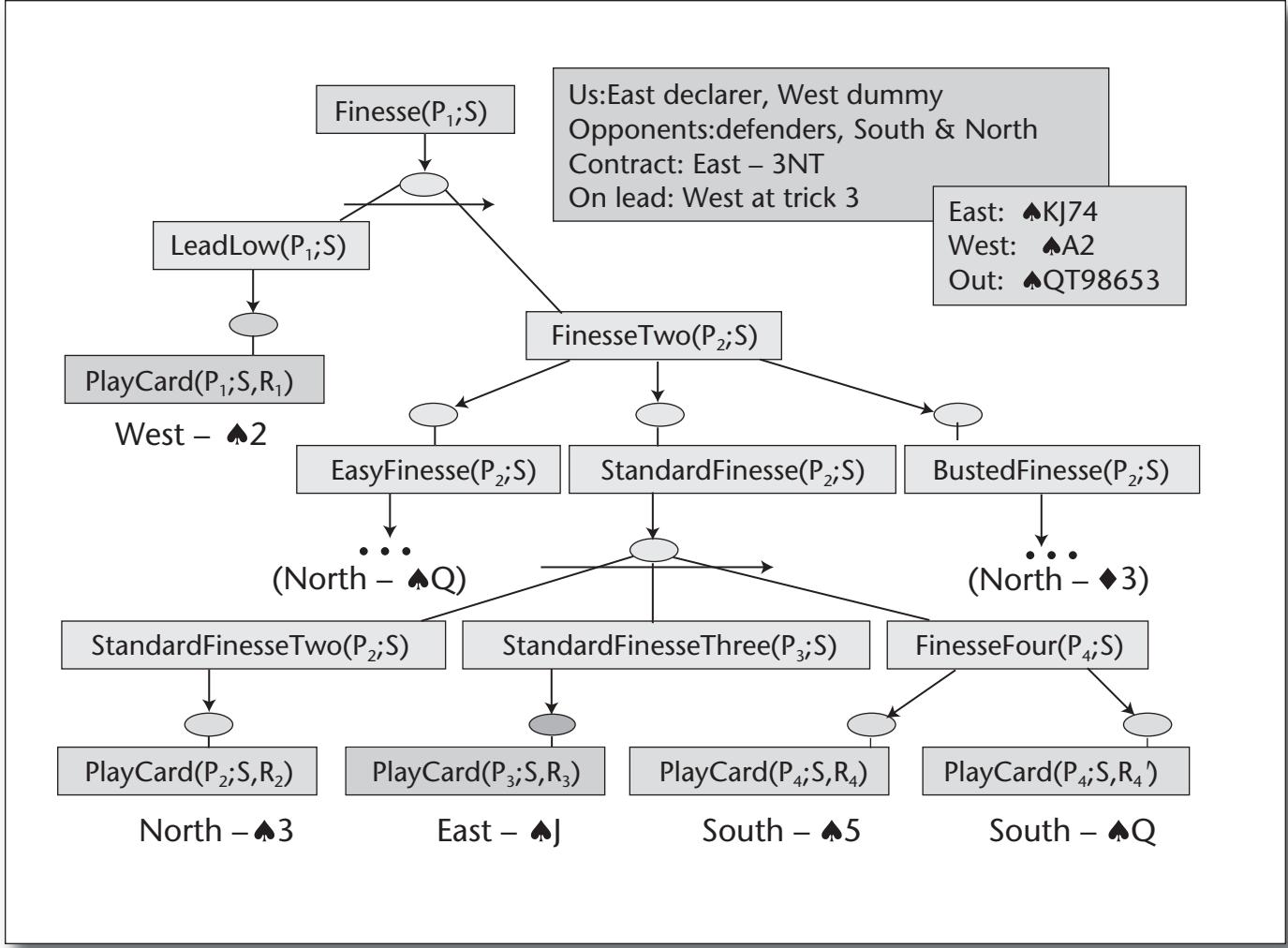


Figure 3. A Hierarchical Plan for a Finesse in Bridge.

order to load the container onto the robot, we might want to start moving the crane before the robot arrives at location1, but we cannot complete the load operation until after the robot has reached location1 and has stopped moving.

In such cases, one approach is to have a separate program called a *scheduler* that sits in between the planner and the controller (see figure 4c), whose purpose is to determine those details. Another approach is to integrate the scheduling function directly into the planner. The latter approach can substantially increase the complexity of the planner, but on complex problems it can be much more efficient than having a separate scheduler.

Controllers

The *controller's* input consists of plans (or schedules, if the system includes a scheduler)

and observations about the current state of the system. The controller's output consists of actions to be performed in the state-transition system.

In figure 4, notice that the controller is *online*. As it performs its actions, it receives *observations*, each observation being a collection of sensor inputs giving information about Σ 's current state. The observations can be modeled as an observation function $\eta : S \rightarrow O$ that maps S into some discrete set of possible observations. Thus, the input to the controller is the observation $o = \eta(s)$, where s is the current state.

If η is a one-to-one function, then from each observation o we can deduce exactly what state Σ is in. In this case we say that the observations provide *complete* information. For example, in figure 5, if there were a collection of sensors that always provided the exact locations of the

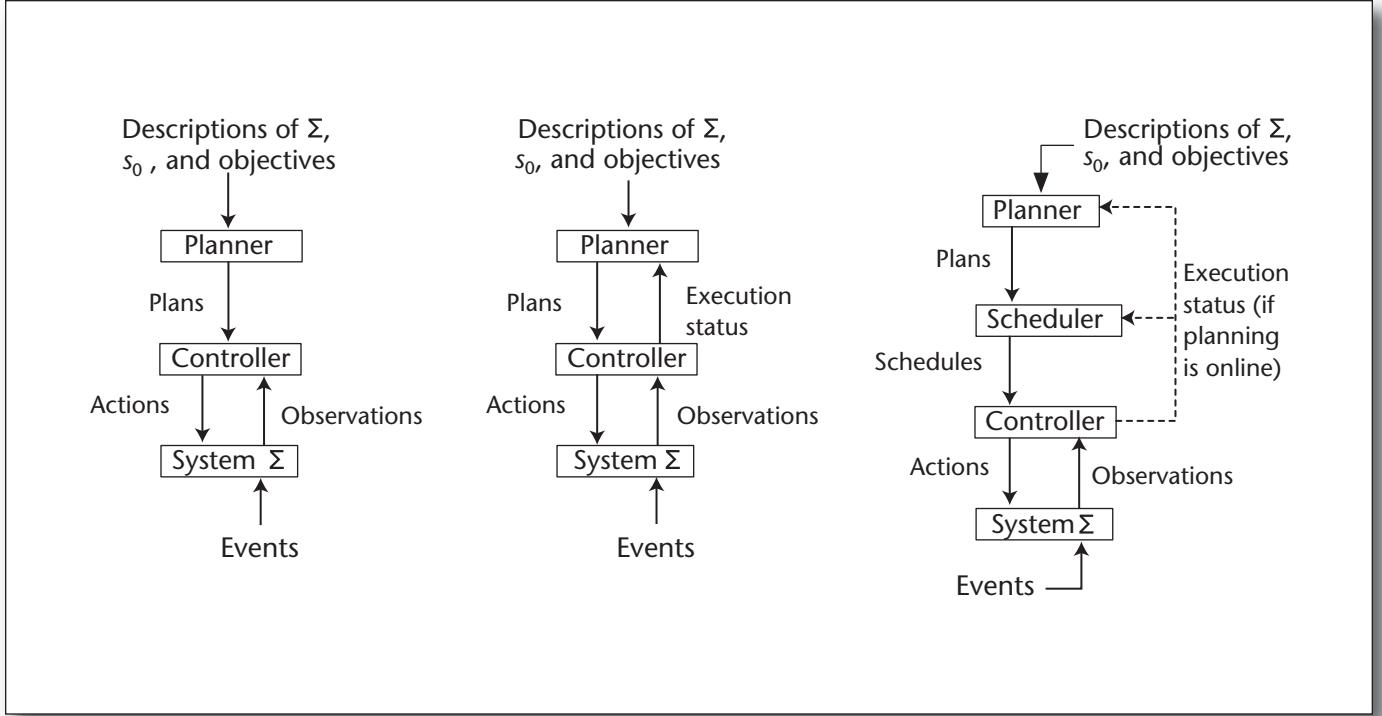


Figure 4. Simple Conceptual Models for (a) *Offline Planning*, (b) *Online Planning*, and (c) *Planning with a Separate Scheduler*.

robot and the container, then this sensor would provide complete information—or, at least, complete information for the level of abstraction used in the figure.

If η is not a one-to-one function, then the best we can deduce from an observation o is that Σ is in one of the states in the set $\eta^{-1}(o) \subseteq S$, and in this case we say that the observations provide *incomplete* information about Σ 's current state. For example, in figure 5, if we had a sensor that told us the location of the robot but not the location of the container, this sensor would provide incomplete information.

Offline and Online Planning

The planner usually works *offline*, that is, it receives no feedback about Σ 's current state. Instead, it relies on a formal description of the system, together with an initial state for the planning problem and the required objective. It is not concerned with the actual state of the system at the time that the planning occurs but instead with what states the system may be in when the plan is executing.

Most of the time, there are differences between Σ and the physical system it represents—for example, the state-transition system in figure 5 does not include the details of the low-level control signals that the controller will need to transmit to the crane and the robot. As

a consequence, the controller and the plan must be robust enough to cope with differences between Σ and the real world. If the differences can become too great for the controller to handle from what is expected in the plan, then more complex control mechanisms will be required than what we have described so far. For example, the planner (and scheduler, if there is one) may need feedback about the controller's execution status (see the dashed lines in the figure), so that planning and acting can need to be interleaved. Interleaving planning and acting will require the planner and scheduler to incorporate mechanisms for supervision, revision, and regeneration of plans and schedules.

Types of Planners

Automated planning systems can be classified into the following categories, based on whether—and in what way—they can be configured to work in different planning domains: domain-specific planners, domain-independent planners, and domain-configurable planners.

Domain-specific planners are planning systems that are tailor-made for use in a given planning domain and are unlikely to work in other domains unless major modifications are made to the planning system. This class

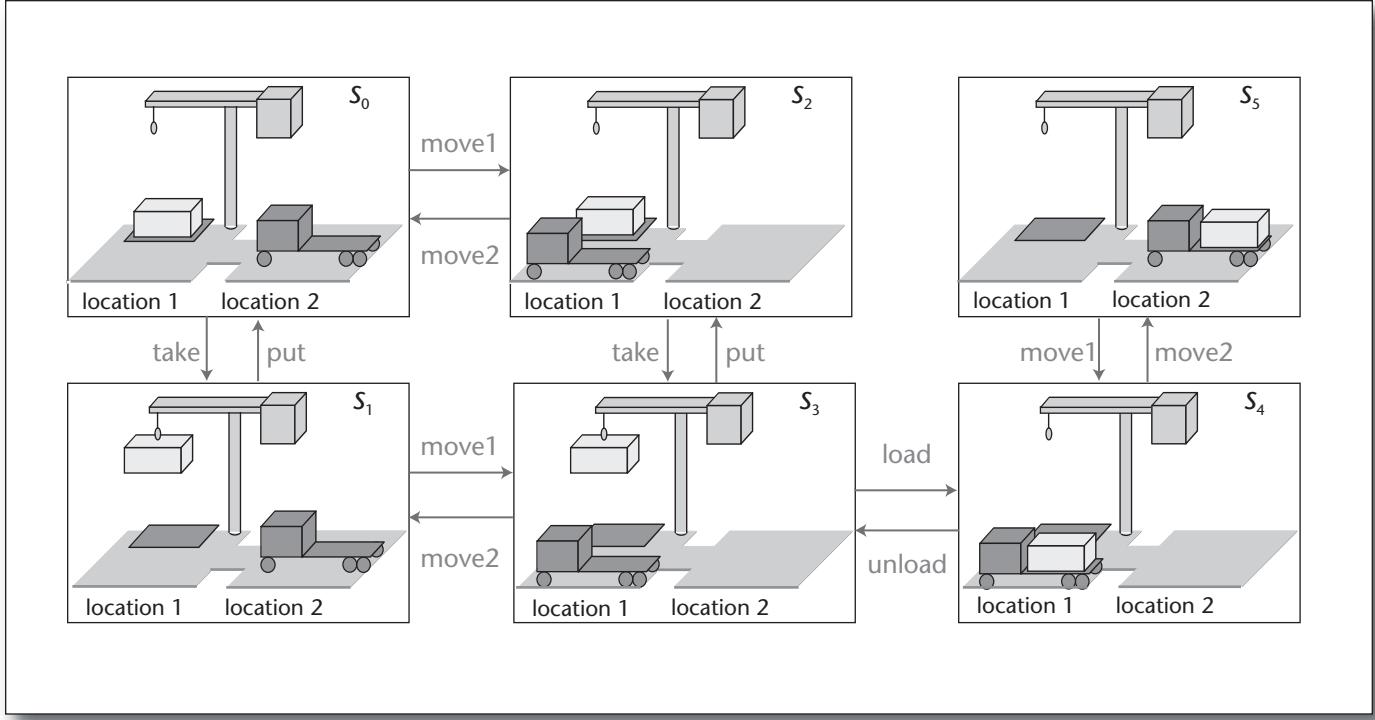


Figure 5. A State-Transition System for a Simple Domain Involving a Crane and a Robot for Transporting Containers.

includes most of the planners that have been deployed in practical applications, such as the ones depicted in figures 1–3.

In **domain-independent planning systems**, the sole input to the planner is a description of a planning problem to solve, and the planning engine is general enough to work in any planning domain that satisfies some set of simplifying assumptions. The primary limitation of this approach is that it isn't feasible to develop a domain-independent planner that works efficiently in every possible planning domain: for example, one probably wouldn't want to use the same planning system to play bridge, bend sheet metal, or control a Mars Rover. Hence, in order to develop efficient planning algorithms, it has been necessary to impose a set of simplifying assumptions that are too restrictive to include most practical planning applications.

Domain-configurable planners are planning systems in which the planning engine is domain-independent but the input to the planner includes domain-specific knowledge to constrain the planner's search so that the planner searches only a small part of the search space. Examples of such planners include HTN Planners such as O-Plan (Tate, Drabble, and Kirby 1994), SIPE-2 (Wilkins 1988), and SHOP2 (Nau et al. 2003), in which the domain-specific knowledge is a collection of methods for

decomposing tasks into subtasks, and **control-rule planners** such as TLPlan (Bacchus and Kabanza 2000) and TALplanner (Kvarnström and Doherty 2001), in which the domain-specific knowledge is a set of rules for how to remove nodes from the search space.⁵

The next two sections are overviews of domain-independent and domain-configurable planners. This article does not include a similar overview of domain-specific planners, since each of these planners depends on the specific details of the problem domain for which it was constructed.

Domain-Independent Planning

For nearly the entire time that automated planning has existed, it has been dominated by research on domain-independent planning. Because of the immense difficulty of devising a domain-independent planner that would work well in *all* planning domains, most research has focused on **classical planning domains**, that is, domains that satisfy the following set of restrictive assumptions:

Assumption A0 (Finite Σ). The system Σ has a finite set of states.

Assumption A1 (Fully Observable Σ). The system Σ is *fully observable*, that is, one has complete knowledge about the state of Σ ; in this case the

observation function η is the identity function.

Assumption A2 (Deterministic Σ). The system Σ is *deterministic*, that is, for every state s and event or action u , $|\gamma(s, u)| \leq 1$. If an action is applicable to a state, its application brings a deterministic system to a single other state. Similarly for the occurrence of a possible event.

Assumption A3 (Static Σ). The system Σ is *static*, that is, the set of events E is empty. Σ has no internal dynamics; it stays in the same state until the controller applies some action.⁶

Assumption A4 (Attainment Goals). The only kind of goal is an *attainment goal*, which is specified as an explicit goal state or a set of goal states S_g . The objective is to find any sequence of state transitions that ends at one of the goal states. This assumption excludes, for example, states to be avoided, constraints on state trajectories, and utility functions.

Assumption A5 (Sequential Plans). A solution plan to a planning problem is a linearly ordered finite sequence of actions.

Assumption A6 (Implicit Time). Actions and events have no duration, they are instantaneous state transitions. This assumption is embedded in the state-transition model, which does not represent time explicitly.

Assumption A7 (Off-line Planning). The planner is not concerned with any change that may occur in Σ while it is planning; it plans for the given initial and goal states regardless of the current dynamics, if any.

In summary, classical planning requires complete knowledge about a deterministic, static, finite system with restricted goals and implicit time. Here planning reduces to the following problem:

Given $\Sigma = (S, A, \gamma)$, an initial state s_0 and a subset of goal states S_g , find a sequence of actions corresponding to a sequence of state transitions (s_0, s_1, \dots, s_k) such that $s_1 \in \gamma(s_0, a_1), s_2 \in \gamma(s_1, a_2), \dots, s_k \in \gamma(s_{k-1}, a_k)$, and $s_k \in S_g$.

Classical planning may appear trivial: planning is simply searching for a path in a graph, which is a well understood problem. Indeed, if we are given the graph Σ explicitly then there is not much more to say about planning for this restricted case. However, it can be shown (Ghallab, Nau, and Traverso 2004) that even in very simple problems, the number of states in Σ can be many orders of magnitude greater than the number of particles in the universe! Thus it is impossible in any practical sense to list all of Σ 's states explicitly. This establishes the need for powerful *implicit* representations that can describe useful subsets of S in a way that both is compact and can easily be searched.

The simplest representation for classical planning is a *set-theoretic* one: a state s is repre-

sented as a collection of propositions, the set of goal states S_g is represented by specifying a collection of propositions that all states in S_g must satisfy, and an action a is represented by giving three lists of propositions: preconditions to be met in a state s for an action a to be applicable in s , propositions to assert, and propositions to retract from s in order to get the resulting state $\gamma(s, a)$. A plan is any sequence of actions, and the plan solves the planning problem if, starting at s_0 , the sequence of actions is executable, producing a sequence of states whose final state is in S_g .⁷

A more expressive representation is the *classical representation*: starting with a function-free first-order language L , a state s is a collection of ground atoms, and the set of goal states S_g is represented by an existentially closed collection of atoms that all states must satisfy. An operator is represented by giving two lists of ground or unground literals: preconditions and effects. An action is a ground instance of an operator. A plan is any sequence of actions, and the plan solves the planning problem if, starting at s_0 , the sequence of actions is executable, producing a sequence of states whose final state satisfies in S_g . The de facto standard for classical planning is to use some variant of this representation.

Classical Planning Algorithms

In the following paragraphs, I provide brief summaries of some of the best-known techniques for classical planning: plan-space planning, planning graphs, state-space planning, and translation into other problems.

Plan-Space Planning. In plan-space planning, the basic idea is to plan for a set of goals $\{g_1, \dots, g_k\}$ by planning for each of the individual goals more-or-less separately, but maintaining various bookkeeping information to detect and resolve interactions among the plans for the individual goals. For example, a simple domain called the Dock Worker Robots domain (Ghallab, Nau, and Traverso 2004), which includes piles of containers, robots that can carry containers to different locations, and cranes that can put containers onto robots or take them off of robots. Suppose there are several piles of containers as shown in state s_0 of figure 6, and the objective is to rearrange them as shown in state s_g . Then a plan-space planner such as UCPOP (Penberthy and Weld, 1992) will produce a partially ordered plan like the one shown in the figure.

Planning Graphs. A *planning graph* is a structure such as the one shown in figure 7. For each n , level n includes every action a such that at level $n - 1$, a 's preconditions are satis-

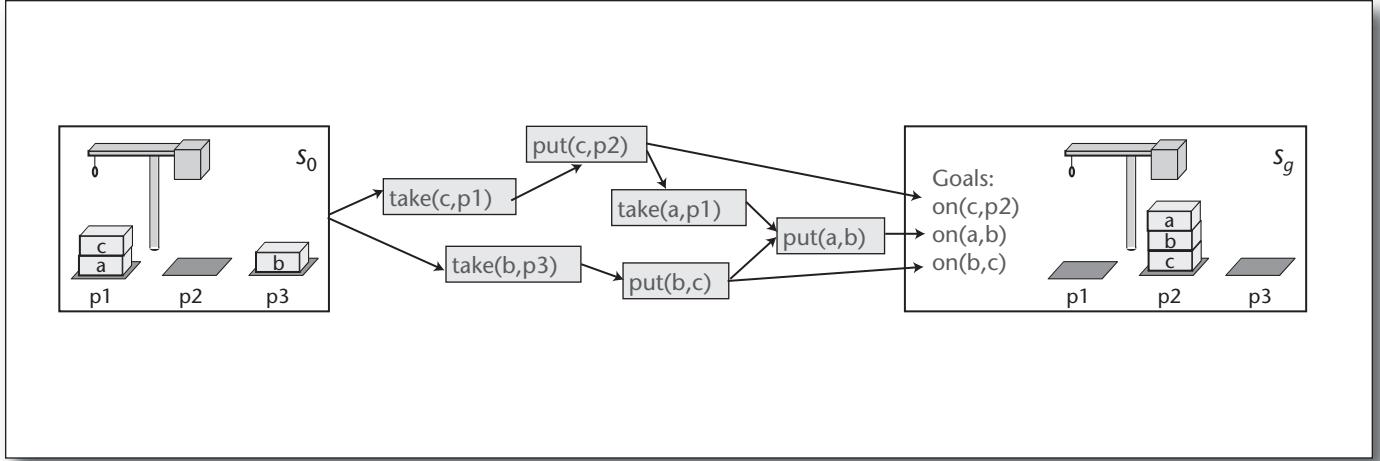


Figure 6. During the Operation of a Plan-Space Planner, Each Node of the Search Space is a Partial Plan like the One Shown Here.

On the left-hand side is the initial state s_0 , on the right-hand side is the goal state s_g , and in the middle is a collection of actions and constraints. Planning proceeds by introducing more and more constraints or actions, until the plan contains no more flaws (that is, it is guaranteed to be executable and to produce the goal).

fied and do not violate certain kinds of mutual-exclusion constraints. The literals at level n include the literals at level $n - 1$, plus all effects of all actions at level n . Thus the planning graph represents a relaxed version of the planning problem in which several actions can appear simultaneously even if they conflict with each other. The basic GraphPlan algorithm is as follows:

```
For  $n = 1, 2, \dots$  until a solution is found,
  Create a planning graph of  $n$  levels.
  Do a backwards state-space search from the
  goal to try to find a solution plan, but restrict
  the search to include only the actions in the
  planning graph.
```

The planning graph can be computed relatively quickly (that is, in a polynomial amount of time), and the restriction that the backward search must operate within the planning graph dramatically improves the efficiency of the backward search. As a result, GraphPlan runs much faster than plan-space planning algorithms. Researchers have created a large number of planning algorithms based on GraphPlan, including IPP, CGP, DGP, LGP, PGP, SGP, TGP, and others.⁸

State-Space Planning. Although state-space search algorithms are very well known, it was not until a few years ago that they began to receive much attention from classical planning researchers, because nobody knew how to come up with a good heuristic function to guide the search. The breakthrough came when it was realized that heuristic values could be computed relatively quickly by extracting

them from relaxed solutions (such as the planning graphs discussed earlier). This has led to planning algorithms such as HSP (Bonet and Geffner 1999) and FastForward (Hoffmann and Nebel 2001).

Translation Into Other Problems. Here, the basic idea consists of three steps. First, translate the planning problem into another kind of combinatorial problem—such as satisfiability or integer programming—for which efficient problem solvers already exist. Second, use a satisfiability solver or integer-programming solver to solve the translated problem. Third, take the solution found by the problem solver and translate it into a plan. This approach has led to planners such as Satplan (Kautz and Selman 1992).

Limitations of Classical Planning

For nearly the entire time that automated planning has existed, it has been dominated by research on classical planning. In fact, for many years the term *domain-independent planning system* was used almost synonymously with classical planning, as if there were no limitations on what kind of planning domains could be represented as classical planning domains. But since classical planning requires all of the restrictive assumptions in the Domain-Independent Planning section, it actually is restricted to a very narrow class of planning domains that exclude most problems of practical interest. For example, Mars exploration (figure 1) and sheet-metal bending (figure 2) satisfy none of the assumptions in the

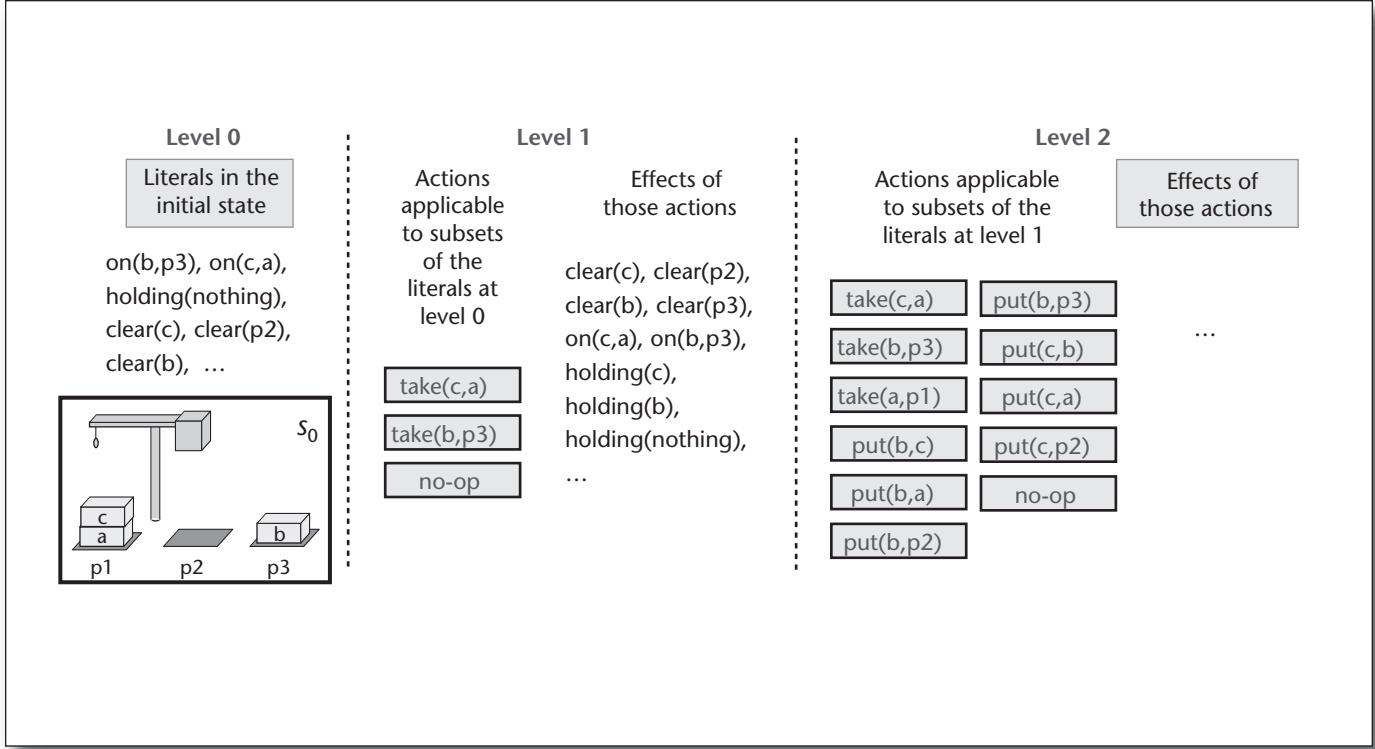


Figure 7. Part of a Planning Graph.

The figure omits several details (for example, mutual-exclusion relationships). The 0'th level contains all literals that are true in the initial state s_0 . For each i , level $i + 1$ includes all actions that are applicable to the literals (or some subset of them) at level i and all of these actions' effects.

Domain-Independent Planning section, and the game of bridge (figure 3) satisfies only A0, A2, and A6.

On the other hand, several of the concepts developed in classical planning have been quite useful in developing planners for non-classical domains. This will be discussed further in the Directions for Growth section.

Domain-Configurable Planners

Domain-specific and domain-configurable planners make use of domain-specific knowledge to constrain the search to a small part of the search space. As an example of why this might be useful, consider the task of traveling from the University of Maryland in College Park, Maryland, to the LAAS research center in Toulouse, France. We may have a number of actions for traveling: walking, riding a bicycle, roller skating, skiing, driving, taking a taxi, taking a bus, taking a train, flying, and so forth. We may also have a large number of locations among which one may travel: for example, all of the cities in the world. Before finding a solution, a domain-independent planner might

first construct a huge number of nonsensical plans, such as the following one:

Walk from College Park to Baltimore, then bicycle to Philadelphia, then take a taxi to Pittsburgh, then fly to Chicago,

In contrast, anyone who has ever had much practical experience in traveling knows that there are only a few reasonable options to consider. For traveling from one location to another, we would like the planner to concentrate only on those options. To do this, the planner needs some domain-specific information about how to create plans.

In a domain-specific planner, the domain-specific information may be encoded into the planning engine itself. Such a planner can be quite efficient in creating plans for the target domain but will not be usable in any other planning domain. If one wants to create a planner for another domain—for example, planning the movements of a robot cart—one will need to build an entirely new planner.

In a domain-configurable planner, the planning engine is domain independent, but the input to the planner includes a *domain description*, that is, a collection of domain-specific

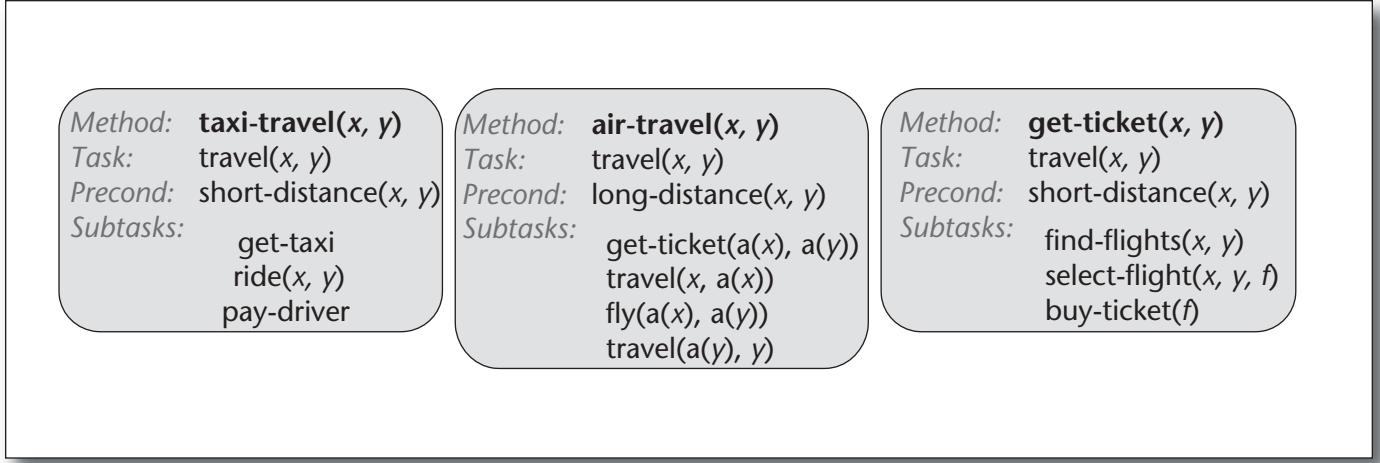


Figure 8. HTN Methods for a Simple Travel-Planning Domain.

In this example, the subtasks are to be done in the order that they appear.

knowledge written in a language that is understandable to the planning engine. Thus, the planning engine can be reconfigured to work in another problem domain by giving it a new domain description.

The existing domain-configurable planners can be divided into two types: hierarchical task network (HTN) planners and control-rule planners. These planners are discussed in the following two sections.

HTN Planners

In a hierarchical task network (HTN) planner, the planner's objective is described not as a set of goal states but instead as a collection of *tasks* to perform.⁹ Planning proceeds by decomposing tasks into subtasks, subtasks into subsubtasks, and so forth in a recursive manner until the planner reaches *primitive* tasks that can be performed using actions similar to the actions used in a classical planning system. To guide the decomposition process, the planner uses a collection of *methods* that give ways of decomposing tasks into subtasks.

As an illustration, figure 8 shows two methods for the task of traveling from one location to another: *air travel* and *taxi travel*. Traveling by air involves the subtasks of purchasing a plane ticket, traveling to the local airport, flying to an airport close to our destination, and traveling from there to our destination. Traveling by taxi involves the subtasks of calling a taxi, riding in it to the final destination, and paying the driver. The preconditions specify that the *air-travel* method is applicable only for long distances and the *taxi-travel* method is applicable only for short distances. Now, consider again the task of traveling from the Uni-

versity of Maryland to LAAS. Since this is a long distance, the *taxi-travel* method is not applicable, so we must choose the *air-travel* method. As shown in figure 9, this decomposes the task into the following subtasks: (1) purchase a ticket from IAD (Washington Dulles) airport to TLS (Toulouse Blagnac), (2) travel from the University of Maryland to IAD, (3) fly from IAD to TLS, and (4) travel from TLS to LAAS.

For the subtasks of traveling from the University of Maryland to BWI and traveling from Logan to MIT, we can use the *taxi-travel* method to produce additional subtasks as shown in figure 9.

HTN-planning research has been much more application-oriented than most other AI-planning research. Domain-configurable systems such as O-Plan (Tate, Drabble, and Kirby 1994), SIPE-2 (Wilkins 1988), and SHOP2 (Nau et al. 2003) have been used in a variety of applications, and domain-specific HTN planning systems have been built for several application domains (for example, Smith, Nau, and Throop [1998]).

Control-Rule Planners

In a control-rule planner, the domain-specific information is a set of rules describing conditions under which the current node can be pruned from the search space. In most cases, the planner does a forward search, starting from the initial state, and the control rules are written in some form of temporal logic. Don't read too much into the name *temporal logic*, because the logical formalisms used in these planners provide only a simple representation of time, as a sequence of states of

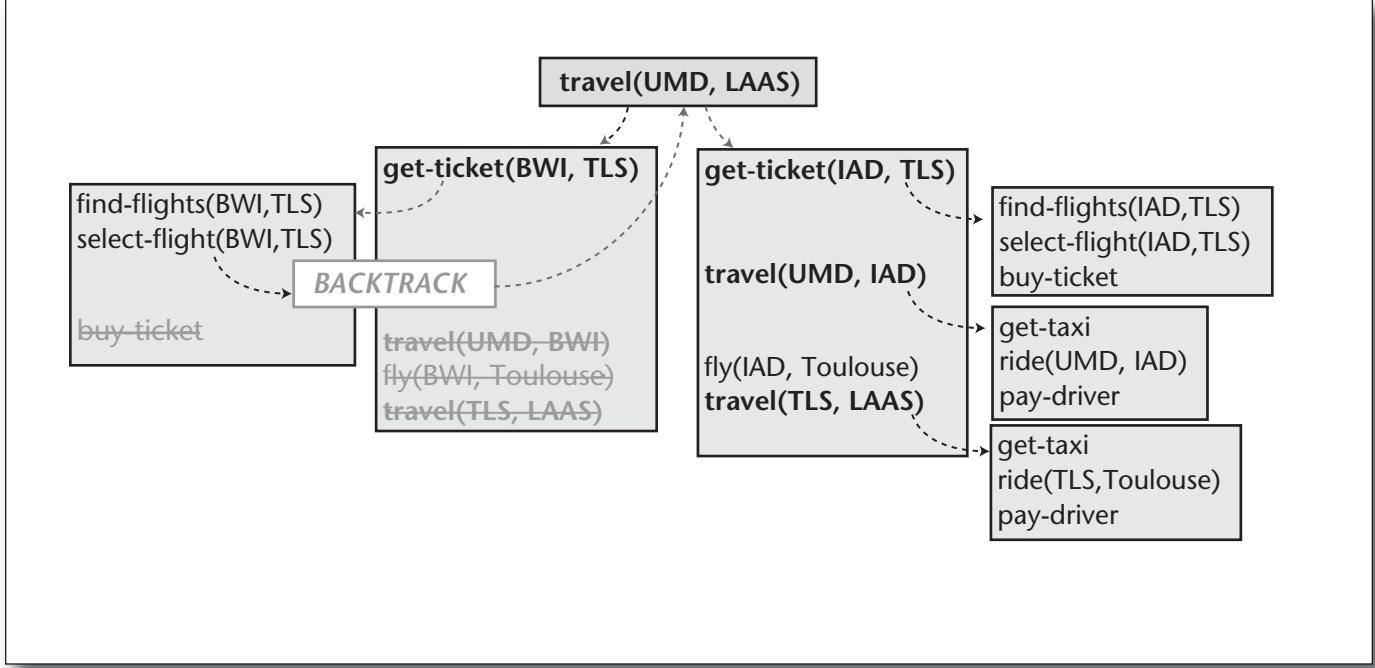


Figure 9. Backtracking Search to Find a Plan for Traveling from UMD to LAAS.

The downward arrows indicate task decompositions; the upward ones indicate backtracking.

the world s_0 = the initial state, $s_1 = \gamma(s_0, a_1)$, $s_2 = \gamma(s_1, a_2)$, ... where a_1, a_2, \dots are the actions of a plan.

As an example, the planning problem of figure 5 is in a simple planning domain called the Dock Worker Robots domain (Ghallab, Nau, and Traverso 2004), which includes piles of containers, robots that can carry containers to different locations, and cranes that can put containers onto robots or take them off of robots. In this domain, suppose that the only kind of goal that we have is to put various containers into various piles. Then the following control rule may be useful:

Don't ever pick up a container from the top of any pile p unless at least one of the containers in p needs to be in another pile.

Here is a way to write this rule in the temporal logic used by TLPlan:

$\square [\text{top}(p, x) \wedge \neg \exists [y: \text{in}(y, p)] \exists [q: \text{GOAL}(\text{in}(y, q))] \wedge (q \neq p) \Rightarrow \bigcirc (\neg \exists [k: \text{holding}(k, x)])]$

What this rule says, literally, is the following:

In every state of the world, if x is at the top of p and there are no y and q such that (1) y is in p , (2) there's a goal saying that y should be in q , and (3) $q \neq p$, then in the next state, no crane is holding x .

If a planner such as TLPlan reaches a state that does not satisfy this rule, it will backtrack and try a different direction in its search space.

Comparisons

The three types of planners — domain-independent, domain-specific, and domain-configurable — compare with each other in the following respects (see table 1): (1) effort to configure the planner for a new domain, (2) performance in a given domain, and (3) coverage across many domains.

Effort to Configure the Planner for a New Domain. Domain-specific planners require the most effort to configure, because one must build an entire new planner for each new domain—an effort that may be quite substantial. Domain-independent planners (provided, of course, that the new domain satisfies the restrictions of classical planning) require the least effort, because one needs to write only definitions of the basic actions in the domain. Domain-configurable planners are somewhere in between: one needs to write a domain description but not an entire planner.

Performance in a Given Domain. A domain-independent planner will typically have the worst level of performance because it will not take advantage of any special properties of the domain. A domain-specific planner, provided that one makes the effort to write a good one, will potentially have the highest level of performance, because one can encode domain-specific problem-solving techniques directly

	Up Front Human Effort	Performance	Coverage
Highest	Domain Specific	Domain Specific	Configurable
	Configurable	Configurable	Domain Independent
Lowest	Domain Independent	Domain Independent	Domain Specific

Table 1. Relative Comparisons of the Three Types of Planners.

into the planner. A sufficiently capable domain-configurable planner should have nearly the same level of performance, because it should be possible to encode the same domain-specific problem-solving techniques into the domain description that one might encode into a domain-specific planner.¹⁰

Coverage across Many Domains. A domain-specific planner will typically work in only one planning domain, hence will have the least coverage. One might think that domain-independent and domain-configurable planners would have roughly the same coverage—but in practice, domain-configurable planners have greater coverage. This is due partly to efficiency and partly to expressive power.

As an example, let us consider the series of semiannual International Planning Competitions, which have been held in 1998, 2000, 2002, 2004, and 2006. All of the competitions included domain-independent planners; and in addition, the 2000 and 2002 competitions included domain-configurable planners. In the 2000 and 2002 competitions, the domain-configurable planners solved the most problems, solved them the fastest, usually found better solutions, and worked in many nonclassical planning domains that were beyond the scope of the domain-independent planners.

A Cultural Bias

If domain-configurable planners perform better and have more coverage than domain-independent ones, then why were there no domain-configurable planners in the 2004 and 2006 International Planning Competitions? One reason is that it is hard to enter them in the competition, because you must write all of the domain knowledge yourself. This is too much trouble except to make a point. The authors of TLPlan, TALplanner, and SHOP2 felt they had already made their point by demonstrating the high performance of their planners in the 2002 competition, hence they didn't feel motivated to enter their planners again.

So why not revise the International Planning Competitions to include tracks in which the necessary domain descriptions are provided to the contestants? There are two reasons. The first is that unlike classical planning, in which PDDL¹¹ is the standard language for representing planning domains, there is no standard domain-description representation for domain-configurable planners. The second is that there is a cultural bias against the idea. For example, when Drew McDermott, in an invited talk at the 2005 International Conference on Planning and Scheduling (ICAPS-05), proposed adding an HTN-planning track to the International Planning Competition, several audience members objected that the use of domain-specific knowledge in a planner somehow constitutes “cheating.”

Whenever I've discussed this bias with researchers in fields such as operations research, control theory, and engineering, they generally have found it puzzling. A typical reaction has been, “Why would anyone not want to use the knowledge they have about a problem they're trying to solve?”

Historically, there is a very good reason for the bias: it was necessary and useful for the development of automated planning as a research field. The intended focus of the field is planning *as an aspect of intelligent behavior*, and it would have been quite difficult to develop this focus if the field had been tied too closely to any particular application area or set of application areas.

While the bias has been very useful historically, I would argue that it is not so useful any more: the field has matured, and the bias is too restrictive. Application domains in which humans want to do planning and scheduling typically have the following characteristics: a dynamically changing world; multiple agents (both cooperative and adversarial); imperfect and uncertain information about the world; the need to consult external information sources (sensors, databases, human users) to get information about the world; time durations,

time constraints, and overlapping actions; durations, time constraints, asynchronous actions; and numeric computations involving resources, probabilities, geometric and spatial relationships. Classical planning excludes all of these characteristics.

Trends

Fortunately, automated-planning research is moving away from the restrictions of classical planning. As an example, consider the evolution of the international planning competitions: The 1998 competition concentrated exclusively on classical planning. The 2000 competition concentrated primarily on classical planning, but one of its tracks (one of the versions of the Miconic-10 elevator domain [Koehler and Schuster 2000]) included some nonclassical elements. The 2002 competition added some elementary notions of time durations, resources. The 2004 competition added inference rules and derived effects, plus a new track for planning in probabilistic domains. The 2006 competition added soft goals, trajectory constraints, preferences, plan metrics, and constraints expressed in temporal logic.

Another reason for optimism is the successful use of automated planning and scheduling algorithms in high-profile projects such as the Remote Agent (on Deep Space 1) (Muscettola et al. 1998) and the Mars Rovers (Estlin et al. 2003). Successes such as this create excitement about building planners that work in the real world; and applications such as the Mars Rovers provide opportunities for synergy between theory and applications: a better understanding of real-world planning leads to better theories, and better theories lead to better real-world planners.

Finally, automated-planning research has produced some very powerful techniques for reducing the size of the search space, and these techniques can be generalized to work in non-classical domains. Examples include partial-order planning, HTN planning, and planning in nondeterministic and probabilistic domains.

Partial-Order Planning. The planning algorithms used in the Remote Agent and the Mars Rovers are based on the plan-space planning technique described in the Classical Planning Algorithms subsection. Some of the primary extensions are ways to reason about time, durations, and resources.

HTN Planning. As I discussed earlier, HTN planners have been used in a variety of application domains. Although most HTN planners have been influenced heavily by concepts from classical planning, they incorporate capabili-

ties that go in various ways beyond the restrictions of classical planning.

Planning in Nondeterministic and Probabilistic Domains. In probabilistic planning domains such as Markov Decision Processes (Boutilier, Dean, and Hanks 1996), the actions have multiple possible outcomes, with probabilities for each outcome (see figure 10a). Nondeterministic planning domains (Cimatti et al. 2003) are similar except that no probabilities are attached to the outcomes (see figure 10b).

A series of recent papers have shown how to extend domain-configurable planning techniques to work in nondeterministic (Kuter et al. 2005) and probabilistic (Kuter and Nau 2005) planning domains. In comparison with previous algorithms for such domains, these new algorithms exhibit substantial performance advantages—advantages analogous to the ones for domain-configurable algorithms in classical planning domains (see the Domain-Configurable Planners section).

Directions for Growth

In my view, some of the more important directions for growth in the near future include planning in multiagent environments, reasoning about time, dynamic external information, acquiring domain knowledge, and cross-pollination with other fields. I'll discuss each of these in greater detail in the following subsections.

Planning in Multiagent Environments

Automated planning research has traditionally assumed that the planner is a monolithic program that solves the problem by itself. But in real-world applications, the planner is generally part of a larger system in which there are other agents, either human or automated or both. When these agents interact with the planner, it is important for the planner to recognize what those agents are trying to accomplish, in order to generate an appropriate response. Examples of such situations include mixed-initiative and embedded planning, assisted cognition, customer service hotlines, and computer games.

Reasoning about Time

Classical planning uses a trivial model of time, consisting of a linear sequence of instantaneous states s_0, s_1, s_2, \dots ; and several temporal logics do the same thing. A more comprehensive model of time would include (1) time durations for actions, overlapping actions, and actions whose durations depend on the conditions under which they are executed; (2) resource assignments, and integrated planning/scheduling; (3) continuous change (for

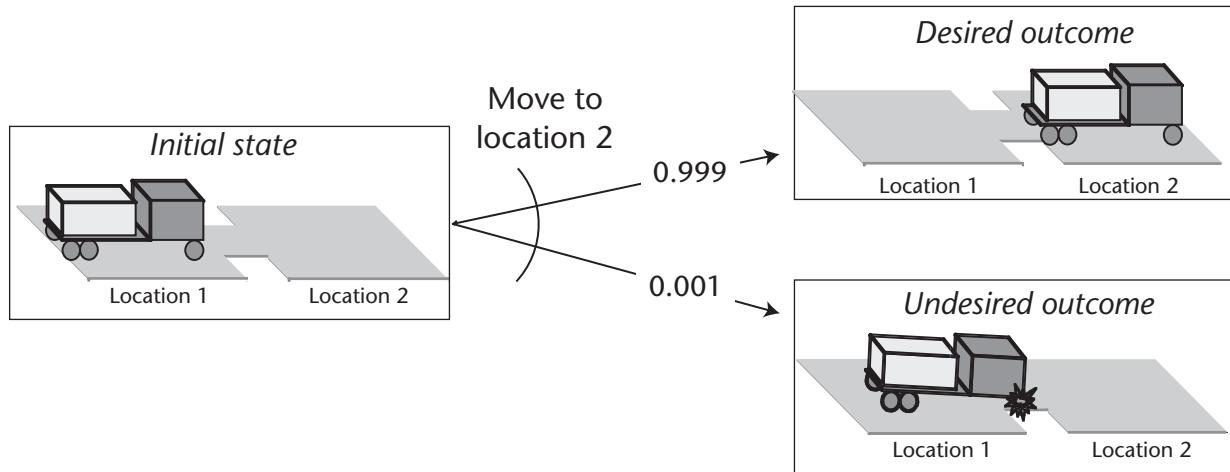


Figure 10. A Simple Example of an Action with a Probabilistic Outcome.

If the robot tries to move to location 2, there is a 99.9 percent chance that it will do so successfully and a 0.1 percent chance it will get a flat tire. An action with a nondeterministic outcome would be similar except that the probabilities would be omitted.

example, vehicle movement); and (4) temporally extended goals (for example, conditions that must be maintained over time).

A healthy amount of growth is already occurring in this direction: for example, various forms of temporal planning were included in the last three International Planning Competitions. But several things remain to be done: for example, the PDDL language used in the competitions does not yet allow actions whose duration depends on the conditions under which they are executed.

Dynamic Environments. In most automated-planning research, the information available is assumed to be static, and the planner starts with all of the information it needs. In real-world planning, planners may need to acquire information from an information source such as a web service, during planning and execution. This raises questions such as What information to look for? Where to get it? How to deal with lag time and information volatility? What if the query for information causes changes in the world? If the planner does not have enough information to infer all of the possible outcomes of the planned actions, or if the plans must be generated in real time, then it may not be feasible to generate the entire plan in advance. Instead, it may

be necessary to interleave planning and plan execution.

Some of these questions, I believe, are sufficiently formalizable that a new track could be developed for them in the International Planning Competitions.

Acquiring Domain Knowledge. At many points in this article, I have emphasized the benefits of using domain knowledge during planning, but this leaves the problem of how to acquire this domain knowledge—whether through machine learning, human input, or some combination of the two. This is one of the least appreciated problems for automated-planning research; but in my view it is one of the most important: if we had good ways of acquiring domain knowledge, we could make planners hundreds of times more useful for real-world problems.

Researchers are starting to realize the importance of this problem. For example, at ICAPS-05 there was an informal “Knowledge Engineering Competition” that was more of an exposition than a competition: the participants exhibited GUIs for creating knowledge bases and ways for planners to learn domain knowledge automatically.

A promising source of planning knowledge is the immense collection of data that is now

available on the web, which is likely to include information about plans in many different contexts. Can we datamine these plans from the web?

Cross-Pollination with Other Fields. Various kinds of planning are studied in many different fields, including computer games, game theory, operations research, economics, psychology, sociology, political science, industrial engineering, systems science, and control theory. Some of the approaches and techniques developed in these fields have much in common. But it is difficult to tell what the relationships are because the research groups are often nearly disjoint, with different terminology, assumptions, and ideas of what's important. Provided that the appropriate bridges can be made among these fields, there is tremendous potential for cross-pollination. Markov decision processes and computational cultural dynamics are two examples.

Markov decision processes (MDPs) are used in operations research, control theory, and several different subfields of AI including automated planning and reinforcement learning. The MDP models used in automated-planning research typically assume the rewards and probabilities are known, but in reinforcement learning (and often in OR and control theory) they are unknown. MDPs in automated-planning research generally have finitely many states with no good continuous approximations, hence use discrete optimization—but the MDP models used in OR and control theory include features such as infinitely many states, continuous sets of states, actions and costs and rewards that are differentiable functions, hence use linear and nonlinear optimization techniques. Many important problems are hybrids of these differing MDP models, and it would be interesting to combine and extend the techniques from the various fields.

We have instituted a new laboratory at the University of Maryland called the Laboratory for Computational Cultural Dynamics. It brings together faculty in computer science, political science, psychology, criminology, systems engineering, linguistics, and business. The objective is to develop the theory and algorithms needed for tools to support decision making in cultural contexts, to help understand how/why decision makers in various cultures make decisions. The potential benefits of such research include more effective cross-cultural interactions, better governance when different cultures are involved, recovery from conflicts and disasters, and improving quality of life in developing countries.

Conclusion

Automated-planning research has made great strides in recent years. Historically, the field was been limited by its focus on classical planning—but the scope of the field is broadening to include a variety of issues that are important for planning in the real world. As a consequence, automated-planning techniques are finding increased use in practical settings ranging from space exploration to automated manufacturing. Some of the most important areas for future growth of the field include reasoning about other agents, temporal planning, planning in dynamic environments, acquiring domain knowledge, and the potential for cross-pollination with other fields.

Notes

1. Austin Tate (*MIT Encyclopedia of the Cognitive Sciences*, 1999).
2. Here we are using a *Markov game* model of a state-transition function, which assumes that actions and events cannot occur at the same time. To include cases where actions and events could occur simultaneously, we would need $\gamma : S \times A \times E \rightarrow 2^S$.
3. Policies have traditionally been defined to be total functions rather than partial functions. But it is not always necessary to know what to do in *every* state of S , because the policy may prevent the system from ever reaching some of the states.
4. Whether to use a plan, a policy, or a more general structure, such as a conditional plan or an execution structure, depends on what kind of planning problem we are trying to solve. In the above example, a plan and a policy work equally well—but more generally, there are some policies that cannot be expressed as plans (for example, in environments where some of the actions have nondeterministic outcomes) and some plans that cannot be expressed as policies (for example, if we come to the same state twice and want to do something different the second time).
5. Note that for an HTN planner or control-rule planner to be domain-configurable, the planning engine must be domain-independent. For example, the version of Bridge Baron mentioned earlier was not domain-configurable because its HTN planning engine was specifically tailored for the game of bridge.
6. Technically, the name of this assumption is inaccurate, because the plan is intended precisely to change the state of the system. What the name means is that the system remains static unless controlled transitions take place.
7. This has also been called STRIPS-style representation, after an early planning system that used a similar representation scheme.
8. Anyone who wants to write a successor of Graph-Plan may want to do so soon, before the supply of three-letter acronyms runs out!
9. In some HTN planners, goals can be specified as

AAAI-08 Author Deadlines

December 1, 2007—

- January 25, 2008: Authors register on the AAAI web site
- January 25, 2008: Electronic abstracts due
- January 30, 2008: Electronic papers due
- March 18-20, 2008: Author feedback about initial reviews
- March 31, 2008: Notification of acceptance or rejection
- April 15, 2008: Camera-ready copy due at AAAI office

tasks such as “achieve(g)” where g is the goal. But as shown in figure 8, tasks can also represent activities that do not correspond to goals in the classical sense.

10. By analogy, in writing a computer system, one can get the highest level of performance by writing assembly code—but one can get nearly the same level of performance with much less effort by writing in a high-level language.

11. See Drew McDermott’s web page, cs-www.cs.yale.edu/homes/dvm.

References

- Bacchus, F., and Kabanza, F. 2000. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence* 116(1–2): 123–191.
- Bonet, B., and Geffner, H. 1999. Planning as Heuristic Search: New Results. In *Proceedings of the European Conference on Planning (ECP)*. Berlin: Springer-Verlag.
- Boutilier, C.; Dean, T. L.; and Hanks, S. 1996. Planning under Uncertainty: Structural Assumptions and Computational Leverage. In *New Directions in AI Planning*, ed. M. Ghallab and A. Milani, 157–171. Amsterdam: IOS Press.
- Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, Strong, and Strong Cyclic Planning Via Symbolic Model Checking. *Artificial Intelligence*, 147(1–2): 35–84.
- Estlin, T.; Castaño, R.; Anderson, B.; Gaines, D.; Fisher, F.; and Judd, M. 2003. Learning and Planning for Mars Rover Science. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*. San Francisco: Morgan Kaufmann Publishers.
- Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. San Francisco: Morgan Kaufmann Publishers.
- Gupta, S. K.; Bourne, D. A.; Kim, K.; and Krishnan, S. S. 1998. Automated Process Planning for Sheet Metal Bending Operations. *Journal of Manufacturing Systems* 17(5): 338–360.
- Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302.
- Kautz, H., and Selman, B. 1992. Planning as Satisfaction. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI)*, 359–363. Chichester, UK: John Wiley and Sons.
- Koehler, J., and Schuster, K. 2000. Elevator Control as a Planning Problem. In *Proceedings of the Fifth International Conference on AI Planning Systems (AIPS)*, 331–338. Menlo Park, CA: AAAI Press.
- Kuter, U., and Nau, D. 2005. Using Domain-Configurable Search Control for Probabilistic Planning. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI)*. Menlo Park, CA: AAAI Press.
- Kuter, U.; Nau, D.; Pistore, M.; and Traverso, P. 2005. A Hierarchical Task-Network Planner Based on Symbolic Model Checking. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 300–309. Menlo Park, CA: AAAI Press.
- Kvarnström, J., and Doherty, P. 2001. TALplanner: A Temporal Logic Based Forward Chaining Planner. *Annals of Mathematics and Artificial Intelligence*, 30(1–4): 119–169.
- Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To Boldly Go Where No AI System has Gone Before. *Artificial Intelligence*, 103(1–2): 5–47.
- Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20(December): 379–404.
- Penberthy, J. S., and Weld, D. 1992. UCPOP: A Sound, Complete, Partial-Order Planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning (KR-92)*. San Francisco: Morgan Kaufmann Publishers.
- Smith, S. J. J.; Nau, D. S.; and Throop, T. 1998. Computer Bridge: A Big Win for AI planning. *AI Magazine*, 19(2): 93–105.
- Tate, A.; Drabble, B.; and Kirby, R. 1994. *O-Plan2: An Architecture for Command, Planning, and Control*. San Mateo, CA: Morgan-Kaufmann Publishers.
- Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann Publishers.



Dana Nau is an AAAI Fellow, a professor of both computer science and systems research at the University of Maryland, and the director of the university's Laboratory

for Computational Cultural Dynamics. He received his Ph.D. from Duke University in 1979, where he was an NSF graduate fellow. He coauthored the automated-planning algorithms that enabled Bridge Baron to win the 1997 world computer bridge championship. His SHOP2 planning system has been used in hundreds of projects worldwide and won an award in the 2002 International Planning Competition. He has more than 300 publications and is coauthor of *Automated Planning: Theory and Practice*, the first comprehensive textbook on automated planning.

Fast Planning Through Planning Graph Analysis*

Avrim L. Blum

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3891
avrim@cs.cmu.edu

Merrick L. Furst

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3891
mxf@cs.cmu.edu

(Final version in *Artificial Intelligence*, 90:281–300, 1997)

Abstract

We introduce a new approach to planning in STRIPS-like domains based on constructing and analyzing a compact structure we call a Planning Graph. We describe a new planner, *Graphplan*, that uses this paradigm. *Graphplan* always returns a shortest-possible partial-order plan, or states that no valid plan exists.

We provide empirical evidence in favor of this approach, showing that *Graphplan* outperforms the total-order planner, Prodigy, and the partial-order planner, UCPOP, on a variety of interesting natural and artificial planning problems. We also give empirical evidence that the plans produced by *Graphplan* are quite sensible. Since searches made by this approach are fundamentally different from the searches of other common planning methods, they provide a new perspective on the planning problem.

Keywords: General Purpose Planning, STRIPS Planning, Graph Algorithms, Planning Graph Analysis.

1 Introduction

In this paper we introduce a new planner, *Graphplan*, which plans in STRIPS-like domains. The algorithm is based on a paradigm we call Planning Graph Analysis. In this approach, rather than immediately embarking upon a search as in standard planning methods, the algorithm instead begins by explicitly constructing a compact structure we call a *Planning Graph*. A Planning Graph encodes the planning problem in such a way that many useful constraints inherent in the problem become explicitly available to reduce the amount of search needed. Furthermore, Planning Graphs can be constructed quickly: they have polynomial size and can be built in polynomial time. It is worth pointing out that a Planning

*This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The first author is also supported in part by NSF National Young Investigator grant CCR-9357793 and a Sloan Foundation Research Fellowship. The second author is supported in part by NSF grant CCR-9119319. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

Graph is *not* the state-space graph, which of course could be huge. In fact, unlike the state-space graph in which a plan is a *path* through the graph, in a Planning Graph a plan is essentially a *flow* in the network flow sense. Planning Graphs are closer in spirit to the Problem Space Graphs (PSGs) of Etzioni [1990], though unlike PSGs, Planning Graphs are based not only on domain information, but also the goals and initial conditions of a problem and an explicit notion of time.

Planning Graphs offer a means of organizing and maintaining search information that is reminiscent of the efficient solutions to Dynamic Programming problems. Planning Graph Analysis appears to have significant practical value in solving planning problems even though the inherent complexity of STRIPS planning, which is at least PSPACE-hard (e.g., see Bylander [1994]), is much greater than the complexity of standard Dynamic Programming problems. We provide empirical evidence on a variety of “natural” and artificial domains showing that Planning Graph Analysis is able to provide a quite substantial improvement in running time.

The **Graphplan** planner uses the Planning Graph that it creates to guide its search for a plan. The search that it performs combines aspects of both total-order and partial-order planners. Like traditional total-order planners, **Graphplan** makes strong commitments in its search. When it considers an action, it considers it at a specific point in time: for instance, it might consider placing the action ‘`move Rocket1 from London to Paris`’ in a plan at exactly time-step 2. On the other hand, like partial-order planners [Chapman, 1987][McAllester and Rosenblitt, 1991][Barrett and Weld, 1994][Weld, 1994], **Graphplan** generates partially ordered plans. For instance, in Veloso’s rocket problem (Figure 1), the plan that **Graphplan** finds is of the form: “In time-step 1, appropriately load all the objects into the rockets, in time-step 2 move the rockets, and in time-step 3, unload the rockets.” The semantics of such a plan is that the actions in a given time step may be performed in any desired order. Conceptually this is a kind of “parallel” plan [Knoblock, 1994], since one could imagine executing the actions in three time steps if one had as many workers as needed to load and unload and fly the rockets.

One valuable feature of our algorithm is that it guarantees it will find the *shortest* plan among those in which independent actions may take place at the same time. Empirically and subjectively these sorts of plans seem particularly sensible. For example, in Stuart Russell’s “flat-tire world” (the goal is to fix a flat tire and then return all the tools back to where they came from; see the UCPOP domains list), the plan produced by **Graphplan** opens the boot (trunk) in step 1, fetches all the tools and the spare tire in step 2, inflates the spare and loosens the nuts in step 3, and so forth until it finally closes the boot in step 12. (See Figure 4.) Another significant feature of our algorithm is that it is not particularly sensitive to the order of the goals in a planning task, unlike traditional approaches. More discussion of this issue is given in Section 3.2. In Section 4 of this paper we present empirical results that demonstrate the effectiveness of **Graphplan** on a variety of interesting “natural” and artificial domains.

An extended abstract of this work appears in [Blum and Furst, 1995].

1.1 Definitions and Notation

Planning Graph Analysis applies to STRIPS-like planning domains [Fikes and Nilsson, 1971]. In these domains, operators have preconditions, add-effects, and delete-effects, all of

The rocket domain (introduced by Veloso [1989]) has three operators: Load, Unload, and Move. A piece of cargo can be loaded into a rocket if the rocket and cargo are in the same location. A rocket may move if it has fuel, but performing the move operation uses up the fuel. In UCPOP format, the operators are:

```
(define (operator move)
  :parameters ((rocket ?r) (place ?from) (place ?to))
  :precondition (:and (:neq ?from ?to) (at ?r ?from) (has-fuel ?r))
  :effect (:and (at ?r ?to) (:not (at ?r ?from)) (:not (has-fuel ?r)))))

(define (operator unload)
  :parameters ((rocket ?r) (place ?p) (cargo ?c))
  :precondition (:and (at ?r ?p) (in ?c ?r))
  :effect (:and (:not (in ?c ?r)) (at ?c ?p)))

(define (operator load)
  :parameters ((rocket ?r) (place ?p) (cargo ?c))
  :precondition (:and (at ?r ?p) (at ?c ?p))
  :effect (:and (:not (at ?c ?p)) (in ?c ?r)))
```

A typical problem might have one or more rockets and some cargo in a start location with a goal of moving the cargo to some number of destinations.

Figure 1: *A Simple Rocket Domain.*

which are conjuncts of propositions, and have parameters that can be instantiated to objects in the world. Operators do not create or destroy objects and time may be represented discretely. An example is given in Figure 1.

Specifically, by a *planning problem*, we mean:

- A STRIPS-like domain (a set of operators),
- A set of objects,
- A set of propositions (literals) called the Initial Conditions,
- A set of Problem Goals which are propositions that are required to be true at the end of a plan.

By an *action*, we mean a fully-instantiated operator. For instance, the operator ‘put ?x into ?y’ may instantiate to the specific action ‘put Object1 into Container2’. An action taken at time t adds to the world all the propositions which are among its Add-Effects and deletes all the propositions which are among its Delete-Effects. It will be convenient to

think of “doing nothing” to a proposition in a time step as a special kind of action we call a *no-op* or *frame* action.

2 Valid Plans and Planning Graphs

We now define what we mean when we say a set of actions forms a valid plan. In our framework, a *valid plan* for a planning problem consists of a set of actions and specified times in which each is to be carried out. There will be actions at time 1, actions at time 2, and so forth. Several actions may be specified to occur at the same time step so long as they do not interfere with each other. Specifically, we say that two actions *interfere* if one deletes a precondition or an add-effect of the other.¹ In a linear plan these *independent* parallel actions could be arranged in any order with exactly the same outcome. A valid plan may perform an action at time 1 if its preconditions are all in the Initial Conditions. A valid plan may perform an action at time $t > 1$ if the plan makes all its preconditions true at time t . Because we have no-op actions that carry truth forward in time, we may define a proposition to be true at time $t > 1$ if and only if it is an Add-Effect of some action taken at time $t - 1$. Finally, a valid plan must make all the Problem Goals true at the final time step.

2.1 Planning Graphs

A Planning Graph is similar to a valid plan, but without the requirement that the actions at a given time step not interfere. It is, in essence, a type of constraint graph that encodes the planning problem.

More precisely, a Planning Graph is a directed, leveled graph² with two kinds of nodes and three kinds of edges. The levels alternate between *proposition levels* containing *proposition nodes* (each labeled with some proposition) and *action levels* containing *action nodes* (each labeled with some action). The first level of a Planning Graph is a proposition level and consists of one node for each proposition in the Initial Conditions. The levels in a Planning Graph, from earliest to latest are: propositions true at time 1, possible actions at time 1, propositions possibly true at time 2, possible actions at time 2, propositions possibly true at time 3, and so forth.

Edges in a Planning Graph explicitly represent relations between actions and propositions. The action nodes in action-level i are connected by “precondition-edges” to their preconditions in proposition level i , by “add-edges” to their Add-Effects in proposition-level $i + 1$, and by “delete-edges” to their Delete-Effects in proposition-level $i + 1$.³

The conditions imposed on a Planning Graph are much weaker than those imposed on valid plans. Actions may exist at action-level i if all their preconditions exist at proposition-level i but there is no requirement of “independence.” In particular, action-level i may

¹Knoblock [1994] describes an interesting less restrictive notion in which several actions may occur at the same time even if one deletes an add-effect of another, so long as those add-effects are not important for reaching the goals.

²A graph is called *leveled* if its nodes can be partitioned into disjoint sets L_1, L_2, \dots, L_n such that the edges only connect nodes in adjacent levels.

³A length-two path from an action a at one level, through a proposition Q at the next level, to an action b at the following level, is similar to a causal link $a \xrightarrow{Q} b$ in a partial-order planner.

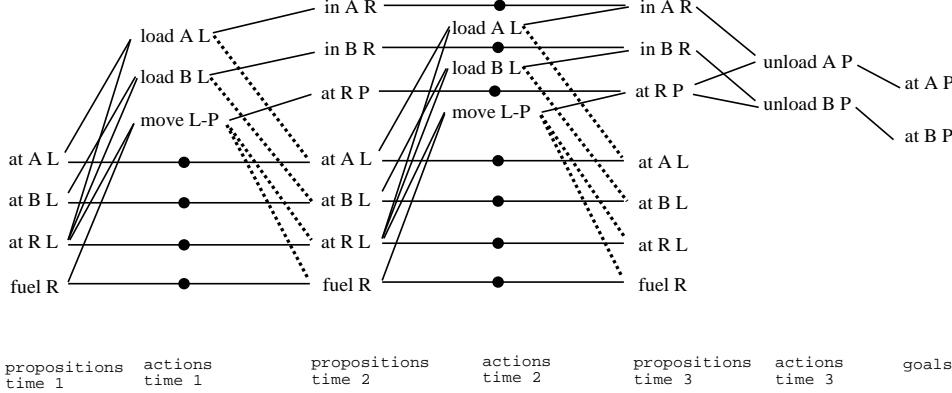


Figure 2: A planning graph for the rocket problem with one rocket R , two pieces of cargo A and B , a start location L and one destination P . For simplicity, the “rocket” parameter has been removed from the actions’ names. Delete edges are represented by dashed lines and no-ops are represented by dots. In the planning graph created by Graphplan for this problem, there would be more action nodes in the second and third action levels.

legally contain *all* the possible actions whose preconditions all exist in proposition-level i . A proposition may exist at proposition-level $i + 1$ if it is an Add-Effect of some action in action-level i (even if it is also a Delete-Effect of some other action in action-level i). Because we allow “no-op actions,” every proposition that appears in proposition-level i may also appear in proposition-level $i + 1$. An example of a Planning Graph is given in Figure 2.

Since the requirements on Planning Graphs are so weak, it is easy to create them. In Section 3.1 we describe how Graphplan constructs Planning Graphs from domains and problems. In particular, any Planning Graph with t action-levels that Graphplan creates will have the following property:

If a valid plan exists using t or fewer time steps, then that plan exists as a subgraph of the Planning Graph.

It is worth noting here that Planning Graphs are not overly large. See Theorem 1.

2.2 Exclusion Relations Among Planning Graph Nodes

An integral part of Planning-Graph Analysis is noticing and propagating certain *mutual exclusion* relations among nodes. Two actions at a given action level in a Planning Graph are *mutually exclusive* if no valid plan could possibly contain both. Similarly, two propositions at a given proposition level are mutually exclusive if no valid plan could possibly make both true. Identifying mutual exclusion relationships can be of enormous help in reducing the search for a subgraph of a Planning Graph that might correspond to a valid plan.

Graphplan notices and records mutual exclusion relationships by propagating them through the Planning Graph using a few simple rules. These rules do not guarantee to find *all* mutual exclusion relationships, but usually find a large number of them.⁴ Specifically, there

⁴In fact, determining *all* mutual exclusion relationships can be as hard as finding a legal plan. For

are two ways in which actions a and b at a given action-level are marked by Graphplan to be exclusive of each other:

[**Interference**] If either of the actions deletes a precondition or Add-Effect of the other. (This is just the standard notion of “non independence” and depends only on the operator definitions.)

[**Competing Needs**] If there is a precondition of action a and a precondition of action b that are marked as mutually exclusive of each other in the previous proposition level.

Two propositions p and q in a proposition-level are marked as exclusive if all ways of creating proposition p are exclusive of all ways of creating proposition q . Specifically, they are marked as exclusive if each action a having an add-edge to proposition p is marked as exclusive of each action b having an add-edge to proposition q .

For instance, in the rocket domain with ‘Rocket1 at London’ and ‘has-fuel Rocket1’ in the Initial Conditions, the actions ‘move Rocket1 from London to Paris’ and ‘load Alex into Rocket1 in London’ at time 1 are exclusive because the first deletes the proposition ‘Rocket1 at London’ which is a precondition of the second. The proposition ‘Rocket1 at London’ and the proposition ‘Rocket1 at Paris’ are exclusive at time 2 because all ways of generating the first (there is only one: a no-op) are exclusive of all ways of generating the second (there is only one: by moving). The actions ‘load Alex into Rocket1 in London’ and ‘load Jason into Rocket1 in Paris’ (assuming we defined the initial conditions to have Jason in Paris) at time 2 are exclusive because they have competing needs, namely the propositions ‘Rocket1 at London’ and ‘Rocket1 at Paris’.

A pair of propositions may be exclusive of each other at every level in a planning graph or they may start out being exclusive of each other in early levels and then become non-exclusive at later levels. For instance, if we begin with Alex and Rocket1 at London (and they are nowhere else at time 1), then ‘Alex in Rocket1’ and ‘Rocket1 at Paris’ are exclusive at time 2, but not at time 3.

2.2.1 The power of exclusion relations

Note that the Competing Needs notion and the exclusivity between propositions are not just logical properties of the operators. Rather, they depend on the interplay between operators and the Initial Conditions.

Consider, for instance, a domain such as the Rocket domain having a move operator. The useful notion that an item cannot be in two places at the same time is not just a function of the operators; if the initial conditions specified that the item started out in two different places, then it *could* continue to be in two places at once. Instead this notion depends both on the definition of ‘move’ and the fact that the item starts out in only one place. The mutual exclusion rules provide a mechanism for propagating this notion through the graph. The reason is that if at time $t - 1$ you can be in only one place, then any two move actions you might perform at time $t - 1$ will be exclusive (any two moves from *different* starting locations are exclusive by Competing Needs and two moves from the *same* starting

instance, consider creating two new artificial goals g_1 and g_2 such that satisfying g_1 requires satisfying half of the original goals and satisfying g_2 requires satisfying the other half. Then, determining whether g_1 and g_2 are mutually exclusive is equivalent to solving the planning problem.

location are exclusive since they delete each others’ preconditions) and therefore you can be in only one place at time t . Propagating these constraints allows the system to use this important fact in planning.

More generally, in many different domains, exclusion relations seem to propagate a variety of intuitively useful facts about the problem throughout the graph.

3 Description of the algorithm

The high-level description of our basic algorithm is the following. Starting with a Planning Graph that only has a single proposition level containing the Initial Conditions, **Graphplan** runs in stages. In stage i **Graphplan** takes the Planning Graph from stage $i - 1$, extends it one time step (the next action level and the following proposition level), and then searches the extended Planning Graph for a valid plan of length i . **Graphplan**’s search either finds a valid plan (in which case it halts) or else determines that the goals are not all achievable by time i (in which case it goes on to the next stage). Thus, in each iteration through this Extend/Search loop, the algorithm either discovers a plan or else proves that no plan having that many time steps or fewer is possible.

Graphplan’s algorithm is sound and complete: any plan the algorithm finds is a legal plan, and if there exists a legal plan then **Graphplan** will find one. In Section 5 we describe how this algorithm may be augmented so that if the Problem Goals are not satisfiable by *any* valid plan, then the planner is guaranteed to halt with failure in finite time. This termination guarantee is one that is not provided by most partial-order planners.

3.1 Extending Planning Graphs

All the initial conditions are placed in the first proposition level of the graph. To create a generic action level, we do the following. For each operator and each way of instantiating preconditions of that operator to propositions in the previous level, insert an action node *if no two of its preconditions are labeled as mutually exclusive*.⁵ Also insert all the no-op actions and insert the precondition edges. Then check the action nodes for exclusivity as described in Section 2.2 above and create an “actions-that-I-am-exclusive-of” list for each action.

To create a generic proposition level, simply look at all the Add-Effects of the actions in the previous level (including no-ops) and place them in the next level as propositions, connecting them via the appropriate add and delete-edges. Mark two propositions as exclusive if all ways of generating the first are exclusive of all ways of generating the second.

As we demonstrate in the following theorem, the time taken by our algorithm to create this graph structure is polynomial in the length of the problem’s description and the number of time steps.

Theorem 1 *Consider a planning problem with n objects, p propositions in the Initial Conditions, and m STRIPS operators each having a constant number of formal parameters. Let*

⁵Checking for exclusions keeps **Graphplan**, for instance, from inserting the action ‘`unload Alex from Rocket1 in Paris`’ in time 2 of the rocket-domain graph when the initial conditions specify that both Alex and the rocket begin in London.

ℓ be the length of the longest add-list of any of the operators. Then, the size of a t -level planning graph created by Graphplan, and the time needed to create the graph, are polynomial in n , m , p , ℓ , and t .

Proof. Let k be the largest number of formal parameters in any operator. Since operators cannot create new objects, the number of different propositions that can be created by instantiating an operator is $O(\ell n^k)$. So, the maximum number of nodes in any proposition-level of the planning graph is $O(p + m\ell n^k)$. Since any operator can be instantiated in at most $O(n^k)$ distinct ways, the maximum number of nodes in any action-level of the planning graph is $O(mn^k)$. Thus the total size of the planning graph is polynomial in n , m , p , ℓ , and t , since k is constant.

The time needed to create a new action and proposition level of the graph can be broken down into (A) the time to instantiate the operators in all possible ways to preconditions in the previous proposition-level, (B) the time to determine mutual exclusion relations between actions, and (C) the time to determine the mutual exclusion relations in the next level of propositions. It is clear that this time is polynomial in the number of nodes in the current level of the graph. ■

Empirically, the part of graph creation that takes the most time is determining exclusion relations. However, empirically, graph creation only takes up a significant portion of Graphplan’s running time in the simpler problems, where the total running time is not very large anyway.

An obvious improvement to the basic algorithm described above (which is implemented in Graphplan) is to avoid searching until a proposition-level has been created in which all the Problem Goals appear and no pair of Problem Goals has been determined to be mutually exclusive.

3.2 Searching for a plan

Given a Planning Graph, Graphplan searches for a valid plan using a backward-chaining strategy. Unlike most other planners, however, it uses a level-by-level approach, in order to best make use of the mutual exclusion constraints. In particular, given a set of goals at time t , it attempts to find a set of actions (no-ops included) at time $t - 1$ having these goals as add effects. The preconditions to these actions form a set of subgoals at time $t - 1$ having the property that if *these* goals can be achieved in $t - 1$ steps, then the original goals can be achieved in t steps. If the goal set at time $t - 1$ turns out not to be solvable, Graphplan tries to find a different set of actions, continuing until it either succeeds or has proven that the original set of goals is not solvable at time t .

In order to implement this strategy, Graphplan uses the following recursive search method. For each goal at time t in some arbitrary order, select some action at time $t - 1$ achieving that goal that is not exclusive of any actions that have already been selected. Continue recursively with the next goal at time t . (Of course, if by good fortune a goal has already been achieved by some previously-selected action, we do not need to select a new action for it.) If our recursive call returns failure, then try a different action achieving our current goal, and so forth, returning failure once all such actions have been tried. Once finished with all the goals at time t , the preconditions to the selected actions make up the new goal

set at time $t - 1$. We call this a “goal-set creation step.” Graphplan then continues this procedure at time step $t - 1$.

A “forward-checking” improvement to this approach (which is implemented in Graphplan and helps modestly in our experiments) is that after each action is considered a check is made to make sure that no goal ahead in the list has been “cut-off.” In other words, Graphplan checks to see if for some goal still ahead in the list, all the actions creating it are exclusive of actions we have currently selected. If there is some such goal, then Graphplan knows it needs to back up right away.

3.2.1 Memoization

One additional aspect of Graphplan’s search is that when a set of (sub)goals at some time t is determined to be not solvable, then before popping back in the recursion it *memoizes* what it has learned, storing the goal set and the time t in a hash table. Similarly, when it creates a set of subgoals at some time t , before searching it first probes the hash table to see if the set has already been proved unsolvable. If so, it then backs up right away without searching further. This memoizing step, in addition to its use in speeding up search, is needed for our termination check described in Section 5.

3.2.2 An example

To make this more concrete, let us consider again the rocket problem in which the Initial Conditions have two fueled rockets and n pieces of cargo at some starting location S and the goal is to move some of the cargo to location X and some to location Y . For this problem, the graph will grow to contain three action levels. The planner will then select some goal, say ‘A at X’, and pick some action at time step 3 such as ‘`unload A from Rocket1 at X`’ making it true. It then marks as not-doable all actions exclusive of this one, such as ‘`unload C from Rocket1 at Y`’, at time step 3. The planner then selects the next goal, say ‘B at X’. If it chooses to make this goal true by performing ‘`unload B from Rocket2 at X`’ at time 3, then it will notice that a goal such as ‘C at Y’ further down in its goal list has been completely cut off, because all ways of making it true are exclusive of the actions already committed to. Thus, Graphplan will instead select ‘`unload B from Rocket1 at X`’, and so on. Once the planner is done with all goals at this level, it then creates a new goal-set at the previous time step consisting of goals such as ‘A in Rocket1’ and ‘Rocket1 at X’ that were the preconditions of the actions selected.

3.2.3 The limited effect of goal orderings

The strategy of working on the subgoals in a somewhat breadth-first-like manner makes Graphplan fairly insensitive to goal-orderings. We now add one final feature to Graphplan’s search strategy that will allow us to make this statement more precise. Let G be a goal set at some time t . We say that a non-exclusive set of actions A at time $t - 1$ is a *minimal set of actions achieving G* if (1) every goal in G is an add-effect of some action in A , and (2) no action can be removed from A so that the add effects of the actions remaining still contain G . The modification to Graphplan’s strategy is to only recurse on minimal action sets. If the set of actions A chosen by Graphplan to achieve some goal-set G is *not* minimal, we back up right away. (For instance, say our goals are g_1 and g_2 ; we pick some action

achieving g_1 and then the action we choose to achieve g_2 happens to also achieve g_1 as well. This would not be minimal.) This modification allows us to make a clean statement about the goal-sets that **Graphplan** considers. Specifically, we can state the following theorem.

Theorem 2 *Let G be a goal set at some time t that is not solvable in t steps. Then, no matter what the ordering of the goals in G , the goal sets at time $t - 1$ that **Graphplan** considers when attempting to achieve G are exactly the preconditions of all the minimal action sets at time $t - 1$ achieving G . (If G is solvable in t steps, then **Graphplan** may halt before considering all those goal sets).*

Proof. We have forced **Graphplan** to consider *only* minimal action sets; we need to show that *every* such set is examined. Let A be some such set, and consider some arbitrary ordering of G . Let a_1 be some action in A achieving the first goal in G (and let's call that goal g_{a_1}). Let a_2 be the action in A achieving the first goal in G not already achieved by a_1 (and let's call that goal g_{a_2}). More generally, let a_i be the action in A achieving the first goal in G not achieved by any of $\{a_1, \dots, a_{i-1}\}$, and we will call that goal g_{a_i} . Notice that all actions in A are given an index in this way because A is minimal. This ordering of the actions implies that at some point in the recursion, a_1 will be the action chosen by **Graphplan** to achieve goal g_{a_1} ; given that that occurs, at some point a_2 will be the action chosen to achieve g_{a_2} , and so forth. Therefore, all actions in A are considered. ■

We can now quantify the limited effect of goal ordering as follows. Suppose **Graphplan** is currently attempting to solve the Problem Goals at some time T and is unsuccessful. Then, the total number of goal-sets examined in the search is *completely independent of* the ordering of the goals. The effect of goal ordering is limited to (A) the amount of time it takes on average to examine a new goal set (perform a goal-set creation step), and (B) the amount of work performed in the final stage at which the Problem Goals *are* found to be solvable (since goal ordering may affect the order in which goal sets are examined). In addition to this theoretical statement, empirically, **Graphplan**'s dependence on goal ordering seems to be quite small: significantly less than that of other planners such as Prodigy and UCPOP.

4 Experimental Results

4.1 Natural domains

We compared **Graphplan** with two popular planners, Prodigy and UCPOP, on several “natural” planning problems from the planning literature. We ran Prodigy with heuristics suggested in Stone et al. [1994] and by Carbonell [Carbonell, personal communication]. It is somewhat unfair to compare exact running times because the planners are written in different languages (**Graphplan** is written in C while the other planners are in compiled Lisp), though partly because of this we ran Prodigy and UCPOP on a faster machine with more memory: we ran **graphplan** on a DECstation 2100 and the other planners on a SPARC10. Nonetheless, we can gain useful information from the curvature of plots of problems size versus time, as well as by comparing other objective measures. In particular, in addition to running time, we also report for **Graphplan** the number of goal-set creation steps (the number of times it creates a goal set at time $t - 1$ from a goal set at time t) and the total

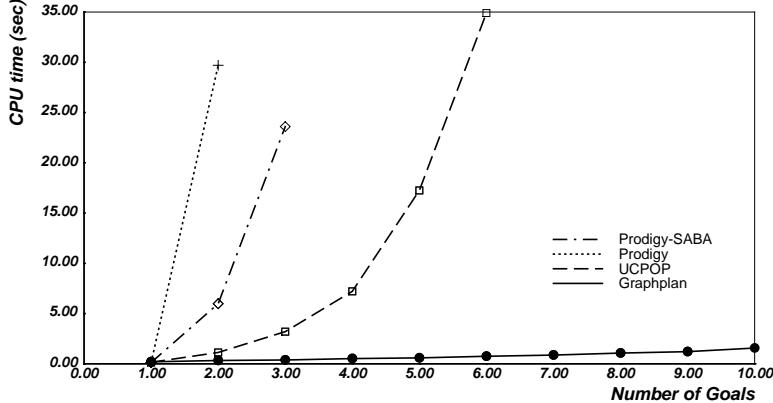


Figure 3: 2-Rockets problem

number of times that it selects a non-noop action to try in its search. These are somewhat analogous to the backward-chaining steps taken by total-order planners.

4.1.1 Rocket

We ran the planners on the rocket domain described in Figure 1 with the following setup. The initial conditions have 3 locations (London, Paris, JFK), two rockets, and n items of cargo. All the objects (rockets and cargo) begin at London and the rockets have fuel. The goal is to get $\lceil n/2 \rceil$ of the objects to Paris and $\lfloor n/2 \rfloor$ of the objects to JFK. The goals are ordered alternating between destinations.

Results of the experiment are in Figure 3. Notice that **Graphplan** significantly outperforms the other two planners on this domain. **Graphplan** does well in this domain for two main reasons: (1) the Planning Graph only grows to 3 time steps, and (2) the mutual exclusion relations allow a small number of commitments (unloading something from Rocket1 in Paris and something else from Rocket2 in JFK) to completely force the remainder of the decisions. In particular, **Graphplan** performs only two goal-set creation steps regardless of the number of goals, and the number of non-noop actions tried is linear in the number of goals. The size of the graph created is also linear in the number of goals: there are 150 nodes total for the problem with two goals, and 37 additional nodes per goal from then on.

The running time of **Graphplan** is completely unaffected by goal ordering for this problem.

4.1.2 Flat Tire

A natural problem of a different sort is Stuart Russell’s “fixing a flat tire” scenario (domain `init-flat-tire`, problem `fixit` in the UCPOP distribution). Unlike the rocket domain, a valid plan for solving this problem requires at least 12 time steps (and 19 actions). While for the rocket domain, **Graphplan** would do pretty well even without the mutual exclusion propagation, here the mutual exclusions are critical and ensure that not too many goal sets will be examined. **Graphplan** solves this problem in 1.1 to 1.3 seconds depending on the goal ordering. The number of goal-set creation steps ranges from a minimum of 105 to a maximum of 246, and the number of non-noop actions tried ranges from 170 to 350. The final graph created contains 786 nodes. Neither UCPOP nor Prodigy found a solution

Step 1:	open boot
Step 2:	fetch wrench boot
	fetch pump boot
	fetch jack boot
	fetch wheel2 boot
Step 3:	inflate wheel2
	loosen nuts the-hub
Step 4:	put-away pump boot
	jack-up the-hub
Step 5:	undo nuts the-hub
Step 6:	remove-wheel wheel1 the-hub
Step 7:	put-on-wheel wheel2 the-hub
	put-away wheel1 boot
Step 8:	do-up nuts the-hub
Step 9:	jack-down the-hub
Step 10:	put-away jack boot
	tighten nuts the-hub
Step 11:	put-away wrench boot
Step 12:	close boot

Figure 4: *Graphplan’s plan for Russell’s “Fixit” problem.*

within 10 minutes for this problem in the standard goal ordering, though it is possible to find goal orderings where they succeed much more quickly. *Graphplan* is not only fast on this domain, but also by producing the shortest partial-order plan, its plan is intuitively “sensible”. Figure 4 shows the plan produced by *Graphplan* for this problem.

4.1.3 Monkey and Bananas

The UCPOP distribution provides three “Monkey and Bananas” problems (originally from Prodigy). Two have a solution and the third does not. Srinivasan and Howe [Srinivasan and Howe, 1995] show experimental results for a variety of partial-order planning heuristics on this domain. They report average running times (on a SPARC IPX, in Common Lisp) of about 90 seconds for most of the methods, though one took 2000 seconds and one took only 30 seconds on average per problem. They report an average number of plans examined in those planners for a task called “flaw selection” ranging from 5,558 to 105,518. *Graphplan* solves these problems much more quickly, taking 0.7 seconds on the first, 3.4 seconds on the second, and 2.8 seconds on the unsolvable one (these times are on a DECstation 2100). *Graphplan* attempts only 6 non-noop actions in solving the first problem, and 90 on the second. On the unsolvable problem, *Graphplan* extends its graph to 7 time steps, at which point it notices that the problem is unsolvable because the graph has “leveled off” and yet there still remain exclusive goals (see Section 5). Thus, on this problem *Graphplan* is able to report that the problem is unsolvable without actually performing any search.

On all three problems, most of the time spent is in graph creation. The graphs for the three problems contain 304, 824, and 700 nodes, respectively.

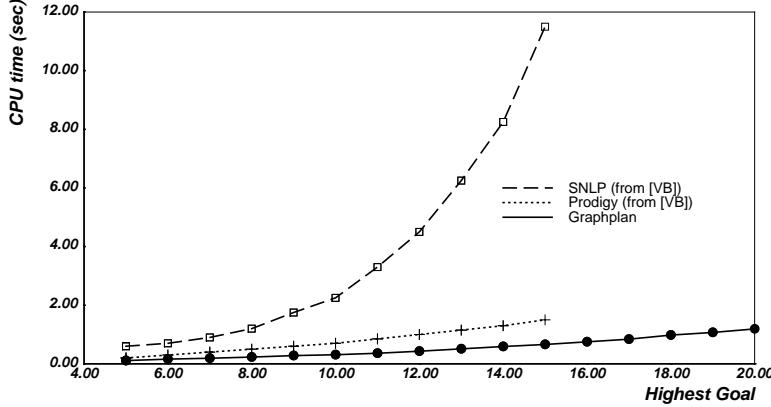


Figure 5: *Link-repeat domain from (Veloso & Blythe 1994)*

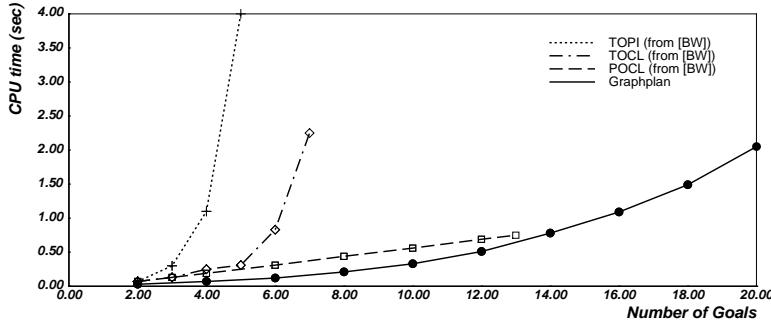


Figure 6: *D¹S¹ domain from (Barrett & Weld 1994)*

4.1.4 The Fridge Domain

The UCPOP distribution provides two “refrigerator fixing” domains. On the first one, **Graphplan** takes 4.0 seconds, performs 2 goal-set creation steps, and attempts 7 non-noop actions. On the second one **Graphplan** takes 11.3 seconds, performs 46 goal-set creation steps, and attempts 258 actions. On these two problems, the graphs created contain 287 and 686 nodes, respectively.

Srinivasan and Howe [1995] report times ranging from 30 to 300 seconds and average number of plans examined from about 9700 to 42000 for the different methods they consider.

4.2 Artificial domains

Barrett and Weld [1994] and Veloso and Blythe [1994] define a collection of artificial domains intended to distinguish the performance characteristics of various planners. On all of these, **Graphplan** is quite competitive with the best performance reported.

We present in Figures 5, 6, 7, and 8 performance data on four of the more interesting domains. All performance results in these figures for the other planners are taken from figures in their respective papers. (Note: in Figure 8, the TOCL and POCL curves effectively coincide.)

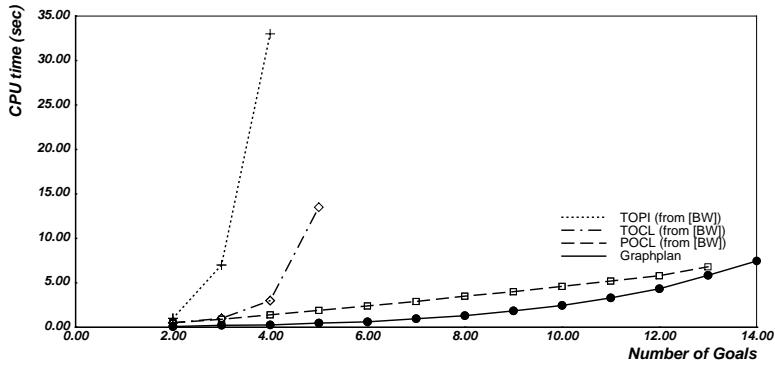


Figure 7: D^1S^2 domain from (Barrett & Weld 1994)

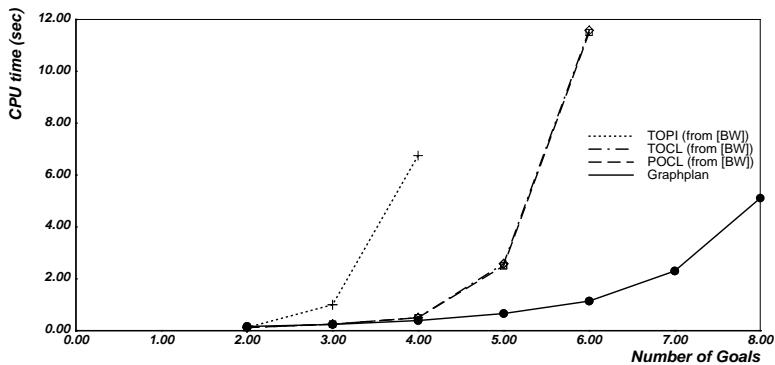


Figure 8: D^mS^{2*} domain from (Barrett & Weld 1994)

4.3 Discussion of Experimental Results

Four major factors seem to account for most of **Graphplan**'s efficiency. They are, in order of empirically-derived importance:

Mutual Exclusion: In many of the examples, the pairwise mutual exclusions relations are able to represent most of the important constraints in the planning problem. (E.g., see the discussion in Section 2.2.1.) Propagating these constraints effectively prunes a large part of the search space.

Consideration of Parallel Plans: In some cases, such as the rocket problem, the valid parallel plans are relatively short compared with the length of the corresponding totally-ordered plans. In such cases neither the cost of Planning Graph construction, nor the cost of search is very large.

Memoizing: By fixing actions at specific points in time, **Graphplan** is able to record the goal sets that it proves to be unreachable in a certain number of time steps from the initial conditions.

Low-level costs: By constructing a Planning Graph in advance of search, **Graphplan** avoids the costs of performing instantiations during the searching phase.

Furthermore, it is worth noting that graph creation is quite fast (as well as being provably polynomial time) and only takes up a significant fraction of the total time on the simpler problems where the total running time is quite short in any case.

5 Terminating on Unsolvable Problems

To a first approximation, **Graphplan** conducts something like an iteratively-deepened search. In the i^{th} stage the algorithm sees if there is a valid parallel plan of length less than or equal to i . As described so far, if no valid plan exists there is nothing that prevents the algorithm from mindlessly running forever through an infinite number of stages.

We now describe a simple and efficient test that can be added after every unsuccessful stage so that if the problem has no solution then **Graphplan** will eventually halt and say “No Plan Exists.”

5.1 Planning Graphs “Level Off”

Assume a problem has no valid plan. First observe that in the sequence of Planning Graphs created there will eventually be a proposition level P such that all future proposition levels are exactly the same as P , *i.e.*, they contain the same set of propositions and have the same exclusivity relations.

The reason for this is as follows. Because of the no-op actions, if a proposition appears in some proposition level then it also appears in all future proposition levels. Since only a finite set of propositions can be created by STRIPS-style operators (when applied to a finite set of initial conditions) there must be some proposition level Q such that all future levels have exactly the same set of propositions as Q . Also, again because of the no-op actions, if propositions p and q appear together in some level and are *not* marked as mutually exclusive,

then they will not be marked as mutually exclusive in any future level. Thus there must be some proposition level P after Q such that all future proposition levels also have exactly the same set of mutual exclusion relations as P .

In fact, it is not hard to see that once two adjacent levels P_n, P_{n+1} are identical, then all future levels will be identical to P_n as well. At this point, we say the graph has *leveled off*.

5.2 A Quick and Easy Test

Let P_n be the first proposition level at which the graph has leveled off. If some Problem Goal does not appear in this level, or if two Problem Goals are marked as mutually exclusive in this level, then **Graphplan** can immediately say that no plan exists. Notice that in this case, **Graphplan** is able to halt without performing any search at all. This is what happened with the unsolvable “Monkey and Bananas” problem discussed in Section 4.1.3. However, in some cases it may be that no plan exists but this simple test does not detect it. A nice example of this is a blocks world with three blocks, in which the goals are for block A to be on top of block B, block B to be on top of block C, and block C to be on top of block A; any two of these goals are achievable but not all three simultaneously. So we need to do something slightly more sophisticated to guarantee termination in all cases.

5.3 A Test to Guarantee Termination

As mentioned earlier, **Graphplan** memoizes, or records, goal sets that it has considered at some level and determined to be unsolvable. Let S_i^t be the collection of all such sets stored for level i after an unsuccessful stage t . In other words, after an unsuccessful stage t , **Graphplan** has determined two things: (1) any plan of t or fewer steps must make one of the goal sets in S_i^t true at time i , and (2) none of the goal sets in S_i^t are achievable in i steps. The modification to **Graphplan** ensure termination is now just the following:

If the graph has leveled off at some level n and a stage t has passed in which $|S_n^{t-1}| = |S_n^t|$, then output “No Plan Exists.”

Theorem 3 *Graphplan outputs “No Plan Exists” if and only if the problem is unsolvable.*

Proof. The easy direction is that if the problem is unsolvable, then **Graphplan** will eventually say that no plan exists. The reason is just that the number of sets in S_n^t is never *smaller* than the number of sets in S_n^{t-1} , and there is a finite maximum (though exponential in the number of nodes at level n).

To see the other direction, suppose the graph has leveled off at some level n and **Graphplan** has completed an unsuccessful stage $t > n$. Notice that any plan to achieve some set in S_{n+1}^t must, one step earlier, achieve some set in S_n^t . This is because of the way **Graphplan** works: it determined each set in S_{n+1}^t was unsolvable by mapping it to sets at time step n and determining that they were unsolvable. Notice also that since the graph has leveled off, $S_{n+1}^t = S_n^{t-1}$. That is because the last $t - n$ levels of the graph are the same no matter how many additional levels the graph has.

Now suppose that after an unsuccessful stage t , $|S_n^{t-1}| = |S_n^t|$ (which implies that $S_n^{t-1} = S_n^t$). This means that $S_{n+1}^t = S_n^t$. Thus, in order to achieve any set in S_{n+1}^t one must

previously have achieved some other set in S_{n+1}^t . Since none of the sets in S_{n+1}^t are contained in the initial conditions, the problem is unsolvable. ■

6 Additional Features

We have discussed so far the basic algorithm used by **Graphplan**. We now describe a few additional features that can be added in a natural way (and have been added as options in our implementation), and discuss their significance.

The first feature is a type of reasoning that is quite natural in our framework. The reasoning is that if the current goal set contains n goals such that no two of them can be made true at the same time by a non-noop action (and none of them are present in the Initial Conditions), then any plan will require at least n steps. For instance, one could use this reasoning in a path-finding domain to show that it must take at least n steps to visit n distinct places. Unfortunately, finding the largest such subset of any given goal set is equivalent to the maximum Clique problem (think of there being a “can’t both be created now” edge between any two propositions that cannot both be made true in the same step). However, we can find a *maximal* such set using greedy methods.

This form of reasoning turns out to be very useful on traveling-salesman-like problems, where the goal is to visit all the nodes in a graph in as few steps as possible. On very dense graphs (such as the complete graph) for which the problem should be easy, **Graphplan** without this reasoning can be quite slow because the pairwise exclusion relations do not propagate well. For instance, on a complete graph, after two time steps any two goals of the form ‘visited X’ will be non-exclusive. However, with this reasoning, **Graphplan**’s performance is more respectable.

A second feature concerns graph creation. Although, as demonstrated in Theorem 1, the graph size is polynomial, it may be unnecessarily large if there are many irrelevant facts in the initial conditions. One way around this problem is to begin with a regression analysis going backward from the goals to determine if any initial conditions may be thrown out. For instance, if our rocket problem contains in the initial conditions a “junkyard” of rockets with no fuel, or some number of irrelevant observers, this method can identify them and set them aside. Of course, performing this regression analysis itself takes some amount of time.

One final feature (not currently in our implementation) that could be added easily is the ability to use the information learned on one planning problem for another problem on the same domain having the same Initial Conditions. Specifically, the same graph and the same memoized unsolvable goal sets could be re-used in this case.

7 Discussion and Future Work

We have described a novel planning algorithm, **Graphplan**. This algorithm uses ideas from standard total-order and partial-order planners, but differs most significantly by taking the position that representing the planning problem in a graph structure — a structure one can analyze, annotate, and play with — can significantly improve efficiency. Performance on the problems we have tried indicate that indeed this can provide a big savings.

We believe that even more significant gains will come from combining the approach of Graphplan with ideas, heuristics, and learning methods that have been developed in the planning literature. Specifically, directions we are currently considering include:

Learning: Learning techniques found to be useful for other planning methods (e.g., [Etzioni, 1990]) may work here as well. In addition, perhaps the new representation used here will suggest other learning approaches not considered previously.

Symmetry detection: Many of the times that planners behave poorly are times when symmetries exist in a problem that the planner does not utilize. Representing the planning problem as a graph may allow for new methods of detecting symmetries that could drastically reduce the search needed.

Two-way searches: Some problems are more easily solved in the forward direction than in the reverse. Prodigy, for instance, is able to create a plan in a forward direction even while it searches from the goals. We would like to incorporate some method for planning in a similar manner. This might involve memoizing solvable goal sets as well as unsolvable ones.

Other information to propagate: Graphplan propagates pairwise exclusion relations in order to speed up its search. There may be other sorts of information that could be propagated forward or backward through the graph that would be useful as well.

Using max-flow algorithms: The original motivation for our approach was that planning graphs, with slight modification, allow one to think of planning as a certain kind of maximum flow problem.⁶ The view of planning as a flow problem requires additional constraints that make the problem NP-hard (in particular, a constraint that certain edges be either unused or else fully saturated, corresponding to the fact that an action may be performed or not performed, but cannot be “partially performed”). Nonetheless, perhaps algorithms for the max-flow problem — and there are many fast algorithms known [Cormen *et al.*, 1990, Goldberg and Tarjan, 1986] — might be useful for guiding the planning process. Empirically, we found that an approach based solely on max-flow algorithms did not perform as well as the method of backward-chaining with mutual exclusion relations described in this paper. A flow-based method, however, may allow one to naturally incorporate other aspects of a planning problem, such as having different costs associated with different actions, in a natural way. We are currently exploring whether flow algorithms can be *combined* with our current approach to improve performance.

7.1 Limitations and Open Problems

One main limitation of Graphplan is that it applies only to STRIPS-like domains. In particular, actions cannot create new objects and the effect of performing an action must be something that can be determined statically. There are many kinds of planning situations

⁶In this problem, one is given a graph containing source and sink nodes, and each edge is labeled with a capacity representing the maximum amount of fluid that may flow across that edge. In a legal flow, for every node except the source or sink, the flow in must equal the flow out. The goal is to flow as much fluid as possible from the source to the sink, without exceeding any of the capacities.

that violate these conditions. For instance, if one of the actions allows the planner to dig a hole of an arbitrary integral depth, then there are potentially infinitely many objects that can be created. Or, suppose we have the action “paint everything in this room red.” The effect of this action cannot be determined statically: the set of objects painted red depends on which happen to be in the room at the time. One open question is whether the Planning Graph Analysis paradigm can be extended to handle settings with these sorts of actions.

A second limitation is that roughly, in order to perform well **Graphplan** requires either that the pairwise mutual exclusion relations capture important constraints of the problem, or else that the ability to perform parallel actions significantly reduces the depth of the graph. Luckily, it appears that at least one of these tends to be true in many natural problems. Section 6 discussed one case (a simple TSP problem), however, in which neither of these occurs and **Graphplan** performs poorly without extra ad-hoc reasoning capabilities. Perhaps additional more powerful types of constraints can be added to **Graphplan** to overcome some of these difficult cases.

Finally, one last limitation worth mentioning is that by guaranteeing to find the *shortest* possible plan, **Graphplan** can make problems more difficult for itself. For instance, when people solve the 16-puzzle, they usually do so using a methodical approach that is easy to perform, but does not guarantee the solution with the fewest moves. If one had to find the solution with the fewest moves, it would be more difficult. Perhaps tradeoffs of this form between plan quality and the speed of planning could be incorporated into the Planning Graph Analysis paradigm.

Accessing Graphplan

Graphplan, including source code, sample domains, and several animations is available via <http://www.cs.cmu.edu/~avrim/graphplan.html>.

Acknowledgements

We thank Jaime Carbonell and the members of the CMU Prodigy group for their helpful advice.

References

- [Barrett and Weld, 1994] A. Barrett and D. Weld. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.
- [Blum and Furst, 1995] A. Blum and M. Furst. Fast planning through planning graph analysis. In *IJCAI95*, pages 1636–1642, Montreal, 1995.
- [Bylander, 1994] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.
- [Carbonell, personal communication] J. Carbonell. 1994. personal communication.
- [Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

- [Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
- [Etzioni, 1990] O. Etzioni. *A Structural theory of explanation-based learning*. PhD thesis, CMU, December 1990. CMU-CS-90-185.
- [Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [Goldberg and Tarjan, 1986] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 136–146, 1986.
- [Knoblock, 1994] C. Knoblock. Generating parallel execution plans with a partial-order planner. In *AIPS94*, pages 98–103, Chicago, 1994.
- [McAllester and Rosenblitt, 1991] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pages 634–639, July 1991.
- [Srinivasan and Howe, 1995] R. Srinivasan and A. Howe. Comparison of methods for improving search efficiency in a partial-order planner. In *IJCAI95*, pages 1620–1626, Montreal, 1995.
- [Stone *et al.*, 1994] P. Stone, M. Veloso, and J. Blythe. The need for different domain-independent heuristics. In *AIPS94*, pages 164–169, Chicago, 1994.
- [Veloso and Blythe, 1994] M. Veloso and J. Blythe. Linkability: Examining causal link commitments in partial-order planning. In *AIPS94*, pages 164–169, Chicago, 1994.
- [Veloso, 1989] M. Veloso. Nonlinear problem solving using intelligent casual-commitment. Technical Report CMU-CS-89-210, Carnegie Mellon University, December 1989.
- [Weld, 1994] D. Weld. An introduction to partial-order planning. *AI Magazine*, 1994.

A Monte Carlo Approach for Football Play Generation

Kennard Laviers

School of EECS

U. of Central Florida

Orlando, FL

klaviers@eeecs.ucf.edu

Gita Sukthankar

School of EECS

U. of Central Florida

Orlando, FL

gitars@eeecs.ucf.edu

Abstract

Learning effective policies in multi-agent adversarial games is a significant challenge since the search space can be prohibitively large when the actions of all the agents are considered simultaneously. Recent advances in Monte Carlo search methods have produced good results in single-agent games like Go with very large search spaces. In this paper, we propose a variation on the Monte Carlo method, UCT (Upper Confidence Bound Trees), for multi-agent, continuous-valued, adversarial games and demonstrate its utility at generating American football plays for Rush Football 2008. In football, like in many other multi-agent games, the actions of all of the agents are not equally crucial to gameplay success. By automatically identifying key players from historical game play, we can focus the UCT search on player groupings that have the largest impact on yardage gains in a particular formation.

Introduction

One issue with learning effective policies in multi-agent adversarial games is that the size of the search space can be prohibitively large when the actions of all players are considered simultaneously. Hence, single-agent reinforcement learning systems are relatively common, whereas multi-agent learning systems are less prevalent. In this paper, we demonstrate a multi-agent learning approach for generating offensive plays in the Rush 2008 football simulator. Rush 2008 simulates a modified version of American football and was developed from the open source Rush 2005 game.¹

To succeed at American football, a team must be able to successfully execute closely-coordinated physical behavior. However, certain players, like the quarterback, clearly have a greater impact on the success of a given play. Yet changing the policy of the quarterback, without making corresponding changes to other teammates, can lead to a catastrophic failure of the play, whereas by simultaneously making changes in a subset of the players (e.g., the quarterback and a receiver) can amplify yardage gains.

Within the Rush football simulator, we observe that each play relies on the efforts of different subgroups within the main team to score team touchdowns. We devised a method

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

¹<http://sourceforge.net/projects/rush2005>

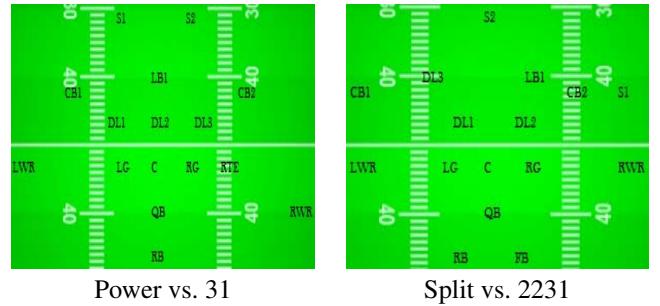


Figure 1: Two match-ups between offensive (bottom) and defensive (top) formations. These lead to a variety of plays, in which different subsets of players can affect the play outcome. We identify frequently-observed coordination patterns from observed traces to guide multi-agent policy search.

to automatically identify these subgroups from historical play data based on: 1) mutual information between the offensive player, defensive blocker, and ball location 2) the observed ball work flow. After extracting these subgroups, we demonstrate how to determine which groups are the most important and use those groups to focus the search for new plays.

Recent advances in Monte Carlo search methods have produced learning agents which effectively play games with very large search spaces where evaluating the value of intermediate states is difficult. In particular, the UCT (Kocsis and Szepesvri 2006) algorithm has produced promising results in a variety of games, ranging from Go (Gelly and Wang 2006) to the real-time strategy game WARGUS (Balla and Fern 2009). Using knowledge of key groups to bound search, we illustrate how UCT can be used to create new playbooks with user-defined levels of difficulty from novice to expert.

Rush Football

Football is a contest of two teams played on a rectangular field bordered on lengthwise sides by an end zone. Unlike American football, Rush teams only have 8 players on the field at a time out of a roster of 18 players, and the field is 100 × 63 yards. The game's objective is to out-score the op-

ponent, where the offense (i.e., the team with possession of the ball), attempts to advance the ball from the line of scrimmage into their opponent's end zone. In a full game, the offensive team has four attempts to get a *first down* by moving the ball 10 yards down the field. If the ball is intercepted or fumbled and claimed by the defense, ball possession transfers to the defensive team. The Rush 2008 simulator only runs one play with the line of scrimmage set to the center of the field. Stochasticity exists in catching (i.e., whether a catch is successful), fumbling, tackles, distance/location of a thrown ball, and the selection of who to throw to if no receiver is "open" when the QB is forced to throw the ball.

The offensive lineup contains the following positions:

Quarterback (QB): given the ball at the start of each play.

The QB hands the ball off or passes it to another player.

Running back (RB): begins in the backfield, behind the line of scrimmage where the ball is placed, with the quarterback and fullback.

Full back (FB): serves largely the same function as the RB.

Wide receiver (WR): primary receiver for pass plays.

Tight end (TE): begins on the line of scrimmage immediately to the outside of the offensive lineman and can receive passes.

Offensive linemen (OL): begin on the line of scrimmage and are primarily responsible for preventing the defense from reaching the ball carrier.

A Rush play is composed of (1) a starting formation and (2) instructions for each player in that formation. A formation is a set of (x,y) offsets from the center of the line of scrimmage. By default, instructions for each player consist of (a) an offset/destination point on the field to run to, and (b) a behavior to execute when they get there. Play instructions are similar to a conditional plan and include choice points where the players can make individual decisions as well as pre-defined behaviors that the player executes to the best of their physical capability. Rush includes three offensive formations (**power**, **pro**, and **split**) and four defensive ones (**23**, **31**, **2222**, **2231**). Each formation has eight different plays (numbered 1-8) that can be executed from that formation. Offensive plays typically include a handoff to the running back/fullback or a pass executed by the quarterback to one of the receivers, along with instructions for a running pattern to be followed by all the receivers. An example play from the **split** formation is given below:

- the quarterback will pass to an open receiver;
- the running back and fullback will run hook routes;
- the left wide receiver will run a corner right route;
- the right wide receiver will run a hook route;
- the other players will block for the ball holder.

Related Work

Rush 2008 was originally developed as a platform for evaluating game-playing agents and has been used to study the problem of learning strategies by observation of real-world

football games (Li *et al.* 2009). A case-based reinforcement learning approach for modifying the quarterback's policy was demonstrated within Rush 2008 (Molineaux *et al.* 2009); note that all of the other players, other than the quarterback, always played the same strategy. (Laviers *et al.* 2009) created an online multi-agent play adaptation system that modified the team's policies in response to the opponents' play. However, the play adaptation system always utilized plays from the existing playbook, rather than creating new plays.

Monte Carlo rollout search algorithms have been used successfully in a number of games (Chung *et al.* 2005; Cazenave and Paris 2005; Cazenave 2009; Ward and Cowling 2009). The Upper Confidence Bound Tree (UCT) was introduced in (Kocsis and Szepesvri 2006) and spawned a host of research efforts (Gelly *et al.* 2006; Gelly and Wang 2006; Gelly and Silver 2007). Our work differs from previous UCT work in its use of key player subgroups to focus search in a multi-agent, continuous-value domain.

A key element of our approach is the use of automatically extracted coordination patterns from historical play data. In the Robocup soccer domain, (Iravani 2009) performed an in-depth analysis of interactions among players to identify multi-level networks. This is conceptually very closely related to the idea of grouping players by movement and workflow patterns. His system constructed agent interactions between players based on closest teammate, Voronoi regions and distance-based clusters. The identified networks were used to model the teams and identify how interactions affect team performance; however these networks were not incorporated into a play generation system.

Method

The basic idea behind our approach is to identify subgroups of coordinated players by observing a large number of football plays. Earlier work, such as (Laviers *et al.* 2009), has shown that appropriately changing the behavior of a critical subgroup (e.g., QB, RB, FB) during an offensive play, in response to a recognized defensive strategy, significantly improves yardage. Our work automatically determines the critical subgroups of players (for each play) by an analysis of spatio-temporal observations to determine all sub-groups, and supervised learning to learn which ones will garner the best results.

Identifying Key Players

In order to determine which players should be grouped together we first must understand dependencies among the eight players for each formation. All players coordinate to some extent but some players' actions are so tightly coupled that they form a *subgroup* during the given play. Changing the command for one athlete in a subgroup without adjusting the others causes the play to lose cohesion, potentially resulting in a yardage loss rather than a gain. We identify subgroups using a combination of two methods, the first based on a statistical analysis of player trajectories and the second on workflow.

The mutual information between two random variables measures their statistical dependence. Inspired by this, our

method for identifying subgroups attempts to quantify the degree to which the trajectories of players are coupled, based on a set of observed instances of the given play. However, the naive instantiation of this idea, which simply computes the dependence between player trajectories without considering the game state is doomed to failure. This is because offensive players' motions are dominated by three factors: 1) its plan as specified by the playbook, 2) the current position of the ball, and 3) the current position of the defensive player assigned to block him.

So, if we want to calculate the relationships between the offensive players, we need to place their trajectories in a context that considers these factors. Our method for doing this is straightforward. Rather than computing statistics on raw player trajectories, we derive a feature that includes these factors and compute statistics between the feature vectors as follows.

First, for each player on the offense, we determine the trajectory of the defensive player assigned to block him. Since this assigned defensive player is typically the opponent that remains closest to the player during the course of the play, we determine the assigned defender to be the one whose average distance to the given player is the least. More formally, for a given offensive player, $o \in \{o_1, \dots, o_8\}$, the assigned defender, $d \in \{d_1, \dots, d_8\}$ is:

$$d = \operatorname{argmin}_{d_i} \sum_{t=1}^T |o(t) - d_i(t)|_2,$$

where $o(t)$ and $d_i(t)$ denote the 2D positions of the given players at time t . Our feature $f(t)$ is simply the centroid (average) of $o(t)$, $d(t)$ and the ball position $b(t)$. We can now compute sets of features $\{f_i\}$ and $\{f_j\}$ from the collection of observed plays for a given pair of offensive players o_i and o_j , treating observations through time simply as independent measurements. We model the distributions F_i and F_j of each of these features as 2D Gaussian distributions with diagonal covariance.

We then quantify the independence between these feature distributions using the symmetricized Kullback-Leibler divergence (Kullback and Leibler 1951):

$$S(o_i, o_j) = D_{KL}(F_i||F_j) + D_{KL}(F_j||F_i),$$

where

$$D_{KL}(F_i||F_j) = \sum_k F_i(k) \log \left(\frac{F_i(k)}{F_j(k)} \right).$$

Pairs of athletes with low $S(\cdot)$ are those whose movements during a given play are closely coupled. We compute the average $S(\cdot)$ score over all pairs (o_i, o_j) in the team and identify as candidate subgroups those pairs whose score falls in the lowest quartile.

The grouping process involves more than just finding the mutual information between players. We must also determine relationships formed based on possession of the football. When the quarterback hands the ball off to the running back or fullback their movements are coordinated for only a brief span of time before the ball is transferred to the next

player. Because of this, the mutual information (MI) algorithm described above does not adequately capture this relationship. We developed another mechanism to identify such *workflows* and add them to the list of MI-based groups.

Our characterization of the workflow during a play is based on ball transitions. Given our dataset, we count transitions from one player to another. The historical data indicates that, in almost all offensive formations, the RB receives the ball the majority of the time, except when the FB is in play and in which case we see the ball typically passed from the QB to either the RB or the FB. Consequently, the {QB, RB, and FB} naturally forms as a group for *running plays* which was identified in (Laviers *et al.* 2009) as a “key group”. The same happens between the QB and the receiving player in *passing plays*, which forms another workflow group {QB, LWR, RWR, and RTE}. The final list of candidates is therefore simply the union of the MI candidates and the workflow candidates.

To use these subgroups we learn a prediction for the yardage impact of changing different extracted subgroups. For these studies, we compared the performance of several supervised classifiers and selected the K* instance-based algorithm, which is similar to Knn, but uses an entropy-based distance measure. To generate a training set, we ran the Rush 2008 football simulator on 450 randomly selected play variations. As the input features, we use the presence of possible observable offensive player actions (runningTo, carryingBall, waiting, charging, blocking, receivingPass, and sweeping); training is performed using 10-fold cross-validation.

Play Generation using UCT

We use Monte Carlo UCT (Kocsis and Szepesvri 2006) to generate new football plays. Using the top-ranked extracted subgroups to focus action investigations yields significant run-time reduction over a standard Monte-Carlo UCT implementation. To search the complete tree without using our subgroup selection method would require an estimated 50 days of processing time as opposed to the 4 days required by our method.

Offensive plays in the Rush 2008 football simulator share the same structure across all formations. Plays start with a **runTo** command which places a player at a strategic location to execute another play command. After the player arrives at this location, there is a decision point in the play structure where an offensive action can be executed. To effectively use a UCT style exploration we needed to devise a mechanism for combining these actions into a hierarchical tree structure where the most important choices are decided first.

Because of the potentially prohibitive number of possible location points, we have UCT initially search through the possible combinations of offensive high-level commands for the key players, even though chronologically the commands occur later in the play sequence. Once the commands are picked for the players, the system employs binary search to search the **runTo** area for each of the players (Figure 2). The system creates a bounding box around each players' historical **runTo** locations, and at level 2 (immediately after the high-level command is selected), the bounding box is split

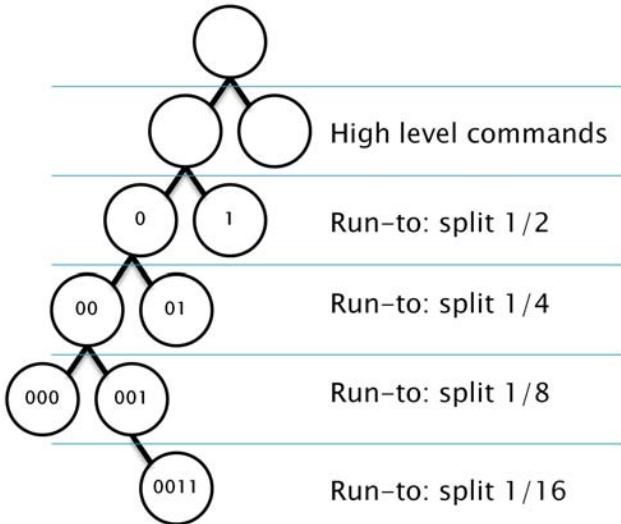


Figure 2: A representation of the UCT sparse tree.

in half. Following Monte Carlo expansion the location is initially randomly selected (Figure 5). At level 3 the space is again divided in half and the process continues until level 5 where the player is provided a **runTo** location which represents 1/16 of the bounding box area. The system takes a sample only at the leaf.

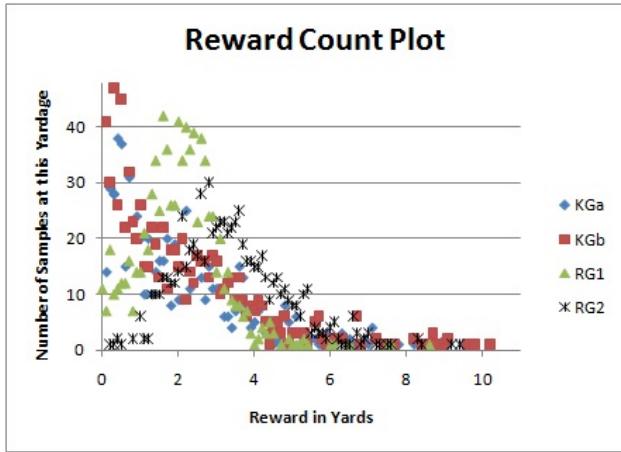


Figure 3: Comparison of randomly selecting players for action modifications vs. using the top-ranked subgroup. We see in the case of the random groups (R1 and R2) most of the reward values are close to the baseline of 2.8 yards while the key groups (K1 and K2) are dispersed more evenly across the yardage spectrum. This indicates that changing those players has a greater impact on the play.

This two dimensional search was designed to maintain as small a sampling as possible without harming the system's chance of finding solutions which produce large yardage gains. To focus the search, the locations each player can move to are bounded to be close (within 1 yard) to the re-

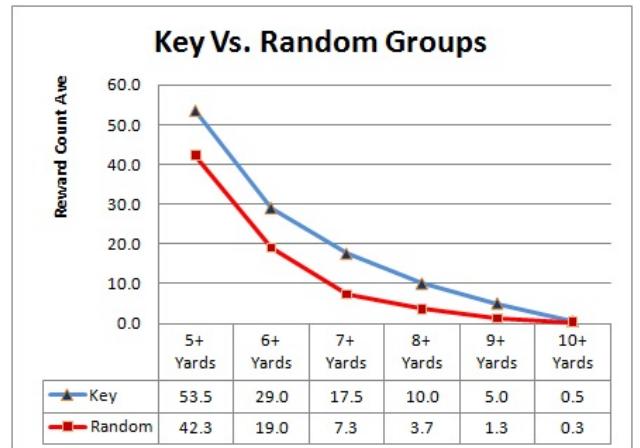


Figure 4: A detailed breakdown of the reward values above 5 yards. This chart clearly indicates that at the higher yardage spectrum the key groups provide a significantly greater number of solutions than the random groups.

gion covered by the specific player in the training data. At the leaf node the centroid of the square is calculated and the player uses that location to execute the **runTo** command. Our method effectively allows the most important features to be searched first and the least important, last.

As mentioned, action modifications are limited to the players in the top ranked subgroup identified using K^* ; the other players execute commands from the original play. Our system needs to determine the best plan over a wide range of opponent defensive configurations. To do this, for each rollout the system randomly samples 50% of all possible defenses (evens or odds, one for testing and the other for training) and returns the average yardage gained in the sampling. Since UCT method provides a ranked search with the most likely solutions grouped near the start of the search, we limit the search algorithm to 1000 nodes with the expectation that a good solution will be found in this search space.

We perform action selection using a variant of the UCT formulation, $\pi(s, a) = \text{argmax}_a(Q^+(s, a))$, where π is the policy used to choose the best action a from state s . Before revisiting a node, each unexplored node from the same branch must have already been explored; selection of unexplored nodes is accomplished randomly. We demonstrate that it is important to correctly identify key players for the formation by examining the effect of randomly selecting players for action modification on the value of $Q(s, a)$ (Figure 3).

Using a similar modification to the bandit as suggested in (Balla and Fern 2009), we adjust the upper confidence calculation $Q^+(s, a) = Q(s, a) + c \times \sqrt{\frac{\log n(s)}{n(s, a)}}$ to employ $c = Q(s, a) + .\varsigma$ where $\varsigma = .0001$ for our domain. The ς causes the system to consider nodes explored less often with zero reward more than nodes with a greater number of repeated zero rewards. Ultimately this allows us to account for the number of times a node is visited and prevent the search to remaining stuck on zero reward nodes, a problem

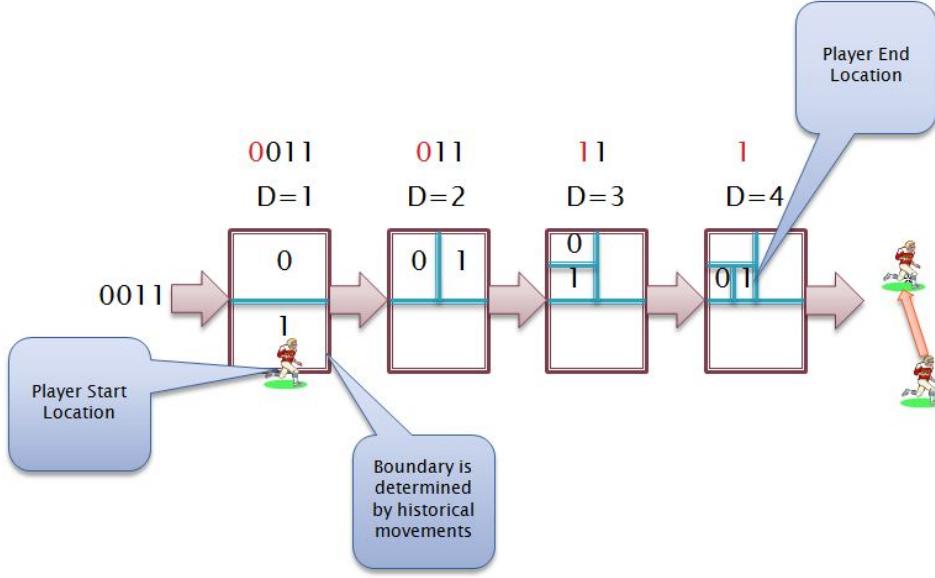


Figure 5: This graph shows how the binary string generated in the search tree creates a location for a player to move to in the `runTo` portion of the play.

we identified in early testing.

We implemented UCT in a distributed system constructed in such a way to prevent multiple threads from sampling the same node. The update function for $n(s, a)$ was modified to increment the counter after the node is visited, but before the leaf is sampled. Since sampling takes close to one second it's imperative to other threads exploring the tree to know when a node is touched to avoid infinite looping at a node.

After a node is sampled the update function is called to update $Q(s, a)$.

$$Q(s, a) \leftarrow Q(s, a) + \frac{1}{n(s, a)} (R - Q(s, a))$$

and

$$R = \frac{\sum_{i=0}^I \gamma_i}{I} / 15$$

where R is the reward, I is the total number of iterations times the number of defenses sampled, and γ is the list of yards gained in each sample. We normalize R by dividing by 15 which is 30% more than the maximum unnormalized reward.

Results

We evaluated the efficacy of this approach at generating passing plays, which require tightly coupled coordination between multiple players to succeed. Our version of UCT was seeded with Pro formation variants (4–8). Figure 7 summarizes experiments comparing UCT (limited by subgroup) against the baseline Rush playbook and play adaptation. Overall, the UCT plays consistently outperform the baseline Rush system and play adaptation using domain knowledge. Viewing the output trace of one UCT search reveals some characteristics of our algorithm. First the system randomly explores, then as promising nodes are found they are

exploited until UCT is confident it has a correct value for that branch before it moves on to other parts of the tree and repeats the process. Developers interested in automatically generating a list of plays can easily pick football plans which produce the level of difficulty the developer is interested in.

Conclusion

In this work, we presented a method for extracting subgroup coordination patterns from historical play data. We demonstrate our method in the Rush 2008 football simulator and believe that it can generalize to other team adversarial games, such as basketball and soccer. Although we have access to the ground truth playbook information executed by the simulator, the Rush plays do not explicitly specify subgroup coordination patterns between players. The subgroup patterns seem to emerge from the interactions between offensive and defensive formations creating different player openings which in turn affect play workflows when the quarterback chooses to pass or hand off to different players. We demonstrate that we can identify and reuse coordination patterns to focus our search over the space of multi-agent policies, without exhaustively searching the set partition of player subgroups. By sorting our candidate subgroups using a ranking learned by K^* , we can reliably identify effective subgroups and improve the performance and speed of the UCT Monte Carlo search.

Acknowledgments

This work was supported in part by DARPA grant HR0011-09-1-0023.

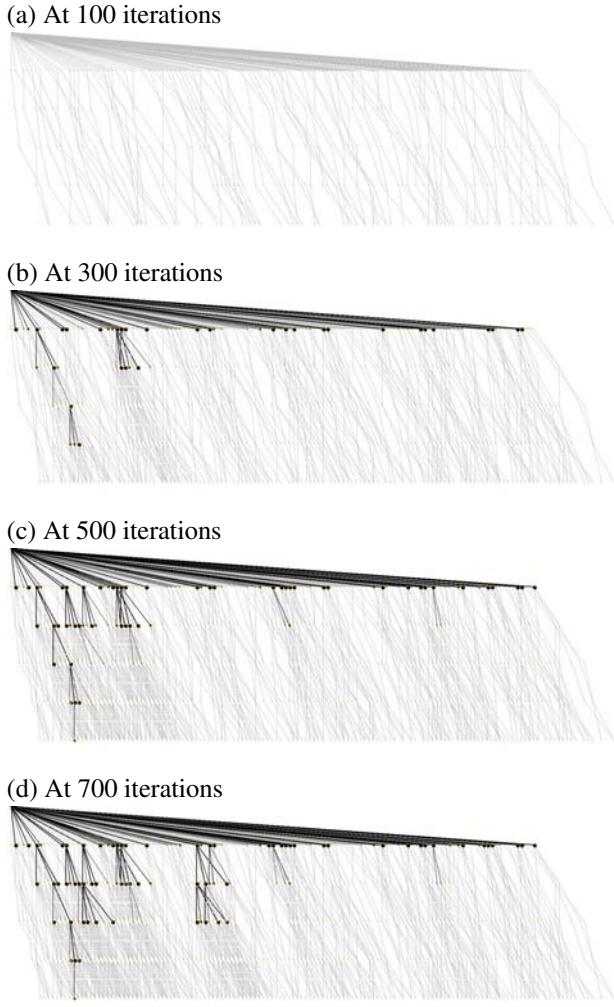


Figure 6: Our UCT variant expands in a very focused direction and quickly identifies a high yardage area of the tree.

References

- R. Balla and A. Fern. UCT for tactical assault planning in real-time strategy games. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2009.
- Tristan Cazenave and Labo Ia Universit Paris. Combining tactical search and Monte-Carlo in the game of Go. In *Proceedings of IEEE Symposium on Computational Intelligence and Games*, pages 171–175, 2005.
- Tristan Cazenave. Nested Monte-Carlo search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 456–461, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.
- Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte Carlo planning in RTS games. In *Proceedings of IEEE Symposium on Computational Intelligence and Games*, 2005.
- Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *In Zoubin Ghahramani, editor, Proceedings of the International Conference of Machine Learning (ICML) 2007*, pages 273–280, 2007.
- Sylvain Gelly and Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go, December 2006.
- Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA, 2006.
- P. Iravani. Multi-level network analysis of multi-agent systems. *RoboCup 2008: Robot Soccer World Cup XII*, pages 495–506, 2009.
- Levente Kocsis and Csaba Szepesvri. Bandit based Monte-Carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.
- S. Kullback and R. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.
- K. Lavers, G. Sukthankar, M. Molineaux, and D. Aha. Improving offensive performance through opponent modeling. In *Proceedings of Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE)*, 2009.
- N. Li, D. Stracuzzi, G. Cleveland, P. Langley, T. Konik, D. Shapiro, K. Ali, M. Molineaux, and D. Aha. Constructing game agents from video of human behavior. In *Proceedings of Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2009.
- M. Molineaux, D. Aha, and G. Sukthankar. Beating the defense: Using plan recognition to inform learning agents. In *Proceedings of Florida Artificial Intelligence Research Society*, 2009.
- C. D. Ward and P. I. Cowling. Monte Carlo search applied to card selection in Magic: The Gathering. In *Proceedings of IEEE Symposium on Computational Intelligence and Games*, pages 9–16, Piscataway, NJ, USA, 2009. IEEE Press.

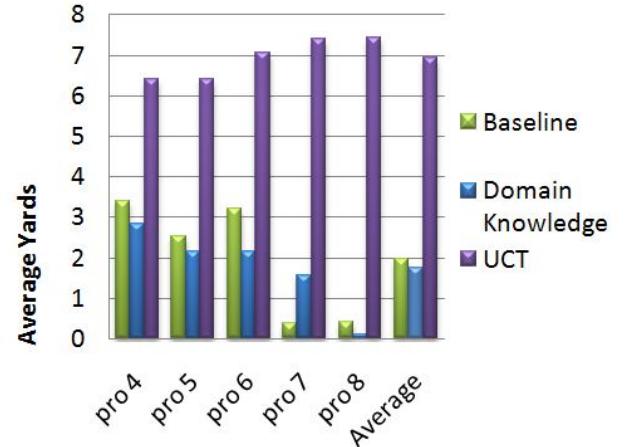
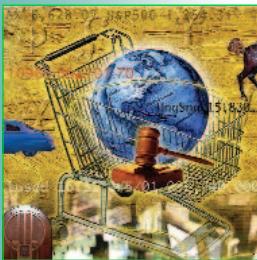


Figure 7: Comparison of multi-agent policy generation methods starting in the Pro formation. Our variant of UCT outperforms the baseline playbook and the domain knowledge play adaptation method.



Autonomous Bidding Agents in the Trading Agent Competition

Designing agents that can bid in online simultaneous auctions is a complex task. The authors describe task-specific details and strategies of agents in a trading agent competition.

Amy Greenwald
Brown University

Peter Stone
AT&T Labs—Research

A natural offshoot of the growing prevalence of online auctions is the creation of autonomous bidding agents that monitor and participate in these auctions. It is straightforward to write a bidding agent to participate in an online auction for a single good, particularly when the value of that good is fixed ahead of time: the agent can bid slightly over the ask price until the auction closes or the price exceeds the value. In simultaneous auctions offering complementary and substitutable goods, however, agent deployment is a much more complex endeavor.

The first trading agent competition (TAC), held in Boston, Massachusetts, on 8 July 2000, challenged participants to design a trading agent capable of bidding in online simultaneous auctions for complimentary and substitutable goods. TAC was organized by a group of researchers and developers led by Michael Wellman of the University of Michigan and Peter Wurman of North Carolina State University. In

a companion article, the tournament organizers present the design and operation of the competition.¹

This article describes the task-specific details of, and the general motivations behind, the four top-scoring agents. First, we discuss general strategies used by most of the participating agents. We then report on the strategies of the four top-placing agents. We conclude with suggestions for improving the design of future trading agent competitions.

General Game Strategies

A TAC game instance lasts 15 minutes and pits eight autonomous bidding agents against one another. Each TAC agent is a simulated travel agent with eight clients, each of whom would like to travel from TACtown to Boston and home again during a five-day period. Each client is characterized by a random set of preferences for arrival and departure dates, hotel rooms, and entertainment tickets. A TAC agent's objective is to maximize the total

utility (a monetary measure of the value of goods to clients) minus total expenses.

Agents have two basic activities: *bidding* and *allocating*. An agent bids, or offers payment, for goods to gain utility; an agent allocates purchased resources to clients to maximize total utility, both during and at the end of the game. To obtain utility for a client, a TAC agent constructs a travel package for that client by placing winning bids in simultaneous auctions for hotel reservations and flights. An agent can obtain additional utility by buying and selling entertainment tickets. After the auctions close, agents have four minutes to report their final allocations of goods to clients. A TAC agent's score is the difference between its clients' utilities and the agent's expenditures; a higher score indicates better performance. For full details, see <http://tac.eecs.umich.edu>.

Bidding Strategies

Figure 1 lists the basic decisions in the agents' inner bidding loops. Most decisions—what to buy, how many to buy, and how much to bid—showed several noteworthy differences. Only the timing of bids showed common features across agents.

Hotel auctions. In TAC auctions, the supply of flights is infinite and airline ticket prices are predictable, but the supply of hotel rooms is finite and hotel prices are unpredictable. Given the risks associated with the hotel reservation auctions, together with their importance in securing feasible travel packages, hotels were the most hotly contested items during the TAC competition. The timing of bidding in hotel auctions is particularly intriguing.

Hotel room prices have no set maximum value. Instead, TAC hotel auctions are ascending (English), m th-price, multiunit auctions and are subject to random closing times given sufficient levels of inactivity. (In a multiunit auction, m units of a good are available; in an m th price auction, the bidders with the m highest prices win the m units, all at the m th highest price.) Most TAC agents refrain from bidding for hotels early in the game unless the ask price has not changed recently, implying that the auction might close early, or the ask price is very low. Ultimately, the most aggressive hotel bidding takes place at the “witching hour”—in the final moments of the game—although precisely when is determined individually by each agent. More often than not, TAC hotel auctions reduce to m th price sealed-bid auctions, resulting in unpredictable final hotel prices that were often out of bounds.

```
(A) REPEAT
  1. Get market prices from server
  2. Decide on what goods to bid
  3. Decide at what prices to bid
  4. Decide for how many to bid
  5. Decide at what time to bid
  UNTIL game over
(B) Allocate goods to clients
```

Figure 1. High-level overview of a TAC agent's bidding decisions. TAC agents run an inner bidding loop, getting price updates from the server and making bidding decisions. Note that the order in which these decisions are made varies according to the agent's strategy.

Treating all current holdings of flights and entertainment tickets as sunk costs, the marginal utility of an as-yet-unsecured hotel room reservation is precisely the utility of the package itself. (Note that this observation holds only when the length of stay is exactly one night; for longer stays, it requires the further assumption that all other hotel rooms in the package are secured.) Throughout the preliminary competition, few agents bid their marginal utilities on hotel rooms. Those that did, however, generally dominated their competitors. Such agents were high bidders, bidding approximately \$1,000, always winning the hotels on which they bid, but paying a price far below their bid. Most agents adopted this high-bidding strategy during the actual competition. The result: many negative scores, since there were often greater than m high bids on a hotel room. In the final competition, the top-scoring TAC agents were those that not only bid aggressively on hotels, but also incorporated risk and portfolio management into their strategy to reduce the likelihood of buying highly demanded, overpriced hotel rooms.

The top-scoring agents were those that not only bid aggressively, but also incorporated risk and portfolio management into their strategy.

Flight auctions. Unlike hotel prices, prices for flights are predictable, with a maximum value of \$600. In particular, expected future prices equal current prices. Since airline prices periodically increase or decrease by a random amount chosen from the set $\{-10, -9, \dots, 9, 10\}$ with equal probability, the expected change in price for each airline auction is 0. (Indeed, it can be shown that if

Table 1. The agents' effectiveness in optimizing final allocations during the competition.

Agent	Strategy	Aggregate	Minimum	Number
ATTac	Optimal	100.0%	100.0%	13/13
RoxyBot	Optimal	100.0%	100.0%	13/13
Aster	Heuristic	99.6%	98.0%	9/13
UmbcTac	Greedy	99.4%	94.5%	7/13

Note: *Aggregate* is the percentage of the optimal utility (ignoring expenditures) achieved with the reported allocation, aggregated over the 13 games each of the top four agents played. *Minimum* is the minimum among these aggregated values. *Number* is the number of times the agent reported an optimal allocation. This information was provided by the TAC organizing team.

the airline auctions are considered in isolation, waiting until the end of the game to purchase tickets is an optimal strategy, except in the rare case where the price hits the lower bound on its value.) Thus, the auction design offers no incentive to bid on airline tickets before the witching hour, since by waiting there is some chance of obtaining information about hotel acquisitions.

There are, however, substantial risks associated with delaying bid submission. These risks arise from unpredictable network and server delays, and can cause bids placed during a game to be received after the game is over. To cope with these risks, most agents dynamically computed the length of their bidding cycles and then placed their flight bids some calculated amount of time before the end of a game. For example, a risk-averse agent might compute the average length of its three longest bidding cycles and then place its flight bids as soon as game time reaches 900 seconds minus twice that delay. A more risk-seeking agent might place its flight bids later, perhaps with game time of 900 seconds minus the minimum length of its five most recent bidding cycles. In practice, flight bids were placed anywhere from five minutes to 30 seconds before the end of the game.

Flight auctions are such that agents who place a winning bid pay not their bid but the current ask price. Thus, most agents bid above the current price—agents often bid the maximum—to ensure that these bids, which were placed at critical moments, were not rejected because of information delays resulting from network asynchrony.

Entertainment auctions. Agents' bidding strategies differ most substantially in auctions for entertainment tickets. While some agents focus on obtaining

complete packages, others make bidding decisions on travel packages alone (that is, flights and hotel rooms) without regard for entertainment packages, essentially breaking the TAC problem down into two subproblems and then solving greedily. The greedy approach, however, is not optimal. For example, if a client does not already have a ticket to an event, then it is preferable to extend the client's stay whenever the utility obtained by assigning that client a ticket to the event exceeds the cost of the ticket and an additional night at the hotel plus any travel penalties incurred. Similarly, it is sometimes preferable to sell entertainment tickets and shorten a client's stay accordingly.

Allocation Strategies

When allocating goods to their clients, most of the agents greedily focus on satisfying each of their clients in turn. Only the top two teams' bidding strategies incorporated global decision-making in which the interests of all their clients were considered simultaneously. (The third-scoring agent used a heuristically based intermediate approach.) This aspect of an agent's design also affects its final allocation algorithm. The percentage of optimal allocations reported by each agent during the competition is listed in Table 1.

Although simpler, the greedy strategy is not always optimal. For example, consider two clients, *A* and *B*, with identical travel preferences and the following entertainment preferences: *A* values the symphony at \$90, the theater at \$80, and baseball at \$70; *B* values the symphony at \$175, the theater at \$150, and baseball at \$125. Suppose each client will be in town on the same night and that one ticket for each entertainment type is for sale for \$50 on that night. An agent using a greedy approach who considers client *A* before *B* will assign *A* the ticket to the symphony and *B* the ticket to the theater, obtaining an overall utility of \$140. It would be optimal, however, to assign *A* the ticket to the theater and *B* the ticket to the symphony, yielding an overall utility of \$155.

Top-Scoring Agents' Strategies

In the remaining sections of this article, we describe the bidding, allocation, and completion strategies; team motivations; and any unique approaches of the four top-scoring agents.

ATTac: Adaptability and Principled Bidding

ATTac placed first by using a principled bidding strategy. Several elements of strategic adaptivity gave ATTac the flexibility to cope with a wide vari-

ety of possible scenarios during the competition. ATTac's design was motivated by its developers' interest in multiagent learning.

Bidding. At every bidding opportunity, ATTac begins by computing the most profitable allocation of goods to clients (denoted G^*), given the currently owned goods and the current prices of hotels and flights. (For hotels, ATTac actually uses predicted closing prices based on the results of previous game instances.) For this computation, ATTac allocates, but does not consider buying or selling, entertainment tickets. In most cases, G^* is computed optimally through mixed-integer linear programming.

ATTac bids in either passive or active mode. In passive mode, ATTac computes the average time it takes to place a bid, keeping its bidding options open until the witching hour. When the time left in the game equals twice the time of an average transaction, ATTac switches to active mode, during which it buys the airline tickets required by G^* and places high bids for the required hotel rooms. ATTac expects to run at most two bidding iterations in active mode.

Based on the current G^* , its current mode, and the average time of its transactions, ATTac bids for flights, hotel rooms, and entertainment tickets. Stone et al. detail ATTac's strategy²; we focus on bidding strategies for entertainment tickets.

On every bidding iteration, ATTac places a buy bid for each type of entertainment ticket and a sell bid for each type of entertainment ticket it currently owns. In all cases, prices depend on the amount of time left in the game, and become less aggressive as time goes on.

For each owned entertainment ticket E , if E is assigned in G^* , let $V(E)$ be the value of E to the client to whom it is assigned in G^* . Since \$200 is E 's maximum possible value to any client under the TAC parameters, ATTac offers to sell E for $\min(200; V(E) + \delta)$ where δ decreases linearly from 100 to 20 based on the time left in the game. ATTac uses a similar "sliding price" strategy for entertainment tickets that it owns but did not assign in G^* (because all clients are either unavailable that night or are already scheduled for that type of entertainment in G^*).

Finally, ATTac bids on each type of entertainment ticket (including those it is also offering to sell) based on the increased value of G^* that would be derived by owning it (that is, G^* is entirely recomputed with a hypothetical additional resource). Again, a sliding price strategy is used,

this time with the buy price increasing as the game proceeds. The sliding price strategy allows the agent to take advantage of large value inconsistencies at the beginning of the game, while capitalizing on small potential utility gains at the end.

Allocation. ATTac relies heavily on computing the current G^* . Since G^* changes as prices change, ATTac needs to recompute it at every bidding opportunity. ATTac used a mixed-integer linear programming approach to compute optimal final allocations in every game of the tournament finals—one of only two entrants to do so (see Table 1).

Using a mixed-integer linear programming approach, ATTac specifies the desired output: a list of new goods to purchase and an allocation of new and owned goods to clients to maximize utility minus cost. ATTac searches for optimal solutions to the defined linear program using "branch and bound" search. This approach will find the optimal allocation, usually in under one second on a 600-MHz Pentium computer.

Online adaptation. TAC appealed to ATTac's developers because it appeared to be a good application for machine-learning techniques, one of the developers' main research interests.³ However, TAC was conducted in such a way that it was impossible to determine competitors' bids; only the current ask prices were accessible. This precluded learning detailed models of opponent strategies. ATTac instead adapts its behavior online in three ways:

- ATTac decides when to switch from passive to active bidding mode based on the observed server latency during the current game instance.
- ATTac adapts its allocation strategy based on the amount of time the linear program takes to determine optimal allocations in the current game instance.
- Perhaps most significantly, ATTac adapts its risk-management strategy to account for potentially skyrocketing hotel prices.

ATTac's "branch and bound" search usually found the optimal allocation in under one second.

For flights, ATTac computed G^* based on current prices. For hotels, however, it predicted current game closing prices based on closing prices in previous games. ATTac divided the eight hotel rooms into four equivalence classes, exploiting symme-

```

(A) REPEAT
  1. Update current prices and holdings
  2. Estimate clearing prices and build
    pricelines
  3. Run completer to find optimal buy/sell
    quantities
  4. Set bid/ask prices strategically
  UNTIL game over
(B) Run optimal allocator

```

Figure 2. RoxyBot's high-level strategy. Roxybot's inner bidding loop refines the generic loop shown in Figure 1.

tries in the game (hotel rooms on days one and four should be equally in demand as rooms on days two and three), assigned priors to the expected closing prices of these rooms, and then adjusted these priors based on closing prices observed during the tournament.

Whenever the actual price for a hotel was less than the predicted closing price, ATTac used the predicted hotel closing price for computing its allocation values. This strategy works well both when hotel prices escalate and when they do not.² Indeed, ATTac performed as well as the other top-finishing teams in the early TAC games when hotel prices stayed low, and then outperformed its competitors in the tournament's final games when hotel prices rose to high levels.

RoxyBot: An Approximately Optimal Agent

RoxyBot's algorithmic core, based on AI heuristic search techniques, incorporates an approximately optimal solver for completion and an optimal solver for allocation. The formulation of the completion problem involves a novel data structure called a *priceline*, which is designed to handle future closing prices, future supply and demand, sunk costs, hedging, and arbitrage in a unified way. RoxyBot's high-level strategy is outlined in Figure 2; full details are available in Boyan and Greenwald.⁴

Allocation. RoxyBot's *allocator*, which runs at the end of a game, helps motivate the completer algorithm used during each bidding cycle. The allocator solves the following problem: Given a set of travel resources purchased at auction, and given the clients' utility functions defined over subsets of travel resources, how can the resources be allocated to the clients so as to maximize the sum of their respective utilities? Although this problem is NP-complete,⁵ an optimal solution based on A^* search is tractable for the dimensions of TAC. Indeed, using an intricate series of admissible heuristics,

RoxyBot pruned the search tree of possible optimal allocations from roughly 10^{20} to 10^3 possibilities. As a result, it typically discovered provably optimal allocations in one half of a second in all of the competition games.

The A^* search traverses a tree of depth 16. Search begins at the top of the tree with the given collection of resources. At each level of the tree, a subset of the remaining resources is allocated to a client, and those resources are subtracted from the pool. Levels 1 through 8 correspond to the choice of feasible travel package (that is, combination of flights and hotel rooms) to assign to clients 1 through 8, respectively. There are 21 such travel packages, including the null package. Levels 9 through 16 of the tree correspond to the choice of entertainment package (that is, sets of entertainment tickets of different types on different days) to assign to clients 1 through 8.

There are 73 entertainment packages, though many are infeasible due to earlier assignments of travel packages. The heuristics compute an upper bound on a quantity (for example, the maximum possible number of feasible packages using good hotels, or arriving on day three). Then, subject to these upper bounds, all as-yet-unassigned clients are assigned their preferred package among those remaining, ignoring conflicts. Caching tricks employed at the start of each game enable these heuristics to be computed very quickly.

Completion. The *completer* that runs during each bidding cycle is the heart of RoxyBot's strategy. It aims to determine the optimal quantity of each resource to buy and sell, given current holdings and forecasted closing prices. Like the allocator, it considers all travel resources from a global perspective and makes integrated decisions about hotel, flight, and entertainment bids. Unlike the allocator, the completer faces the added complexity that the resources being assigned may not yet be in hand, but may still need to be purchased at auction. Furthermore, it might be more profitable to sell in-hand entertainment tickets than allocate them to its own clients.

RoxyBot's completer relies on its pricelines to reason about the trade-offs involved with each resource. The priceline transparently handles either one-sided or double-sided auctions, short-selling of resources, hedging, and both limited and unlimited supply and demand. This construction greatly simplifies the completer's task because the cost of a package equals the totals of the corresponding pricelines, and the value of a package to a client

equals the client's utility for that package minus its cost. Given the pricelines and the corresponding client valuations of packages, A^* search can be used to find the optimal set of buying and selling decisions. Unfortunately, most of the A^* heuristics used in RoxyBot's optimal allocator were not applicable in the completer scenario, and running times for an optimal completer occasionally took as long as 10 seconds. Nonetheless, using a greedy, nonadmissible heuristic and a variable-width beam search over the same search space, in practice RoxyBot usually found an optimal completion within about three seconds. Therefore, during the competition, RoxyBot used beam search rather than provably optimal A^* search.

Estimation. RoxyBot's priceline data structures describe the costs of market resources. However, in auctions such as those fundamental to the TAC setup, costs are not known in advance. Therefore, the actual input to RoxyBot's pricelines are but estimates of auction closing prices and estimates of market supply and demand (current holdings are known), which RoxyBot produces using machine-learning techniques. However, the final round of the TAC competition was too short and the agent strategies too different from the preliminary rounds for Roxybot to effectively use most of the learning algorithms that were developed. Only entertainment ticket price estimates were adaptively set, using an adjustment process based on Widrow-Hoff updating.⁶ In future competitions, RoxyBot's creators hope that TAC will be more suited to the use of learning algorithms for price-estimation based on bidding patterns observed during a game instance and an agent's own clients' preferences.

Aster: Flexible Cost Estimation

Aster, the third-placing agent, was designed by members of the Strategic Technologies and Architectural Research Laboratory at InterTrust Technologies. Aster's cost estimation framework is flexible and can respond to strategic behavior of competing agents. Aster's allocation heuristics are relatively simple and fast, and they produce high-quality solutions.

Like RoxyBot, Aster runs a loop. During each iteration, Aster gets the status of all auctions, estimates the costs of resources, computes a tentative allocation based on estimated costs, and bids for some desired resources. After all auctions close, Aster runs a sophisticated algorithm to compute the final allocation.

Bidding. Aster bids using one of two strategies that correspond to game stages before and after the witching hour. Like ATTac, Aster initiates its pre-commit bidding stage as late as feasible, based on previous delay in accessing the AuctionBot, with the hope that it will be able to complete at least one iteration during the active stage. During the pre-commit stage, Aster does not bid on flights. In committed stage, Aster places all necessary flight bids to achieve the current allocation.

In the precommit stage, Aster places limited bids on hotel rooms, trying to capture early closings. At the same time, it tries to avoid engaging in price hikes by placing bids at the minimum allowable increment (that is, the ask price plus \$1). Aster limits its bids for each client to at most two consecutive nights, even if the allocator has scheduled the client for a longer stay. (With at most two nights, if the hotel price on one night shoots up, Aster can drop the expensive night without loss. If it bid for more than two nights and the price on a middle night were to shoot up, it could get stuck with a room or two for the outer nights.) During the committed stage, Aster bids for every night of each client's allocated stay. The amount of these bids equals the utility due to that client.

Aster's strategy for buying and selling entertainment tickets is independent of game stage. It sets the bid and ask prices for tickets by using their utility in the current allocation as well as precomputed expected utilities for other trading agents. Notably, Aster's goal is to obtain greater utility than the other agents, not to maximize its own utility. In some games, Aster profited by buying and selling the same entertainment ticket.

Allocation. On each iteration, Aster computes a tentative allocation of resources by using a local search algorithm that considers pairs of clients in turn, given estimated costs and current holdings. It starts with all clients having no resources and then iterates over all pairs of clients, deallocating their current resources and allocating new resources to maximize utility. This procedure uses the cost vectors as stacks: Deallocating a resource frees up the cost of the last allocated copy, and allocating a resource incurs the cost of an additional copy. Repeated iterations are conducted until utility does not improve.

Aster's cost estimation framework is flexible and can respond to strategic behavior of competing agents.

Aster uses heuristic search to complete its final allocation. To compute the globally optimal allocation of its travel goods to its clients, it searches a tree consisting of all possible travel packages (that is, arrival dates, departure dates, and hotel types) for all clients. Then, at each leaf of this tree, Aster computes an entertainment assignment by iterating over all pairs of clients, deallocating and reallocating entertainment tickets optimally, until the entertainment allocation cannot be improved.

The above search algorithm is not optimal because the entertainment ticket assignment process is only locally optimal, but is not optimal over all clients viewed from the global perspective. Thus, after this first search, Aster searches again, in an attempt to compute an optimal entertainment allocation over all clients while keeping their travel packages fixed. The allocation heuristic performs well, usually finding an optimal or a near-optimal solution (see Table 1).

Aster uses pruning in both searches to cut execution time. Although not provably optimal, Aster's designers believe that such approximate approaches will scale better to larger games than exact approaches, since the size of the search tree can be explicitly controlled.

Estimation. Just as RoxyBot computes a priceline for each resource, Aster computes a cost vector, whose i th entry gives the cost of holding or acquiring the i th copy of that resource. Also like RoxyBot, when estimating costs, Aster treats sunk costs as no costs. For example, the estimated cost for flights is zero for currently held tickets and the current ask price for additional tickets.

For hotels, estimating costs is tricky because both price and holdings are unknown until an auction closes. Aster predicts the closing price for a hotel room by linearly extrapolating previous ask prices on the basis of current time. This extrapolated price is then adjusted as follows: For rooms Aster has hypothetically won, the cost is reduced; the amount of this reduction depends on the probability that these winnings would be ultimately realized (the higher the bid, the higher the probability, and the lower the estimated cost). For additional rooms, the cost is increased exponentially to model potential increases in closing prices due to Aster's own bids.

Since the AuctionBot provides only one bid and

ask quote per entertainment ticket, Aster assumes that the cost of buying an additional ticket is the current ask price and that the cost of further tickets is infinite. For all tickets that Aster currently holds, the opportunity cost of one ticket is set to the current bid price, while the opportunity cost of all the remaining tickets is set to zero.

UmbcTAC:

Network Sensitivity and Adaptability

UmbcTAC, created at University of Maryland Baltimore County, placed fourth. UmbcTAC was particularly sensitive to network load: It adapted to network performance more frequently than competing agents and received more frequent updates.

Bidding. UmbcTAC maintains the most profitable itinerary for each client individually, based on the latest price quotes (as opposed to solving the full eight-client optimization problem). UmbcTAC balances several strategies to avoid switching travel plans too frequently against some strategies to encourage switching travel plans early on:

- When a client's itinerary is changed, the value of the goods that will no longer be needed is subtracted from the value of the new itinerary as a penalty for changing plans. Therefore, the client's travel plans will not change unless the new plan's profit overrides the value of wasted goods. Wasted goods that are already won, or would be won if an auction closed immediately, are marked as free goods, their prices are set to 0, and they are treated as sunk costs. Free goods may then be used in other clients' itineraries.
- UmbcTAC changes a client's travel plan only if the profit difference between the new and the old plans exceeds a threshold value (typically between \$10 and \$100).
- It is important to change a client's itinerary as early in the game as possible because the earlier the plan changes, the more likely it is that the obsolete bids either will not win or will win at a low price. To ensure that at least one client changes to a better plan, UmbcTAC risks wasting one good in each round by setting the penalty for the first wasted good to 0.

Once the desired goods have been determined, UmbcTAC sets its bid prices as follows:

- *Flights.* The agent bids a price significantly higher than the current price to ensure that the client gets the ticket.

UmbcTAC bases its bidding strategy on the computed network delay between the agent and the TAC server.

- **Hotels.** The agent computes the price increment, defined as the difference between the current price quote and the previous price quote. It sets the bid price to be the current price plus the price increment. During the bidding hour, UmbcTAC bids for hotels at a price such that if it wins a hotel at that price, the client's utility would be 0.
- **Entertainment.** The agent buys entertainment tickets for a client if the client is available (that is, in town and without an entertainment ticket for that night or of that type). It buys the ticket that the client most prefers at the market value. Any extra tickets are sold at auction at an ask price equal to the average of the preference values of all UmbcTAC's clients.

UmbcTAC continually bids for hotels to guard against the possibility of hotel auctions closing early. It bids for airline tickets and raises hotel bids to their limit only in the last few seconds of the game.

Allocation. At the end of the game, UmbcTAC allocates the purchased flights and hotel rooms greedily to the clients according to the most recent travel plans used during the game. If a client cannot be satisfied, its goods are taken back and marked as free goods, which other clients can then try to use to improve utility. Entertainment tickets are also allocated greedily. This strategy is simple and nearly optimal. UmbcTAC begins by allocating an entertainment ticket to the available client with the largest preference value for that ticket.

Bandwidth management. In TAC, prices change every second, and hotel auctions may close at any time. Therefore, keeping bidding data up to date is very important. UmbcTAC bases its bidding strategy on the computed network delay between the agent and the TAC server. When the network delays are longer than usual, UmbcTAC is more aggressive, offering higher prices and bidding for flights earlier. To save network bandwidth, the agent never bids for any entertainment tickets during the last three minutes of the game. To save time, the agent does not change travel plans during its last two or three bidding opportunities. On average, UmbcTAC updates its bidding data every four to six seconds, providing a significant advantage over the 8- to 20-second delays reported by others.

Suggestions for Future Competitions

The first trading agent competition drew 22 entrants from around the world. Although partici-

pants' experiences were overwhelmingly positive, we propose a few modifications to the structure of future tournaments.

- There is no incentive to buy airline tickets until the end of the game. If the price of flights tended to increase, or if availability was limited, agents would have to balance the advantage of keeping their options open against the savings of committing to itineraries earlier.
- The hotel auctions were effectively reduced to sealed-bid auctions. In particular, there was no incentive for agents to reveal their preferences before the very end of the game. As a result, it was impossible for agents to model market supply and demand and thereby estimate prices.

The phenomenon of English auctions with set closing times reducing to sealed-bid auctions has been observed in other online auction houses such as eBay. Roth and Ockenfels argue that in such auctions, it is in fact an equilibrium strategy to place multiple bids (with increasing valuations) and to bid at the last possible moment.⁷ This observation contradicts the usual intuition pertaining to second-price sealed-bid auctions, namely that a single bid at one's true valuation is a dominant strategy.

The information structure of the TAC setup made it impossible to observe the bidding patterns of individual agents.

Amazon runs online auctions in which the length of the auction is extended beyond its original closing time, say T , by 10 minutes each time a new (winning) bid is received. In this case, equilibrium behavior dictates that all bidders bid their true valuations before time T . If the TAC hotel auctions were implemented in the style of Amazon, rather than eBay, agents would likely bid earlier. In this way, TAC agents would obtain more information pertaining to the specific market supply and demand induced by the random client preferences realized in each game instance, and could use this information to estimate hotel prices. Unfortunately, Amazon-style auctions have the downside that they might never end!

- Activity in the entertainment auctions was limited. This outcome, however, is not obviously correlated with the design of the entertainment auction mechanism. On the contrary, if more

- structure were added to the flight auctions, and if the hotel auctions were modified, interest in entertainment ticket auctions might increase.
- The information structure of the TAC setup made it impossible to observe the bidding patterns of individual agents. Nonetheless, the strategic behavior of individual agents often profoundly affected market dynamics, particularly in the hotel auctions. It seems that either the dimensions of the game should be extended such that the impact of any individual agent's bidding patterns is truly negligible; or, to avoid issues of scalability, it should be possible to directly model the effect of the behavior of each individual agent. If information were available about the bidding behavior of the agents (such that other agents could induce clients' preferences, and therefore market supply, demand, and prices), TAC agents might learn to predict market behavior as a game proceeds.

The agents developed for TAC are a first step toward creating autonomous bidding agents for real, simultaneous interacting auctions. One such auction is the Federal Communications Commission's auction of radio spectrum.^{8,9} For companies that are trying to achieve national radio coverage, the values of the different licenses interact in complex ways. Perhaps autonomous bidding agents will impact bidding strategies in future auctions of this type. Indeed, the ATTac developers created straightforward bidding agents in a realistic FCC Auction Simulator.¹⁰ In a more obvious application, an extended version of TAC agents could become a useful tool to travel agents, or to end users who wish to create their own travel packages. □

Acknowledgments

This article is the result of the efforts of many people, including all of the TAC finalists. The authors are particularly indebted to Justin Boyan (RoxyBot), Umesh Maheshwari (Aster), and Youyong Zou (UmbcTAC) for their contributions to their respective sections.

References

1. TAC Team, "A Trading Agent Competition," *IEEE Internet Computing*, vol. 5, no. 2, Mar./Apr. 2001, pp. 43-51.
2. P. Stone et al., "Attac-2000: An Adaptive Autonomous Bidding Agent," to be published in *Proc. Fifth Int'l Conf. Autonomous Agents*, 2001.
3. P. Stone, *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*, MIT Press, Cambridge, Mass., 2000.
4. J. Boyan and A. Greenwald, "RoxyBot: A Dynamic Bidding Agent for Simultaneous Auctions," available at <http://www.cs.brown.edu/people/amgreen/> (current Mar. 2001).
5. A. Roth and A. Ockenfels, "Late Minute Bidding and the Rules for Ending Second-Price Auctions: Theory and Evidence from a Natural Experiment on the Internet," working paper, Harvard University, Cambridge, Mass., 2000.
6. D. Cliff and J. Bruton, *Zero is Not Enough: On the Lower Limit of Agent Intelligence for Continuous Double Auction Markets*, tech. report HPL-97-141, Hewlett-Packard, Bristol, UK, 1997.
7. M.H. Rothkopf, A. Pekefic, and R.M. Harstad, "Computationally Manageable Combinatorial Auctions," *Management Science*, vol. 44, no. 8, 1998, pp. 1131-1147.
8. R.J. Weber, "Making More from Less: Strategic Demand Reduction in the FCC Spectrum Auctions," 1996, available online at http://www.kellogg.nwu.edu/faculty/weber/PAPERS/pcs_auc.htm (current Mar. 2001).
9. P.C. Cramton, "The FCC Spectrum Auctions: An Early Assessment," *J. Economics and Management Strategy*, vol. 6, no. 3, 1997, pp. 431-495.
10. J.A. Csirik et al., "FAucs: An FCC Spectrum Auction Simulator for Autonomous Bidding Agents," submitted to *Proc. 17th Int'l Conf. Artificial Intelligence*, 2001; available at <http://www.research.att.com/~pstone/papers.html>.

Amy Greenwald is an assistant professor of computer science at Brown University in Providence, Rhode Island. Her primary area of interest is multiagent learning on the Internet, which she approaches using game-theoretic models of computational interactions. She completed her PhD at the Courant Institute of Mathematical Sciences of New York University, where she received the Janet Fabri Memorial Prize for a doctoral dissertation of exceptional quality.

Peter Stone is a senior technical staff member in the Artificial Intelligence Principles Research Department at AT&T Labs Research where he investigates multiagent learning. He received a PhD in 1998 and an MS in 1995 from Carnegie Mellon University, both in computer science. He received his BS in mathematics from the University of Chicago in 1993. Stone's research interests include planning and machine learning, particularly in multiagent systems. Stone is the author of *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer* (MIT Press, 2000) and was awarded the Allen Newell Medal for Excellence in Research in 1997.

Readers can contact Greenwald at the Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, amgreen@cs.brown.edu; or Stone at AT&T Labs-Research, 180 Park Ave., Room A273, Florham Park, NJ 07932, pstone@research.att.com.

Bidding under Uncertainty: Theory and Experiments

Amy Greenwald
Department of Computer Science
Brown University, Box 1910
Providence, RI 02912
amy@brown.edu

Justin Boyan
ITA Software
141 Portland Street
Cambridge, MA 02139
jab@itasoftware.com

Abstract

This paper describes a study of agent bidding strategies, assuming combinatorial valuations for complementary and substitutable goods, in three auction environments: sequential auctions, simultaneous auctions, and the Trading Agent Competition (TAC) Classic hotel auction design, a hybrid of sequential and simultaneous auctions. The problem of bidding in sequential auctions is formulated as an MDP, and it is argued that expected marginal utility bidding is the optimal bidding policy. The problem of bidding in simultaneous auctions is formulated as a stochastic program, and it is shown by example that marginal utility bidding is not an optimal bidding policy, even in deterministic settings. Two alternative methods of approximating a solution to this stochastic program are presented: the first method, which relies on expected values, is optimal in deterministic environments; the second method, which samples the nondeterministic environment, is asymptotically optimal as the number of samples tends to infinity. Finally, experiments with these various bidding policies are described in the TAC Classic setting.

1 Introduction

One of the key challenges autonomous bidding agents face is to determine how to bid on complementary and substitutable goods—i.e., goods with combinatorial valuations—in auction environments. Complementary goods are goods with superadditive valuations: $v(A\bar{B}) + v(\bar{A}B) \leq v(AB)$; substitutable goods are goods with subadditive valuations: $v(A\bar{B}) + v(\bar{A}B) \geq v(AB)$. In general, it is impossible to assign independent valuations to complementary goods, which can be

worthless in isolation, or to substitutable goods, which can be worthwhile only in isolation. Thus, the simple bidding strategy “for each good x , bid its valuation” is inapplicable in this framework. This paper investigates a class of bidding strategies for various auction environments, assuming combinatorial valuations for complementary and substitutable goods.

Specifically, we consider three auction environments: sequential auctions, simultaneous auctions, and the hybrid of sequential and simultaneous auctions implemented in TAC Classic.¹ As the name suggests, in sequential auctions, goods are sold sequentially, in some fixed, known order. Here, agents can reason about each good in turn, basing future decisions on past outcomes. But in simultaneous auctions all goods are sold simultaneously. Here, agents must reason about all goods simultaneously, with only one opportunity to make one bidding decision that pertains to all goods. In the TAC Classic (hotel) auction design [10], auctions close sequentially, but in some random, unknown order. Here, before each auction closes, agents must reason about all goods in as yet open auctions simultaneously, but after each auction closes, agents can base future decisions on past outcomes.

Rather than attempt to reason about the valuations of goods independently, bidding agents that operate in these auction environments can reason about *marginal* valuations, or the valuation of a good x relative to a set of goods X . In particular, if an agent holds the goods in X , it can ask questions such as: “what is the marginal benefit of buying x ?” or “what is the marginal cost of selling x ?” In doing so, the agent reasons about the *set* of goods $X \cup \{x\}$ or $X \setminus \{x\}$, relative to the set X —the valuations of which are well-defined. In Section 2 of this paper, we prove that the simple bidding strategy “for each good x , bid its (average) *marginal utility*” is optimal in sequential auctions. But not so in simultaneous auctions, as the following example shows.

¹Visit <http://www.sics.se/tac> for details.

Example 1.1 Consider a set of $N > 1$ goods that are being auctioned off simultaneously. Assume the value of one or more of these goods is 2, while the auction price of each good is 1, deterministically.² In this setting, bidding marginal utilities amounts to bidding 1 on each good. In doing so, this strategy obtains utility $2 - N < 1$. In contrast, any strategy that bids 1 on exactly one good obtains utility $2 - 1 = 1$. Thus, bidding marginal utilities is suboptimal. \square

Beyond seeking an optimal bidding policy for TAC Classic hotel auctions, the goal of this research is to derive optimal solutions to the problems of bidding on goods with combinatorial valuations in sequential and simultaneous auctions. The difficulty that an agent faces in optimizing its behavior in these problems is due to the uncertainty that arises from its lack of information about the other agents' bidding strategies. In this paper, we make the simplifying assumption that the price of each good is given by an exogenous probability distribution, which is determined by the collective behavior of all competing agents, but which ignores the behavior of the optimizing agent. Consequently, even if the optimizing agent places a winning bid, the price of the good does not depend on this bid.

The bidding strategies analyzed in this study, which were inspired by agent strategies in the international Trading Agent Competition [10] (TAC Classic), are all based on the notion of marginal utility. Specifically, we study expected marginal utility bidding, implemented in ATTAC-01 [9], and variants of policy search, implemented in RoxyBOT-00 [7] and RoxyBOT-02 [6]. Note that these teams were two of the few to exploit stochastic price information in their agent design, beyond computing straightforward expected values [11]. Although marginal utility bidding is not optimal in simultaneous auctions, we prove that it is the optimal bidding policy in sequential auctions, and we show empirically that it is a reasonable heuristic for bidding in TAC Classic hotel auctions.

2 Sequential Auctions

In this section, we formulate the bidding problem in sequential auctions—a sequential decision problem—as a Markov decision process (MDP), and we compute the optimal policy. As an example, consider the MDP depicted in Figure 1, which represents an agent's bidding problem in two sequential auctions for two goods A and B . There are seven states in this MDP, drawn in three stages. In the first stage, a bidding decision is made regarding good A , and in the second stage,

²Think of the BUY IT Now option at eBay.com, which serves as an example of deterministic auction prices.

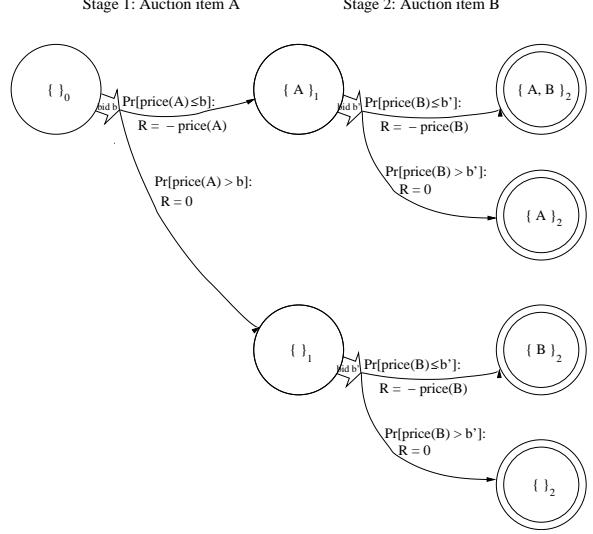


Figure 1: The sequential auction problem as an MDP: An example with two goods A and B .

a bidding decision is made regarding good B . Along the way, the agent earns negative rewards equal to the price of each good for which it places a winning bid. A winning bid for the j th good is a bid for an amount that is greater than or equal to the (uncertain) price of that good. Transitions are stochastic: *e.g.*, given bid b' , the system transitions from state $(\{A\}, 1)$ to state $(\{A, B\}, 2)$ with probability equal to the probability that b' is a winning bid; otherwise the system transitions to state $(\{A\}, 2)$. In the final stage, the agent earns rewards equal to the valuation of the subset of the set $\{A, B\}$ it successfully acquires.

In general, a finite set of n goods $N = \{x_1, \dots, x_n\}$ is given. Let $J \subseteq N = \{x_1, \dots, x_j\}$. Now a state at stage j is denoted (X, j) , where $X \subseteq J$ is the set of current holdings and good x_{j+1} is up for auction. The actions available at each state are real-valued bids. (Note that the action space is continuous.) A bid b for good x_j at state $(X, j-1)$ is declared to be a winning bid if the good's randomly sampled price $p_j \leq b$. All other bids are losing bids. Transitions, which depend on state-action pairs, are stochastic. Given bid b , a transition is made from state $(X, j-1)$ to state $(X \cup \{x_j\}, j)$ with probability equal to the probability that b is a winning bid; otherwise a transition is made from $(X, j-1)$ to (X, j) . Rewards are associated with state-action-state triples at all stages $0 \leq j \leq n-1$ as follows: a winning bid at stage j earns reward $-p_j$, where p_j is the price of the j th good; a losing bid earns no rewards. In addition, a valuation function $v : 2^N \rightarrow \mathbb{R}$, which assigns a valuation to each subset of N , defines the rewards at stage n , at which point the auction ends.

Definition 2.1 The bidding problem in sequential auctions is defined by the following MDP:

- States: each state at stage j , for $0 \leq j \leq n$, is denoted (X, j) , where $X \subseteq J$.
- Actions: the actions available at all states $(X, j - 1)$ at stage $j - 1$ are bids $b_j \in \mathbb{R}^+$ on good x_j .
- Transitions: $P((X \cup \{x_j\}, j) | (X, j - 1), b) = F_j(b)$, and $P((X, j) | (X, j - 1), b) = 1 - F_j(b)$, where $F_j(x) = \Pr[p_j \leq x]$ for good j : i.e., F_j is the cdf over prices p_j .
- Rewards:
 - For all states (X, n) with $X \subseteq 2^N$ at stage n , $R((X, n)) = v(X)$.
 - For all other states (X, j) with $X \subseteq J$ at stage j , if b is a winning bid for good j , then reward $-p_j$ is incurred; otherwise, no cost is incurred. Define the following:

$$r((X, j - 1), b, p) = \begin{cases} -p & \text{if } p \leq b \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Now

$$R((X, j - 1), b) = \int_{-\infty}^{\infty} r((X, j - 1), b, p) f_j(p) dp \quad (2)$$

where $f_j = F'_j$ is the pdf over prices p_j .

The optimal policy in this MDP is described by Bellman's equations [1]:

$$\pi((X, j)) \in \arg \max_b Q((X, j), b) \quad (3)$$

$$\begin{aligned} Q((X, j), b) &= R((X, j), b) + \\ F_j(b)V((X \cup \{x_{j+1}\}), j+1) + (1 - F_j(b))V((X, j+1)) \\ V((X, j)) &= \max_b Q((X, j), b) \end{aligned} \quad (4) \quad (5)$$

Theorem 2.2 *The following bidding policy is optimal in this MDP: at state $((X, j - 1))$, place bid $b_j^* = V((X \cup \{x_j\}), j) - V((X, j))$.*

Proof 2.2 For arbitrary p_j , two cases arise: Case $b_j^* \geq p_j$: If $b \geq p_j$, then $Q((X, j - 1), b) = V(((X \cup \{x_j\}), j)) - p_j = Q((X, j - 1), b_j^*)$. If, however, $b < p_j$, then $Q((X, j - 1), b) = V((X, j)) \leq V(((X \cup \{x_j\}), j)) - p_j = Q((X, j - 1), b_j^*)$, since $b_j^* \geq p_j$, by assumption. The case in which $b_j^* < p_j$ is symmetric. \square

This MDP formulation is related to that of Boutilier, *et al.* [4]. The most notable difference is in the two state representations. In their model, states consist of

current holdings together with an endowment, which decreases as goods are acquired. In our model, rather than decrease an endowment, expenses are modeled as negative rewards. Our representation leads to a notable savings in the size of the state space, but cannot model an agent with a finite budget. Note that Theorem 2.2 is not applicable in their model.

2.1 Marginal Utility

In this section, we present an interpretation of the optimal bidding policy in sequential auctions in terms of marginal utilities. Computing marginal utilities depends on solving the so-called *acquisition problem* [5]: “Given the set of goods that I already own, and given market prices and supply, on what set of additional goods should I place bids so as to maximize my utility: i.e., valuation less costs?”

Definition 2.3 Given a set of goods X that an agent already owns, a set of goods Y supplied by the market s.t. $X \cap Y = \{\}$, a vector of prices \vec{p} with $p_k \in \mathbb{R}_+$ for all $k \in Y$, and a combinatorial valuation function $v : 2^{X \cup Y} \rightarrow \mathbb{R}$ satisfying free disposal (i.e., if $Y \subseteq X$, $v(Y) \leq v(X)$),

- the *acquisition* function $\alpha(X, Y, \vec{p})$ is defined as follows:

$$\alpha(X, Y, \vec{p}) = \max_{Z \subseteq X \cup Y} \left(v(Z) - \sum_{k \in Z \cap Y} p_k \right) \quad (6)$$

- the *marginal utility* of good $x \notin X \cup Y$ is defined as follows: $\mu(x, X, Y, \vec{p}) = \alpha(X \cup \{x\}, Y, \vec{p}) - \alpha(X, Y, \vec{p})$.

In words, the marginal utility of good $x \notin X \cup Y$ is the difference between the utility of $X \cup \{x\} \cup Y$, assuming x costs 0, and the utility of $X \cup Y$ (equivalently, the utility of $X \cup \{x\} \cup Y$, assuming x costs ∞).

Definition 2.4 Given a set of goods X that an agent already owns, a set of goods Y supplied by the market s.t. $X \cap Y = \{\}$, a joint probability density function f describing the prices of the goods in Y , and a combinatorial valuation function $v : 2^{X \cup Y} \rightarrow \mathbb{R}$,

- the *expected acquisition* function $\bar{\alpha}(X, Y)$ is defined as follows:

$$\bar{\alpha}(X, Y) = \mathbb{E}_{\vec{p}}[\alpha(X, Y, \vec{p})] = \int_{\vec{p}} \alpha(X, Y, \vec{p}) f(\vec{p}) d\vec{p} \quad (7)$$

- the *expected marginal utility* of good $x \notin X \cup Y$ is given by: $\bar{\mu}(x, X, Y) = \bar{\alpha}(X \cup \{x\}, Y) - \bar{\alpha}(X, Y)$.

Thus, the expected marginal utility at state $(X, j - 1)$ is the difference between the expected value of the solution to acquisition assuming x_j costs 0, and the expected value of this solution assuming x_j costs ∞ .

The proof of the following (key) lemma is omitted.

Lemma 2.5 *For all states (X, j) in the MDP, $V((X, j)) = \bar{\alpha}(X, Y_j)$, where $Y_j = \{x_{j+1}, \dots, x_n\}$.*

Corollary 2.6 *For all states (X, j) in the MDP, and for all goods x_j , $\bar{\mu}(x_j, X, Y_j) = V((X \cup \{x_j\}, j)) - V((X, j))$, where $Y_j = \{x_{j+1}, \dots, x_n\}$.*

Proof 2.7 The proof follows immediately from Lemma 2.5:

$$\begin{aligned}\bar{\mu}(x_j, X, Y_j) &= \bar{\alpha}(X \cup \{x_j\}, Y_j) - \bar{\alpha}(X, Y_j) \\ &= V((X \cup \{x_j\}, j)) - V((X, j))\end{aligned}$$

Corollary 2.8 *Expected marginal utility bidding is the optimal bidding policy in sequential auctions.*

Proof 2.8 The proof follows from Theorem 2.2 together with Corollary 2.6. \square

Returning to the example in Figure 1, the optimal policy at state $(\{\}, 0)$ is simply to bid the expected marginal utility of good A , which is the difference between the value of state $(\{A\}, 1)$ and state $(\{\}, 1)$. Similarly, the optimal policy at state $(\{A\}, 1)$ is to bid the expected marginal utility of good B given that the agent already owns good A , which is the difference between the value of states $(\{A, B\}, 2)$ and $(\{A\}, 2)$; and, the optimal policy at state $(\{\}, 1)$ is to bid the expected marginal utility of good B given that the agent owns nothing as of yet, which is the difference between the values of states $(\{B\}, 2)$ and $(\{\}, 2)$. In what follows, we abbreviate expected marginal utility bidding by \bar{MU} , and we abbreviate marginal utility bidding (assuming deterministic prices) by MU.

3 Simultaneous Auctions

In simultaneous auctions, an agent must make all its bidding decisions *a priori*, with only probabilistic knowledge of what it may or may not win.

For example, suppose that a camera and a flash are being auctioned off simultaneously, and that an agent assigns valuation \$750 to these two goods together, but assigns valuation \$0 to either good alone. In addition, suppose that the price of the flash is known with certainty: it is \$50; but the price of camera will be \$500 with probability $\frac{1}{2}$ and \$1000 with probability $\frac{1}{2}$. Given these assumptions, what is an optimal bidding policy? If it happens that the camera sells

for \$500, then it is optimal to bid \$500 for the camera and \$50 for the flash. But, if it happens that the camera sells for \$1000, then it is optimal to bid \$0 for both the camera and the flash. Evaluating these two policies, bidding (\$0,\$0) yields \$0 utility, while bidding (\$500,\$50) yields \$200 utility half the time, and $-\$50$ utility half the time. Thus, the expected value of bidding (\$500,\$50) is \$75, and this is the best possible.

In this section, we formulate the problem of bidding in simultaneous auctions as a stochastic program whose solution is an optimal bidding policy in the expected sense. Here, the expectation is computed over all possible stochastic outcomes (*a.k.a.* scenarios): e.g., if there are two goods, this set of scenarios includes win good 1, win good 2, win both goods, win neither good. Since the number of scenarios is exponential in the number of goods, computing an optimal solution to this stochastic program is intractable for large numbers of goods. We discuss three methods for approximating an optimal bidding policy in this environment: one heuristic approach (expected MU bidding), and two approximation schemes (the so-called *expected value method* with MU bidding and a stochastic sampling technique). In the next section, we describe experiments with all three of these strategies in the TAC Classic auction framework.

3.1 Problem Statement

Given valuation function $v : 2^X \rightarrow \mathbb{R}$, let \vec{v} denote the vector of valuations, with v_i as the valuation of the i th subset of X . As in Section 2.1, good prices are described by the joint probability function f . We seek a set of bids that maximizes total expected utility: i.e., expected valuation less expected cost. That is, we seek a set of bids *today*, before any uncertainty is resolved, that maximizes our expected utility *tomorrow*, after all uncertainty is resolved. Before tackling the bidding problem, we first describe the *allocation* problem, which arises after all uncertainty is resolved, since it is at this point that all winnings are allocated to bundles.

Let \vec{p} denote the vector of prices with $p_{jk} \in \mathbb{R}_+$ as the price of the k th copy of good j . Let the continuous decision variables $b_{jk} \in \mathbb{R}_+$ denote the bid placed on the k th copy of good j ; let the binary decision variables $a_{ijk} \in \{0, 1\}$ indicate whether or not the k th copy of good j is allocated to bundle i . Define the following:

$$\begin{aligned}\pi(\vec{a}, \vec{b}, \vec{p}, \vec{v}) &= - \sum_{jk} p_{jk} (\mathbf{1}[p_{jk} \leq b_{jk}]) \\ &+ \sum_i v_i \left(\prod_{j \in i} \mathbf{1} \left[n_{ij} \leq \sum_k a_{ijk} \mathbf{1}[p_{jk} \leq b_{jk}] \right] \right)\end{aligned}\quad (8)$$

where n_{ij} denotes the number of copies of good j that

are essential to bundle i , and $\mathbf{1}[x \leq y]$ is an indicator function that evaluates to 1 if $x \leq y$ and otherwise evaluates to 0. According to π , goods for which winning bids are placed incur costs, but are also allocated to bundles that secure valuations, as long as enough copies of all goods that are essential to a bundle are indeed allocated to that bundle. The *allocation* problem can be described by the following integer program:

$$\max_{\vec{a}} \pi(\vec{a}, \vec{b}, \vec{p}, \vec{v}) \quad (9)$$

$$\text{subject to: } \sum_i a_{ijk} \leq 1, \quad \forall j, k \quad (10)$$

$$a_{ijk} \in \{0, 1\}, \quad \forall i, j, k \quad (11)$$

The first set of constraints states that each copy k of each good j can be allocated to at most one bundle.

The following stochastic program [3] solves the bidding problem in simultaneous auctions:

$$\max_{\vec{b}} \int_{\vec{p}} \max_{\vec{a}} \pi(\vec{a}, \vec{b}, \vec{p}, \vec{v}) f(\vec{p}) d\vec{p} \quad (12)$$

$$\text{subject to: } b_{jk} \in \mathbb{R}_+, \quad \forall j, k \quad (13)$$

Notice that the allocation problem is nested inside the bidding problem.

Later, we refer to the deterministic version of the bidding problem as *completion*: given prices \vec{p} ,

$$\max_{\vec{a}, \vec{b}'} \pi'(\vec{a}, \vec{b}', \vec{p}, \vec{v}) \quad (14)$$

$$\text{subject to: } b'_{jk} \in \{0, 1\}, \quad \forall j, k \quad (15)$$

where

$$\begin{aligned} \pi'(\vec{a}, \vec{b}', \vec{p}, \vec{v}) = \\ \sum_i v_i \left(\prod_{j \in i} \mathbf{1} \left[n_{ij} \leq \sum_k a_{ijk} b'_{jk} \right] \right) - \sum_{jk} p_{jk} b'_{jk} \end{aligned} \quad (16)$$

3.2 Heuristics & Approximation Algorithms

In this section, we discuss three methods for approximating an optimal solution to this stochastic program: one heuristic approach inspired by ATTAC-01 (expected MU bidding), and two approximation schemes inspired by ROXYBOT (the so-called *expected value method* with MU bidding and a stochastic sampling technique). First, we show that the heuristic of bidding expected marginal utilities, while optimal in sequential auctions, is suboptimal in simultaneous auctions. Second, we show that the expected value method with MU bidding is also suboptimal, although this approach is optimal when prices are deterministic. Third, we discuss an asymptotically optimal sampling method: i.e., as the number of samples grows, the value of the approximate solution approaches the value of the stochastic programming solution.

3.2.1 ATTAC-01: Expected MU Bidding

In Example 1.1, in the introduction, we argued that marginal utility bidding is suboptimal in simultaneous auctions with deterministic prices. We present a second example here in which prices are nondeterministic and *expected* marginal utility bidding is suboptimal.

Example 3.1 Let $v(x) = v(y) = v(xy) = 1$. Assume the prices of goods x and y are described by the following bipolar distribution: $p(a) = 1$, with probability $\frac{1}{2}$, and $p(a) = 101$, with probability $\frac{1}{2}$, for all $a \in \{x, y\}$. Now expected marginal utility bidding gives rise to the policy “Bid 1 on both goods” in this example, since $\mu(x, \emptyset, \{y\}, \vec{p}) = \mu(y, \emptyset, \{x\}, \vec{p}) = 1$ under all price samples \vec{p} . But then expected marginal utility bidding earns expected utility $-\frac{1}{4}$. The policy “Bid 0 on both goods” (which earns expected utility 0) dominates expected marginal utility bidding in this example.

x	y	$\mu(x)$	$\mu(y)$	Evaluation
1	1	1	1	-1
1	101	1	1	0
101	1	1	1	0
101	101	1	1	0
Average		1	1	$-\frac{1}{4}$

3.2.2 ROXYBOT-00: Expected Value Method with MU Bidding

In this section, we show (i) the expected value method with MU bidding (EVMU) is optimal when prices are deterministic; but, (ii) in general, the expected value method, and therefore EVMU, is suboptimal.

It is common to approximate the solution to stochastic programs using the so-called *expected value method* (see, for example, [3]). This method solves the deterministic version of the problem, assuming all stochastic inputs have deterministic values equal to their expected values. Applying this method to our stochastic program yields a solution to Equation 14: i.e., an optimal set of goods on which to bid. As in the deterministic setting, it is straightforward to transform a solution to this problem into an optimal bidding policy: bid $\$0$, whenever $b_j = 1$; bid $\$0$, whenever $b_j = 0$. But, in general the expected value method is suboptimal.

Recall the discussion of the camera and the flash introduced at the beginning of this section. One attempt to approximate the optimal solution (in the expected sense) is obtained by solving the deterministic variant of the problem: assume the price of the camera is $\$750$ (its expected price), while the price of the flash is $\$50$. Under these assumptions, the cost exceeds the valuation of the camera and the flash; thus, the optimal policy is to bid $(\$0, \$0)$, which yields $\$0$ utility.

In this example, the so-called *value of stochastic information* (i.e., the difference between the solutions to Equation 12 and Equation 14) is \$75.

The following example shows that there is value to stochastic information not only in simultaneous auctions, but even in an auction for only one good (which is both a sequential and a simultaneous auction).

Example 3.2 Consider only one good a of value \$100. Suppose a 's price is \$1 with probability .9, but that a 's price is \$1 million with probability .1. Thus, the expected price of good a is roughly \$100,0010. The optimal policy using the expected value method is to bid \$0, which scores \$0. But now consider the bidding policy “bid \$100.” This policy scores \$99 with probability .9, and \$0 with probability .1. Thus, on average, this policy scores roughly \$89. “Bid \$100” dominates the expected value method in this example. Indeed, “bid \$100,” which corresponds to bidding expected marginal utility, is optimal, since this auction is sequential. \square

In the introduction, we made an important simplifying assumption, namely, the price of each good is given by an exogenous probability distribution, which is determined by the collective behavior of all competing agents, but which ignores the behavior of the optimizing agent. In TAC Classic hotel auctions, for example, this assumption is violated: an agent’s bid *can* impact the prices of goods—a winning agent could even pay what it bids. As a heuristic that is applicable in this more general setting, we propose the following bidding policy: bid marginal utilities on all goods in an optimal set A^* computed using the expected value method. If it is possible that an agent could pay what it bids, then marginal utility seems to be a reasonable upper bound on what it should bid, since bidding marginal utility on some good $x \in A^*$ is indeed optimal if the prices of all other goods $y \neq x \in A^*$ are deterministic. In summary, we propose the following bidding policy: (i) set p_j equal to the mean of P_j , (ii) solve the completion problem (i.e., Equation 14), and (iii) bid marginal utilities on all goods in the optimal completion: i.e., all goods for which $b_j = 1$. We call this strategy EVMU. It was implemented in ROXYBOT-00 [7], and it was shown that *EVMU bidding is optimal in simultaneous auctions when prices are deterministic* in Greenwald [6].

3.2.3 ROXYBOT-02: MU Bidding Policy Search

As a third means of approximating an optimal solution to the problem of bidding in simultaneous auctions, one possible technique called *Sample Average Approximation* (SAA) method solves the stochastic program

using only a subset of the scenarios, randomly sampled according to the scenario distribution. An important theoretical justification for this method is that as the sample size increases, the solution converges to an optimal solution in the expected sense. Indeed, the convergence rate is exponentially fast [8]. In Benisch, *et al.* [2], we apply this technique to the TAC SCM scheduling problem.³

Here we explore an alternative means of approximating an optimal solution to the stochastic program, namely policy search, which in the absence of any clever heuristics, is simply brute-force, generate-and-test. This solution technique generates a set of candidate policies, evaluates them, and selects the best one. Candidates are evaluated over multiple samples: for each sample, the candidate’s score is computed, and scores are averaged over all samples (as in the policy evaluation process in Example 3.1). If it were to evaluate all possible candidate policies under infinitely many samples, this method would tend towards outputting an optimal bidding policy.

A variant of this approach is at the heart of ROXYBOT-02, which generates candidate policies via the following heuristic: (i) determine p_j by sampling from the price distributions; (ii) solve the completion problem (i.e., Equation 14), and (iii) bid marginal utilities on all goods in the optimal completion: i.e., all goods for which $b_j = 1$. More generally, ROXYBOT-02 can generate policies according to any of the aforementioned algorithms: EVMU, MU, or expected MU. By including in the space of candidate policies those generated by these alternative strategies, we can ensure (probabilistically) that ROXYBOT-02 dominates the others.

In TAC-02, ROXYBOT-02 generated its candidate policies using ROXYBOT-00’s internals. In Example 3.2, this instantiation of ROXYBOT-02 generates two policies: if the sample price is \$1, its bidding policy is “bid \$100;” if the sample price is \$1 million, its bidding policy is “bid \$0.” “Bid \$100” scores \$89, on average, whereas “bid \$0” scores \$0. Thus, ROXYBOT-02 employs the policy “bid \$100”, and scores \$89, on average. This form of policy search outperforms EVMU in this example. Indeed, there is value in exploiting stochastic information beyond expected values. In the next section, we demonstrate this effect in TAC Classic.

4 Experiments

Our analysis of agent bidding strategies in the previous two sections was based on the assumption that prices are determined exogenously. In particular, MU

³See www.sics.se/tac for a description of TAC SCM.

is optimal in sequential auctions if prices are deterministic and exogenous; EVMU is optimal in simultaneous auctions if prices are deterministic and exogenous; and, policy search is approximately optimal in simultaneous auctions, even if prices are uncertain, but still exogenous. In this section, we discuss experiments designed to ascertain the power of these strategies in the TAC Classic hotel auctions, a hybrid of sequential and simultaneous auctions, in which prices are endogenous, rather than exogenous.

In TAC Classic hotel auctions, the TAC seller auctions off 16 hotel rooms in ascending, multi-unit, sixteenth price auctions. These auctions close sequentially. The order of the auction closings is unknown to the agents. In fact, MU bidding is optimal even in sequential auctions which close in some random, unknown order *if bids can be withdrawn*. The difficulty in TAC classic hotel auctions is that bids cannot be retracted. Moreover, when agents submit bids, they must “beat the quote,” according to the following rules:

Let a be the sixteenth highest price. Any new bid b must satisfy the following conditions to be admitted to the auction: (i) b must offer to buy at least one unit at a price of $a + 1$ or greater; (ii) if the agent’s current bid c would have resulted in a purchase of q units, then the new bid b must offer to buy at least q units, again at a price of $a + 1$ or greater.

4.1 Setup

In our experiments, we pitted 4 TAC agents bidding according to one strategy against 4 TAC agents bidding according to another strategy (*e.g.*, 4 RoXYBot-02 agents vs. 4 RoXYBot-00 agents). We refer to each set of 4 TAC agents in one game as a team. We played numerous games between pairs of teams—exact numbers depended on which teams were participating. In RoXYBot-02, we arbitrarily fixed the number of samples $n = 50$. No attempt was made to optimize this parameter. None of the other algorithms used in this study—RoXYBot-00, MU, and $\overline{\text{MU}}$ —have any tunable parameters.

Before running any experiments, we played 500 training games between RoXYBot-00 and MU, initializing price estimates to the competitive equilibrium prices derived in Wellman, *et al.* [11]. Using data collected from these training games, we generated distributions over clearing prices for each good. The distributions were represented by a lookup table over five salient features of the domain, the details of which are beyond the scope of this paper. In each cell of the table, 10 numbers were stored, corresponding to the predictions at percentiles 5, 10, 15, ..., 95. This representation was chosen for its simplicity and its weak assumptions about the shape of the underlying price

distributions. We sought to capture the highly skewed and multimodal distributions that arise in practice in TAC games.

A sample output after the run of one game instance is shown in the table below.

Agent	Score	Rank
RoXYBot-02	3802	1
RoXYBot-02	3116	6
RoXYBot-02	3166	5
RoXYBot-02	3447	4
RoXYBot-00	2521	8
RoXYBot-00	3696	2
RoXYBot-00	2788	7
RoXYBot-00	3600	3

Generally speaking, scores in our experiments are not as high as scores in actual competitions, because bidding marginal utilities, as do all eight agents in our experiments, tends to lead to high prices.

4.2 Evaluation

To evaluate our results, we use two statistical tests: the z -test, which we use to compare scores, and the Wilcoxon test, which we use to compare rankings. In our context, the inputs to the z -test are two sample datasets of scores, one per team, over many game instances. The z -test outputs the probability that the difference between the means of these datasets is positive: *i.e.*, the probability that the mean of the second is greater than the mean of the first. Our input to the Wilcoxon test⁴ is a list of pairs of average rankings, one per game. The test measures the significance of the difference between these rankings.

4.3 Results

Our experimental results are depicted numerically in Table 1 and graphically in Figure 2. MU outperforms expected MU; RoXYBot-00 outperforms MU; and, RoXYBot-02 outperforms RoXYBot-00. Moreover, these results are transitive: RoXYBot-00 outperforms expected MU bidding; RoXYBot-02 outperforms MU bidding; and RoXYBot-02 outperforms expected MU bidding.

The numbers in Table 1 reveal that with high confidence, RoXYBot-02 is expected to score higher than RoXYBot-00 and MU. On the other hand, although RoXYBot-00 is expected to score higher than MU, the lower confidence level supporting this conclusion makes this result less credible. The outcome of all

⁴For a description of the Wilcoxon test, visit http://fonsg3.let.uva.nl/Service/Statistics/Signed_Rank_Test.html.

Teams	Means		<i>z</i> -test	Wilcoxon	Games
MU < MU	964	1908	.999	.999	25
MU < RoxyBot-00	1508	1612	.793	.803	75
RoxyBot-00 < RoxyBot-02	1837	2031	.977	.996	50
MU < RoxyBot-00	1334	2034	.999	.999	25
MU < RoxyBot-02	1705	1987	.976	.993	50
MU < RoxyBot-02	915	1920	.999	.999	25

Table 1: Numerical Results: Means, *z*-test, Wilcoxon test, and Sample Size.

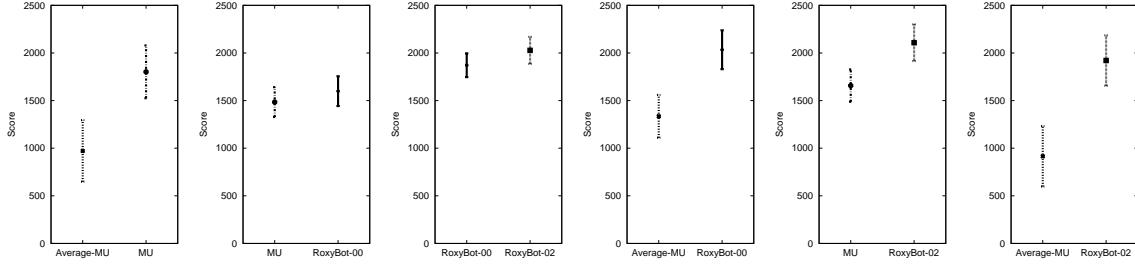


Figure 2: Graphical Results: 95% Confidence Intervals on the Means. All agent strategies clearly outperform expected MU bidding. Other graphs reveal narrower performance distinctions.

Wilcoxon tests, and the graphs depicted in Figure 2, reinforce the outcome of the *z*-tests.

Notably, with high confidence in the *z*-tests and Wilcoxon tests, all strategies are expected to score higher than expected MU, and with 95% confidence, all strategies outperform expected MU. Since expected MU bidding is the optimal policy in sequential auctions, we conclude that TAC classic hotel auctions are more similar in spirit to simultaneous auctions than to sequential auctions.

References

- [1] R. Bellman. *Dynamic Programming*. Princeton University Press, New Jersey, 1957.
- [2] M. Benisch, A. Greenwald, V. Naroditskiy, and M. Tschantz. A stochastic programming approach to TAC SCM. In *ACM Conference on Electronic Commerce*, pages 152–160, May 2004.
- [3] John Birge and Francois Louveaux. *Introduction to Stochastic Programming*. Springer, New York, 1997.
- [4] Craig Boutilier, Moisés Goldszmidt, and Bikash Sabata. Sequential auctions for the allocation of resources with complementarities. In *Proceedings of Sixteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 527–534, August 1999.
- [5] J. Boyan and A. Greenwald. Bid determination in simultaneous auctions: An agent architecture. In *Proceedings of Third ACM Conference on Electronic Commerce*, 210–212 2001.
- [6] A. Greenwald. Bidding marginal utility in simultaneous auctions. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1463–1464, August 2003.
- [7] A. Greenwald and J. Boyan. Bidding algorithms for simultaneous auctions: A case study. In *Proceedings of Third ACM Conference on Electronic Commerce*, 115–124 2001.
- [8] A. Shapiro and T. Homen-De-Mello. On rate convergence of monte carlo approximations of stochastic programs. *SIAM Journal on Optimization*, 11:70–86, 2001.
- [9] P. Stone, R.E. Schapire, M.L. Littman, J.A. Csirik, and D. McAllester. Decision-theoretic bidding based on learned density models in simultaneous, interacting auctions. *Journal of Artificial Intelligence Research*, 19:209–242, 2003.
- [10] M.P. Wellman, A. Greenwald, P. Stone, and P.R. Wurman. The 2001 Trading Agent Competition. In *Proceedings of the Fourteenth Innovative Applications of Artificial Intelligence Conference*, pages 935–941, July 2002.
- [11] M.P. Wellman, D.M. Reeves, K.M. Lochner, and Y. Vorobeychik. Price prediction in a Trading Agent Competition. To appear, JAIR, 2003.

RoxyBot-06: Stochastic Prediction and Optimization in TAC Travel

Amy Greenwald

*Department of Computer Science, Brown University
Providence, RI 02912 USA*

AMY@CS.BROWN.EDU

Seong Jae Lee

*Computer Science and Engineering, University of Washington
Seattle, WA 98195 USA*

SEONGJAE@U.WASHINGTON.EDU

Victor Naroditskiy

*Department of Computer Science, Brown University
Providence, RI 02912 USA*

VNARODIT@CS.BROWN.EDU

Abstract

In this paper, we describe our autonomous bidding agent, RoxyBot, who emerged victorious in the travel division of the 2006 Trading Agent Competition in a photo finish. At a high level, the design of many successful trading agents can be summarized as follows: (i) price prediction: build a model of market prices; and (ii) optimization: solve for an approximately optimal set of bids, given this model. To predict, RoxyBot builds a stochastic model of market prices by simulating *simultaneous ascending auctions*. To optimize, RoxyBot relies on the *sample average approximation* method, a stochastic optimization technique.

1. Introduction

The annual Trading Agent Competition (TAC) challenges its entrants to design and build autonomous agents capable of effective trading in an online travel¹ shopping game. The first TAC, held in Boston in 2000, attracted 16 entrants from six countries in North America, Europe, and Asia. Excitement generated from this event led to refinement of the game rules, and continuation of regular tournaments with increasing levels of competition over the next six years. Year-by-year, entrants improved their designs, developing new ideas and building on previously successful techniques. Since TAC's inception, the lead author has entered successive modifications of her autonomous trading agent, RoxyBot. This paper reports on RoxyBot-06, the latest incarnation and the top scorer in the TAC-06 tournament.

The key feature captured by the TAC travel game is that goods are highly interdependent (e.g., flights and hotels must be coordinated), yet the markets for these goods operate independently. A second important feature of TAC is that agents trade via three different kinds of market mechanisms, each of which presents distinct challenges. Flights are traded in a posted-price environment, where a designated party sets a price that the other parties

1. There are now four divisions of TAC: Travel, Supply Chain Management (SCM), CAT (TAC backwards), and Ad Auctions (AA). This paper is concerned only with the first; for a description of the others, see the papers by Arunachalam and Sadeh (2005), Cai et al. (2009), Jordan and Wellman (2009), respectively. In this paper, when we say TAC, we mean TAC Travel.

must “take or leave.” Hotels are traded in simultaneous ascending auctions, like the FCC spectrum auctions. Entertainment tickets are traded in continuous double auctions, like the New York Stock Exchange. In grappling with all three mechanisms while constructing their agent strategies, participants are confronted by a number of interesting problems.

The success of an autonomous trading agent such as a TAC agent often hinges upon the solutions to two key problems: (i) *price prediction*, in which the agent builds a model of market prices; and (ii) *optimization*, in which the agent solves for an approximately optimal set of bids, given this model. For example, at the core of RoxyBot’s 2000 architecture (Greenwald & Boyan, 2005) was a *deterministic* optimization problem, namely how to bid given price predictions in the form of point estimates. In spite of its effectiveness in the TAC-00 tournament, a weakness of the 2000 design was that RoxyBot could not explicitly reason about variance within prices. In the years since 2000, we recast the key challenges faced by TAC agents as several different *stochastic* bidding problems (see, for example, the paper by Greenwald & Boyan, 2004), whose solutions exploit price predictions in the form of distributions. In spite of our perseverance, RoxyBot fared unimpressively in tournament conditions year after year, until 2006. Half a decade in the laboratory spent searching for bidding heuristics that can exploit stochastic information at reasonable computational expense finally bore fruit, as RoxyBot emerged victorious in TAC-06. In a nutshell, the secret of RoxyBot-06’s success is: (hotel) price prediction by simulating simultaneous ascending auctions, and optimization based on the sample average approximation method. Details of our approach are the subject of the present article.

Overview This paper is organized as follows. Starting in Section 2, we summarize the TAC market game. Next, in Section 3, we present a high-level view of RoxyBot’s 2006 architecture. In Section 4, we describe RoxyBot’s price prediction techniques for flights, hotels, and entertainment, in turn. Perhaps of greatest interest is our hotel price prediction method. Following Wellman et al. (2005), we predict hotel prices by computing approximate competitive equilibrium prices. Only, instead of computing those prices by running the tâtonnement process, we simulate simultaneous ascending auctions. Our procedure is simpler to implement than tâtonnement, yet achieves comparable performance, and runs sufficiently fast. In Section 5, we describe RoxyBot’s optimization technique: sample average approximation. We argue that this approach is optimal in pseudo-auctions, an abstract model of auctions. In Section 6.1, we describe simulation experiments in a controlled testing environment which show that our combined approach—simultaneous ascending auctions for hotel price prediction and sample average approximation for bid optimization—performs well in practice in comparison with other reasonable bidding heuristics. In Section 6.2, we detail the results of the TAC-06 tournament, further validating the success of RoxyBot-06’s strategy, and reporting statistics that shed light on the bidding strategies of other participating agents. Finally, in Section 7, we evaluate the collective behavior of the autonomous agents in the TAC finals since 2002. We find that the accuracy of competitive equilibrium calculations has varied from year to year and is highly dependent on the particular agent pool. Still, generally speaking, the collective appears to be moving toward competitive equilibrium behavior.

2. TAC Market Game: A Brief Summary

In this section, we summarize the TAC game. For more details, see <http://www.sics.se/tac/>.

Eight agents play the TAC game. Each is a simulated travel agent whose task is to organize itineraries for its clients to travel to and from “TACTown” during a five day (four night) period. In the time allotted (nine minutes), each agent’s objective is to procure travel goods as inexpensively as possible, trading off against the fact that those goods are ultimately compiled into feasible trips that satisfy its client preferences to the greatest extent possible. The agents know the preferences of their own eight clients only, not the other 56.

Travel goods are sold in simultaneous auctions as follows:

- Flight tickets are sold by “TACAir” in dynamic posted-pricing environments. There are flights both to and from TACTown on each applicable day. No resale of flight tickets by agents is permitted.

Flight price quotes are broadcast by the TAC server every ten seconds.

- Hotel reservations are sold by the “TAC seller” in multi-unit ascending call markets. Specifically, 16 hotel reservations are sold in each hotel auction to the 16 highest bidders at the 16th highest price. The TAC seller runs eight hotel auctions, one per night-hotel combination (recall that travel takes place during a four night period; moreover, there are two hotels: a good one and a bad one). No resale of hotel reservations by agents is permitted. Nor is bid withdrawal allowed.

More specifically, the eight hotel auctions clear on the minute with exactly one auction closing at each of minutes one through eight. (The precise auction to close is chosen at random, with all open auctions equally likely to be selected.) For the auction that closes, the TAC server broadcasts the final closing price, and informs each agent of its winnings. For the others, the TAC server reports the current ask price, and informs each agent of its “hypothetical quantity won” (HQW).

- Agents are allocated an initial endowment of entertainment tickets, which they trade among themselves in continuous double auctions (CDAs). There are three entertainment events scheduled each day.

Although the event auctions clear continuously, price quotes are broadcast only every 30 seconds.

One of the primary challenges posed by TAC is to design and build autonomous agents that bid effectively on interdependent (i.e., complementary or substitutable) goods that are sold in separate markets. Flight tickets and hotel reservations are complementary because flights are not useful to a client without the corresponding hotel reservations, nor vice versa. Tickets to entertainment events (e.g., the Boston Red Sox and the Boston Symphony Orchestra) are substitutable because a client cannot attend multiple events simultaneously.

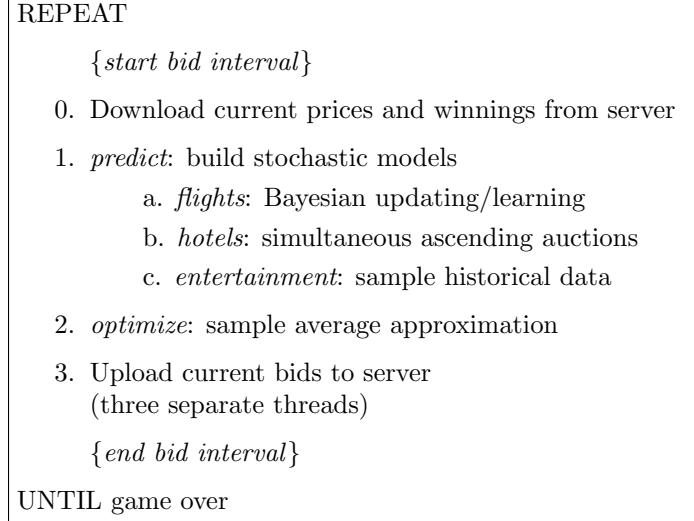


Table 1: A high-level view of RoxyBot-06’s architecture.

3. RoxyBot-06’s Architecture: A High-Level View

In our approach to the problem of bidding on interdependent goods in the separate TAC markets, we adopt some simplifying assumptions. Rather than tackle the game-theoretic problem of characterizing strategic equilibria, we focus on a single agent’s (decision-theoretic) problem of optimizing its own bidding behavior, assuming the other agents’ strategies are fixed. In addition, we assume that the environment can be modeled in terms of the agent’s predictions about market clearing prices. These prices serve to summarize the relevant information hidden in other agents’ bidding strategies. These two assumptions—fixed other-agent behaviors and market information encapsulated by prices—support the modular design of RoxyBot-06 and many other successful TAC agents, which consists of two key stages: (i) price prediction; and (ii) optimization.

The optimization problem faced by TAC agents is a dynamic one that incorporates aspects of sequentiality as well as simultaneity in auctions. The markets operate simultaneously, but in addition, prices are discovered incrementally over time. In principle, a clairvoyant agent—one with knowledge of future clearing prices—could justifiably employ an open-loop strategy: it could solve the TAC optimization problem once at the start of the game and place all its bids accordingly, never reconsidering those decisions. A more practical alternative (and the usual approach taken in TAC²), is to incorporate into an agent’s architecture a closed loop, or *bidding cycle*, enabling the agent to condition its behavior on the evolution of prices. As price information is revealed, the agent improves its price predictions, and reoptimizes its bidding decisions, repeatedly.

One distinguishing feature of RoxyBot-06 is that it builds stochastic models of market clearing prices, rather than predicting clearing prices as point estimates. Given its stochastic price predictions, stochastic optimization lies at the heart of RoxyBot-06. Assuming time is

2. An exception is *livingagents* (Fritschi & Dorer, 2002), the winner of TAC 2001.

discretized into stages, or bid intervals, during each iteration of its bidding cycle, RoxyBot-06 faces an n -stage stochastic optimization problem, where n is the number of stages remaining in the game. The key input to this optimization problem is a sequence of $n - 1$ stochastic models of future prices, each one a joint probability distribution over all goods conditioned on past prices and past hotel closings. The solution to this optimization problem, and the output of each iteration of the bidding cycle, is a vector of bids, one per good (or auction).

Table 1 presents a high-level view of RoxyBot-06’s architecture, emphasizing its bidding cycle. At the start of each bid interval, current prices and winnings are downloaded from the TAC server. Next, the key prediction and optimization routines are run. In the prediction module, stochastic models of flight, hotel, and entertainment prices are built. In the optimization module, bids are constructed as an approximate solution to an n -stage stochastic optimization problem. Prior to the end of each bid interval, the agents’ bids are uploaded to the TAC server using three separate threads: (i) the flight thread bids on a flight only if its price is near its predicted minimum; (ii) the hotel thread bids on open hotels only if it is moments before the end of a minute; and (iii) the entertainment thread places bids immediately.

We discuss the details of RoxyBot-06’s price prediction module first, and its optimization module second.

4. Price Prediction

In this section, we describe how RoxyBot-06 builds its stochastic models of flight, hotel, and event prices. Each model is a discrete probability distribution, represented by a set of “scenarios.” Each scenario is a vector of “future” prices—prices at which goods can be bought and sold after the current stage. For flights, the price prediction model is not stochastic: the future buy price is simply RoxyBot-06’s prediction of the expected minimum price during the current stage. For hotels, the future buy prices are predicted by Monte Carlo simulations of simultaneous ascending auctions to approximate competitive equilibrium prices. There are no current buy prices for hotels. For entertainment, RoxyBot-06 predicts future buy and sell prices based on historical data. Details of these price prediction methods are the focus of this section.

4.1 Flights

Efforts to deliberate about flight purchasing start with understanding the TAC model of flight price evolution.

4.1.1 TAC FLIGHT PRICES’ STOCHASTIC PROCESS

Flight prices follow a biased random walk. They are initialized uniformly in the range $[250, 400]$, and constrained to remain in the range $[150, 800]$. At the start of each TAC game instance, a bound z on the final perturbation value is selected for each flight. These bounds are not revealed to the agents. What is revealed to the agents is a sequence of random flight prices. Every ten seconds, TACAIR perturbs the price of each flight by a random value that depends on the hidden parameter z and the current time t as follows: given constants $c, d \in \mathbb{R}$ and $T > 0$, each (intermediate) bound on the perturbation value

is a linear function of t :

$$x(t, z) = c + \frac{t}{T}(z - c) \quad (1)$$

The perturbation value at time t is drawn uniformly from one of the following ranges (see Algorithm 1):

- $U[-c, x(t, z)]$, if $x(t, z) > 0$
- $U[-c, +c]$, if $x(t, z) = 0$
- $U[x(t, z), +c]$, if $x(t, z) < 0$

Observe that the expected perturbation value in each case is simply the average of the corresponding upper and lower bounds. In particular,

- if $x(t, z) > c$, then the expected perturbation is positive;
- if $x(t, z) \in (0, c)$, then the expected perturbation is negative;
- if $x(t, z) \in (-c, 0)$, then the expected perturbation is positive;
- otherwise, if $x(t, z) \in \{-c, 0, c\}$, then the expected perturbation is zero.

Moreover, using Equation 1, we can compute the expected perturbation value conditioned on z :

- if $z \in [0, c]$, then $x(t, z) \in [0, c]$, so prices are expected not to increase;
- if $z \in [c, c + d]$, then $x(t, z) \in [c, c + d]$, so prices are expected not to decrease;
- if $z \in [-c, 0]$, then $x(t, z) \in [-c, c]$, so prices are expected not to increase while $t \leq \frac{cT}{c-z}$ and they are expected not to decrease while $t \geq \frac{cT}{c-z}$.

The TAC parameters are set as follows: $c = 10$, $d = 30$, $T = 540$, and z uniformly distributed in the range $[-c, d]$. Based on the above discussion, we note the following: given no further information about z , TAC flight prices are expected to increase (i.e., the expected perturbation is positive); however, conditioned on z , TAC flight prices may increase *or* decrease (i.e., the expected perturbation can be positive or negative).

4.1.2 RoxyBot-06's FLIGHT PRICES PREDICTION METHOD

Although the value of the hidden parameter z is never revealed to the agents, recall that the agents do observe sample flight prices, say y_1, \dots, y_t , that depend on this value. This information can be used to model the probability distribution $P_t[z] \equiv P[z \mid y_1, \dots, y_t]$. Such a probability distribution can be estimated using Bayesian updating. Before RoxyBot-06, agents Walverine (Cheng et al., 2005) and Mertacor (Toulis et al., 2006) took this approach. Walverine uses Bayesian updating to compute the next expected price perturbation and then compares that value to a threshold, postponing its flight purchases if prices are not expected to increase by more than that threshold. Mertacor uses Bayesian updating to estimate the time at which flight prices will reach their minimum value. RoxyBot uses Bayesian updating to compute the expected minimum price, as we now describe.

Algorithm 1 getRange(c, t, z)

```

compute  $x(t, z)$  {Equation 1}
if  $x(t, z) > 0$  then
     $a = -c; b = \lceil x(t, z) \rceil$ 
else if  $x(t, z) < 0$  then
     $a = \lfloor x(t, z) \rfloor; b = +c$ 
else
     $a = -c; b = +c$ 
end if
return  $[a, b]$  {range}

```

RoxyBot-06's implementation of Bayesian updating is presented in Algorithm 2. Letting $Q_0[z] = \frac{1}{c+d} = P[z]$, the algorithm estimates $P_{t+1}[z] = P[z | y_1, \dots, y_{t+1}]$ as usual:

$$P[z | y_1, \dots, y_t] = \frac{P[y_1, \dots, y_t | z]P[z]}{\sum_{z'} P[y_1, \dots, y_t | z']P[z'] dz'} \quad (2)$$

where

$$P[y_1, \dots, y_t | z] = \prod_{i=1}^t P[y_i | y_1, \dots, y_{i-1}, z] \quad (3)$$

$$= \prod_{i=1}^t P[y_i | z] \quad (4)$$

Equation 4 follows from the fact that future observations are independent of past observations; observations depend only on the hidden parameter z .

The only thing left to explain is how to set the values $P[y_i | z]$, for $i = 1, \dots, t$. As described in the pseudocode, this is done as follows: if y_{t+1} is within the appropriate range at that time, then this probability is set uniformly within the bounds of that range; otherwise, it is set to 0. Presumably, Walverine's and Mertacor's implementations of Bayesian updating are not very different from this one.³ However, as alluded to above, how the agents make use of the ensuing estimated probability distributions does differ.

RoxyBot-06 predicts each flight's price to be its expected minimum price. This value is computed as follows (see Algorithm 3): for each possible value of the hidden parameter z , RoxyBot simulates an “expected” random walk, selects the minimum price along this walk, and then outputs as its prediction the expectation of these minima, averaging according to $P_t[z]$. We call this random walk “expected,” since the perturbation value Δ is an expectation (i.e., $\Delta = \frac{b-a}{2}$) instead of a sample (i.e., $\Delta \sim U[a, b]$). By carrying out this computation, RoxyBot generates flight price predictions that are point estimates. The implicit decision to make only RoxyBot-06's hotel and event price predictions stochastic was made based on our intuitive sense of the time vs. accuracy tradeoffs in RoxyBot's optimization module, and hence warrants further study.

3. We provide details here, because corresponding details for the other agents do not seem to be publicly available.

Algorithm 2 Flight_Prediction(c, d, t, y_{t+1}, Q_t)

```

for all  $z \in \{-c, -c + 1, \dots, d\}$  do
     $[a, b] = \text{getRange}(c, t, z)$ 
    if  $y_{t+1} \in [a, b]$  then
         $P[y_{t+1} | z] = \frac{1}{b-a}$ 
    else
         $P[y_{t+1} | z] = 0$ 
    end if
     $Q_{t+1}[z] = P[y_{t+1} | z]Q_t[z]$ 
end for{update probabilities}
for all  $z \in \{-c, -c + 1, \dots, d\}$  do
     $P_{t+1}[z] = \frac{Q_{t+1}[z]}{\sum_{z'} Q_{t+1}[z'] dz'}$ 
end for{normalize probabilities}
return  $P_{t+1}$  {probabilities}

```

Algorithm 3 Expected_Minimum_Price(c, t, t', p_t, P_t)

```

for all  $z \in R$  do
     $\min[z] = +\infty$ 
    for  $\tau = t + 1, \dots, t'$  do
         $[a, b] = \text{getRange}(c, \tau, z)$ 
         $\Delta = \frac{b-a}{2}$  {expected perturbation}
         $p_\tau = p_{\tau-1} + \Delta$  {perturb price}
         $p_\tau = \max(150, \min(800, p_\tau))$ 
        if  $p_\tau < \min[z]$  then
             $\min[z] = p_\tau$ 
        end if
    end for
end for
return  $\sum_z P_t[z] \min[z] dz$ 

```

4.2 Hotels

In a competitive market where each individual's effect on prices is negligible, equilibrium prices are prices at which supply equals demand, assuming all producers are profit-maximizing and all consumers are utility-maximizing. RoxyBot-06 predicts hotel prices by simulating *simultaneous ascending auctions* (SimAA) (Cramton, 2006), in an attempt to approximate competitive equilibrium (CE) prices. This approach is inspired by Walverine's (Cheng et al., 2005), where the *tâtonnement* method (Walras, 1874) is used for the same purpose.

4.2.1 SIMULTANEOUS ASCENDING AUCTIONS

Let \vec{p} denote a vector of prices. If $\vec{y}(\vec{p})$ denotes the cumulative supply of all producers, and if $\vec{x}(\vec{p})$ denotes the cumulative demand of all consumers, then $\vec{z}(\vec{p}) = \vec{x}(\vec{p}) - \vec{y}(\vec{p})$ denotes the

excess demand in the market. The tâtonnement process adjusts the price vector at iteration $n + 1$, given the price vector at iteration n and an adjustment rate α_n as follows: $\vec{p}_{n+1} = \vec{p}_n + \alpha_n \vec{z}(\vec{p}_n)$. SimAA adjusts the price vector as follows: $\vec{p}_{n+1} = \vec{p}_n + \alpha \max\{\vec{z}(\vec{p}_n), 0\}$, for some fixed value of α . Both of these processes continue until excess demand is non-positive: i.e., supply exceeds demand.

Although competitive equilibrium prices are not guaranteed to exist in TAC markets (Cheng et al., 2003), the SimAA adjustment process, is still guaranteed to converge: as prices increase, demand decreases while supply increases; hence, supply eventually exceeds demand. The only parameter to the SimAA method is the magnitude α of the price adjustment. The smaller this value, the more accurate the approximation (assuming CE prices exist), so the value of α can be chosen to be the lowest value that time permits.

The tâtonnement process, on the other hand, is more difficult to apply as it is not guaranteed to converge. The Walverine team dealt with the convergence issue by decaying an initial value of α . However, careful optimization was required to ensure convergence to a reasonable solution in a reasonable amount of time. In fact, Walverine found it helpful to set initial prices to certain non-zero values. This complexity is not present when using simultaneous ascending auctions to approximate competitive equilibrium prices.

4.2.2 PREDICTION QUALITY

In TAC, cumulative supply is fixed. Hence, the key to computing excess demand is to compute cumulative demand. Each TAC agent knows the preferences of its own clients, but must estimate the demand of the others. Walverine computes a single hotel price prediction (a point estimate) by considering its own clients' demands together with those of 56 “expected” clients. Briefly, the utility of an expected client is an average across travel dates and hotel types augmented with fixed entertainment bonuses that favor longer trips (see the paper by Cheng et al., 2005, for details). In contrast, RoxyBot-06 builds a stochastic model of hotel prices consisting of S scenarios by considering its own clients' demands together with S random samples of 56 clients. A (random or expected) client's demand is simply the quantity of each good in its optimal package, given current prices. The cumulative demand is the sum total of all client's individual demands.

In Figure 1, we present two scatter plots that depict the quality of various hotel price predictions at the beginning of the TAC 2002 final games. All price predictions are evaluated using two metrics: Euclidean distance and the “expected value of perfect prediction” (EVPP). Euclidean distance is a measure of the difference between two vectors, in this case the actual and the predicted prices. The value of perfect prediction (VPP) for a client is the difference between its surplus (value of its preferred package less price) based on actual and predicted prices. EVPP is the VPP averaged over the distribution of client preferences.⁴

On the left, we plot the predictions generated using the competitive equilibrium approximation methods, tâtonnement and SimAA, *both* with fixed $\alpha = \frac{1}{24}$, making expected, random, and exact predictions. The “exact” predictions are computed based on the actual clients in the games, not just the client distribution; hence, they serve as a lower bound on the performance of these techniques on this data set. Under both metrics, and for both expected and random, SimAA's predictions outperform tâtonnement's.

4. See the paper by Wellman et al. (2004) for details.

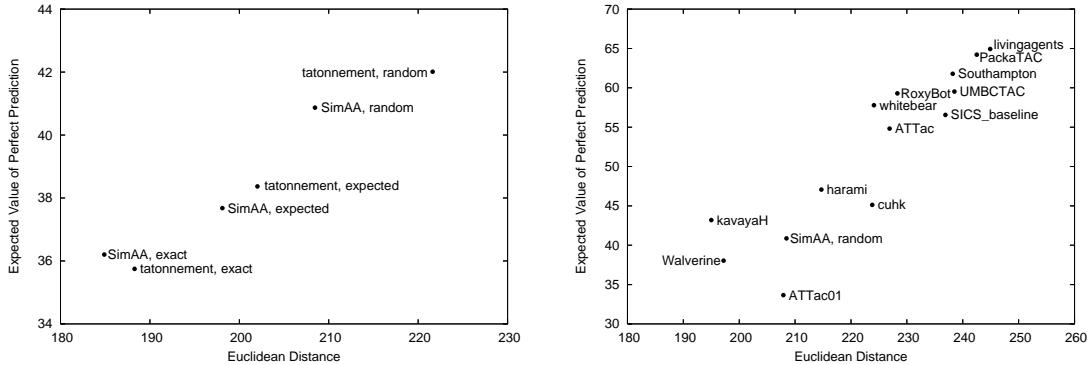


Figure 1: EVPP and Euclidean Distance for the CE price prediction methods (tâtonnement and SimAA with $\alpha = \frac{1}{24}$; expected, random, and exact) and the TAC 2002 agents' predictions in the 2002 finals (60 games). The plot on the left shows that SimAA's predictions are better than tâtonnement's and that expected's are better than random's. RoxyBot-06's method of hotel price prediction (SimAA, Random) is plotted again on the right. Note the differences in scales between the two plots.

Since α is fixed, and tâtonnement is not guaranteed to converge under this condition, this outcome is not entirely surprising. What is interesting, though, is that SimAA expected performs comparably to Walverine (see the right plot).⁵ This is interesting because SimAA has fewer parameter settings than tâtonnement—only a single α value as compared to an initial α value together with a decay schedule—and moreover, we did not optimize its parameter setting. Walverine's parameter settings, on the other hand, were highly optimized.

We interpret each prediction generated using randomly sampled clients as a sample scenario, so that a set of such scenarios represents draws from a probability distribution over CE prices. The corresponding vector of predicted prices that is evaluated is actually the average of multiple (40) such predictions; that is, we evaluate an estimate of the mean of this probability distribution. The predictions generated using sets of random clients are not as good as the predictions with expected clients (see Figure 1 left), although with more than 40 sets of random clients, the results might improve. Still, the predictions with random clients comprise RoxyBot-06's stochastic model of hotel prices, which is key to its bidding strategy. Moreover, using random clients helps RoxyBot-06 make better interim predictions later in the game as we explain next.

4.2.3 PREDICTION QUALITY OVER TIME: INTERIM PRICE PREDICTION

The graphs depicted in Figure 1 pertain to hotel price predictions made at the beginning of the game, when all hotel auctions are open. In those CE computations, prices are initialized to 0. As hotel auctions close, RoxyBot-06 updates the predicted prices of the hotel auctions

5. With the exception of the RoxyBot-06 data point (i.e., SimAA random), this plot was produced by the Walverine team (Wellman et al., 2004).

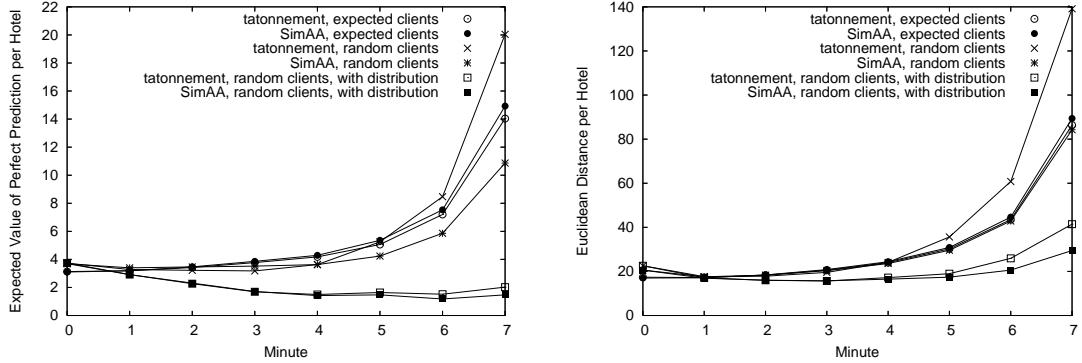


Figure 2: EVPP and Euclidean Distance in TAC 2006 finals (165 games) of the CE price prediction methods with and without distribution as the game progresses. Distribution improves prediction quality.

that remain open. We experimented with two ways of constructing interim price predictions. The first is to initialize and lower bound the prices in the hotel markets at their closing (for closed auctions) or current ask (for open auctions) prices while computing competitive equilibrium prices.⁶ The second differs in its treatment of closed auctions: we simulate a process of distributing the goods in the closed auctions to the clients who want them most, and then exclude the closed markets (i.e., fix prices at ∞) from further computations of competitive equilibrium prices.

Regarding the second method—the distribution method—we determine how to distribute goods by computing competitive equilibrium prices again. As explained in Algorithm 4, all hotels (in both open and closed auctions) are distributed to *random* clients by determining who is willing to pay the competitive equilibrium prices for what. It is not immediately obvious how to distribute goods to expected clients; hence, we enhanced only the prediction methods with random clients with distribution.

Figure 2, which depicts prediction quality over time, shows that the prediction methods enhanced with distribution are better than the predictions obtained by merely initializing the prices of closed hotel auctions at their closing prices. Hotels that close early tend to sell for less than hotels that close late; hence, the prediction quality of any method that makes decent initial predictions is bound to deteriorate if those predictions remain relatively constant throughout the game.

4.2.4 RUN TIME

Table 2 shows the run times of the CE prediction methods on the TAC 2002 (60 games) and TAC 2006 (165 games) finals data set at minute 0, as well as their run times during

6. At first blush, it may seem more sensible to *fix* the prices of closed hotels at their closing prices, rather than merely lower bound them (i.e., allow them to increase). If some hotel closed at an artificially low price, however, and if that price were not permitted to increase, then the predicted prices of the hotels complementing the hotel in question would be artificially high.

Algorithm 4 Distribute

```

1: for all hotel auctions  $h$  do
2:   initialize price to 0
3:   initialize supply to 16
4: end for
5: compute competitive equilibrium prices {Tâtonnement or SimAA}
6: for all closed hotel auctions  $h$  do
7:   distribute units of  $h$  to those who demand them at the computed competitive equi-
      librium prices
8:   distribute any leftover units of  $h$  uniformly at random
9: end for

```

minutes 1–7 on the TAC 2006 finals data set. What the numbers in this table convey is that SimAA’s run time, even with distribution, is reasonable. For example, at minute 0, SimAA sample takes on the order of 0.1 seconds. At minutes 1–7, this method without distribution runs even faster. This speed increase occurs because CE prices are bounded below by current ask prices and above by the maximum price a client is willing to pay for a hotel, and current ask prices increase over time, correspondingly reducing the size of the search space. SimAA sample with distribution at minutes 1–7 takes twice as long as SimAA sample without distribution at minute 0 because of the time it takes to distribute goods, but the run time is still only (roughly) 0.2 seconds. Our implementation of tâtonnement runs so slowly because we fixed α instead of optimizing the tradeoff between convergence rate and accuracy, so the process did not converge, and instead ran for the maximum number of iterations (10,000). In summary, SimAA is simpler than tâtonnement to implement, yet performs comparably to an optimized version of tâtonnement (i.e., Walverine), and runs sufficiently fast.

	Exp Tât	Exp SimAA	Sam Tât	Sam SimAA	Dist Tât	Dist SimAA
2002, minute 0	2213	507	1345	157	—	—
2006, minute 0	2252	508	1105	130	1111	128
2006, average 1–7	2248	347	1138	97	2249	212

Table 2: Run times for the CE price prediction methods, in milliseconds. Experiments were run on AMD Athlon(tm) 64 bit 3800+ dual core processors with 2M of RAM.

4.2.5 SUMMARY

The simulation methods discussed in this section—the tâtonnement process and simultaneous ascending auctions—were employed to predict hotel prices only. (In our simulations, flight prices are fixed at their expected minima, and entertainment prices are fixed at 80.) In principle, competitive equilibrium (CE) prices could serve as predictions in all TAC markets. However, CE prices are unlikely to be good predictors of flight prices, since flight prices are determined exogenously. With regard to entertainment tickets, CE prices might

have predictive power; however, incorporating entertainment tickets into the tâtonnement and SimAA calculations would have been expensive. (In our simulations, following Wellman et al., 2004, client utilities are simply augmented with fixed entertainment bonuses that favor longer trips.) Nonetheless, in future work, it could be of interest to evaluate the success of these or related methods in predicting CDA clearing prices.

Finally, we note that we refer to our methods of computing excess demand as “client-based” because we compute the demands of each client on an individual basis. In contrast, one could employ an “agent-based” method, whereby the demands of agents, not clients, would be calculated. Determining an agent’s demands involves solving so-called *completion*, a deterministic (prices are known) optimization problem at the heart of RoxyBot-00’s architecture (Greenwald & Boyan, 2005). As TAC completion is NP-hard, the agent-based method of predicting hotel prices is too expensive to be included in RoxyBot-06’s inner loop. In designing RoxyBot-06, we reasoned that an architecture based on a stochastic pricing model generated using the client-based method and randomly sampled clients would outperform one based on a point estimate pricing model generated using the agent-based method and some form of expected clients, but we did not verify our reasoning empirically.

4.3 Entertainment

During each bid interval, RoxyBot-06 predicts current and future buy and sell prices for tickets to all entertainment events. These price predictions are optimistic: the agent assumes it can buy (or sell) goods at the least (or most) expensive prices that it expects to see before the end of the game. More specifically, each current price prediction is the best predicted price during the current bid interval.

RoxyBot-06’s estimates of entertainment ticket prices are based on historical data from the past 40 games. To generate a scenario, a sample game is drawn at random from this collection, and the sequences of entertainment bid, ask, and transaction prices are extracted. Given such a history, for each auction a , let $trade_{ai}$ denote the price at which the last trade before time i transacted; this value is initialized to 200 for buying and 0 for selling. In addition, let bid_{ai} denote the bid price at time i , and let ask_{ai} denote the ask price at time i .

RoxyBot-06 predicts the future buy price in auction a after time t as follows:

$$future_buy_{at} = \min_{i=t+1, \dots, T} \min\{trade_{ai}, ask_{ai}\} \quad (5)$$

In words, the future buy price at each time $i = t + 1, \dots, T$ is the minimum of the ask price after time i and the most recent trade price. The future buy price at time t is the minimum across the future buy prices at all later times. The future sell price after time t is predicted analogously:

$$future_sell_{at} = \max_{i=t+1, \dots, T} \max\{trade_{ai}, bid_{ai}\} \quad (6)$$

Arguably, RoxyBot-06’s entertainment predictions are made in the simplest possible way: past data are future predictions. It is likely one could improve upon this naive approach by using a generalization technique capable of “learning” a distribution over these data, and then sampling from the learned distribution.

4.4 Summary

In this section, we described RoxyBot-06’s price prediction methods. The key ideas, which may be transferable if not beyond TAC, at least to other TAC agents, are as follows:

1. RoxyBot makes stochastic price predictions. It does so by generating a set of so-called “scenarios,” where each scenario is a vector of future prices.
2. For each flight, RoxyBot uses Bayesian updating to predict its expected minimum price.
3. For hotels, RoxyBot uses a method inspired by Walcerine’s: it approximates competitive equilibrium prices by simulating simultaneous ascending auctions, rather than the usual tâtonnement process.

5. Optimization

Next, we characterize RoxyBot-06’s optimization routine. It is (i) stochastic, (ii) global, and (iii) dynamic. It takes as input stochastic price predictions; it considers its flight, hotel, and entertainment bidding decisions in unison; and it simultaneously reasons about bids to be placed in both current and future stages of the game.

5.1 Abstract Auction Model

Recall that our treatment of bidding is decision-theoretic, rather than game-theoretic. In particular, we focus on a single agent’s problem of optimizing its own bidding behavior, assuming the other agents’ strategies are fixed. In keeping with our basic agent architecture, we further assume that the environment can be modeled in terms of the agent’s predictions about market clearing prices. We introduce the term *pseudo-auction* to refer to a market mechanism defined by these two assumptions—fixed other-agent behaviors and market information encapsulated by prices. The optimization problem that RoxyBot solves is one of bidding in pseudo-auctions, not (true) auctions. In this section, we formally develop this abstract auction model and relate it to TAC auctions; in the next, we define and propose heuristics to solve various pseudo-auction bidding problems.

5.1.1 BASIC FORMALISM

In this section, we formalize the basic concepts needed to precisely formulate bidding under uncertainty as an optimization problem, including: packages—sets of goods, possibly multiple units of each; a function that describes how much the agent values each package; pricelines—data structures in which to store the prices of each unit of each good; and bids—pairs of vectors corresponding to buy and sell offers.

Packages Let G denote an ordered set of n distinct goods and let $N \in \mathbb{N}^n$ represent the multiset of these goods in the marketplace, with N_g denoting the number of units of each good $g \in G$. A *package* M is a collection of goods, that is, a “submultiset” of N . We write $M \subseteq N$ whenever $M_g \leq N_g$ for all $g \in G$.

It is instructive to interpret this notation in the TAC domain. The flights, hotel rooms, and entertainment events up for auction in TAC comprise an ordered set of 28 distinct

goods. In principle, the multiset of goods in the TAC marketplace is:

$$N^{\text{TAC}} = \langle \underbrace{\infty, \dots, \infty}_{8 \text{ flights}}, \underbrace{16, \dots, 16}_{8 \text{ hotels}}, \underbrace{8, \dots, 8}_{12 \text{ events}} \rangle \in \mathbb{N}^{28}$$

In practice, however, since each agent works to satisfy the preferences of only eight clients, it suffices to consider the multiset of goods:

$$N^{\text{TAC}8} = \langle \underbrace{8, \dots, 8}_{8 \text{ flights}}, \underbrace{8, \dots, 8}_{8 \text{ hotels}}, \underbrace{8, \dots, 8}_{12 \text{ events}} \rangle \subseteq N^{\text{TAC}}$$

A trip corresponds to a package, specifically some $M \subseteq N^{\text{TAC}8}$ that satisfies the TAC feasibility constraints.

Given $A, B \subseteq N$, we rely on the two basic operations, \oplus and \ominus , defined as follows: for all $g \in G$,

$$\begin{aligned} (A \oplus B)_g &\equiv A_g + B_g \\ (A \ominus B)_g &\equiv A_g - B_g \end{aligned}$$

For example, if $G = \{\alpha, \beta, \gamma\}$ and $N = \langle 1, 2, 3 \rangle$, then $A = \langle 0, 1, 2 \rangle \subseteq N$ and $B = \langle 1, 1, 1 \rangle \subseteq N$. Moreover, $(A \oplus B)_\alpha = 1$, $(A \oplus B)_\beta = 2$, and $(A \oplus B)_\gamma = 3$; and $(A \ominus B)_\alpha = -1$, $(A \ominus B)_\beta = 0$, and $(A \ominus B)_\gamma = 1$.

Value Let \mathcal{N} denote the set of all submultisets of N : i.e., packages comprised of the goods in N . We denote $v : \mathcal{N} \rightarrow \mathbb{R}$ a function that describes the value the bidding agent attributes to each viable package.

In TAC, each agent's objective is to compile packages for $m = 8$ individual clients. As such, the agent's value function takes special form. Each client c is characterized by its own value function $v_c : \mathcal{N} \rightarrow \mathbb{R}$, and the agent's value for a collection of packages is the sum of its clients' respective values for those packages: given a vector of packages $\vec{X} = (X_1, \dots, X_m)$,

$$v(\vec{X}) = \sum_{c=1}^m v_c(X_c). \quad (7)$$

Pricelines A *buyer priceline* for good g is a vector $\vec{p}_g \in \mathbb{R}_+^{N_g}$, where the k th component, p_{gk} , stores the *marginal cost* to the agent of acquiring the k th unit of good g . For example, if an agent currently holds four units of a good \tilde{g} , and if four additional units of \tilde{g} are available at costs of \$25, \$40, \$65, and \$100, then the corresponding buyer priceline (a vector of length 8) is given by $\vec{p}_{\tilde{g}} = \langle 0, 0, 0, 0, 25, 40, 65, 100 \rangle$. The leading zeros indicate that the four goods the agent holds may be “acquired” at no cost.

We assume buyer pricelines are nondecreasing. Note that this assumption is WLOG, since an optimizing agent buys cheaper goods before more expensive ones.

Given a set of buyer pricelines $P = \{\vec{p}_g \mid g \in G\}$, we define costs additively, that is, the *cost* of the goods in multiset $Y \subseteq N$ is given by:

$$\begin{aligned} \forall g, \quad \text{Cost}_g(Y, P) &= \sum_{k=1}^{Y_g} p_{gk}, \\ \text{Cost}(Y, P) &= \sum_{g \in G} \text{Cost}_g(Y, P). \end{aligned} \quad (8)$$

A *seller priceline* for good g is a vector $\vec{\pi}_g \in \mathbb{R}_+^{N_g}$. Much like a buyer priceline, the k th component of a seller priceline for g stores the *marginal revenue* that an agent could earn from the k th unit it sells. For example, if the market demands four units of good \tilde{g} , which can be sold at prices of \$20, \$15, \$10, and \$5, then the corresponding seller priceline is given by $\vec{\pi}_{\tilde{g}} = \langle 20, 15, 10, 5, 0, 0, 0, 0, 0 \rangle$. Analogously to buyer pricelines, the tail of zero revenues indicates that the market demands only four of those units.

We assume seller pricelines are nonincreasing. Note that this assumption is WLOG, since an optimizing agent sells more expensive goods before cheaper ones.

Given a set of seller pricelines $\Pi = \{\vec{\pi}_g \mid g \in G\}$, we define revenue additively, that is, the *revenue* associated with multiset $Z \subseteq N$ is given by:

$$\forall g, \quad \text{Revenue}_g(Z, \Pi) = \sum_{k=1}^{Z_g} \pi_{gk}, \quad (9)$$

$$\text{Revenue}(Z, \Pi) = \sum_{g \in G} \text{Revenue}_g(Z, \Pi). \quad (10)$$

If a priceline is constant, we say that prices are *linear*. We refer to the constant value as a *unit price*. With linear prices, the cost of acquiring k units of good g is k times the unit price of good g .

Bids An agent submits a bid β expressing offers to buy or sell various units of the goods in the marketplace. We divide β into two components $\langle \vec{b}, \vec{a} \rangle$, where for each good g the bid consists of a *buy offer*, $\vec{b}_g = \langle b_{g1}, \dots, b_{gN_g} \rangle$, and a *sell offer*, $\vec{a}_g = \langle a_{g1}, \dots, a_{gN_g} \rangle$. The bid price $b_{gk} \in \mathbb{R}_+$ (resp. $a_{gk} \in \mathbb{R}_+$) represents an offer to buy (sell) the k th unit of good g at that price.

By definition, the agent cannot buy (sell) the k th unit unless it also buys (sells) units $1, \dots, k-1$. To accommodate this fact, we impose the following constraint: Buy offers must be nonincreasing in k , and sell offers nondecreasing. In addition, an agent may not offer to sell a good for less than the price at which it is willing to buy that good: i.e., $b_{g1} < a_{g1}$. Otherwise, it would simultaneously buy and sell good g . We refer to these restrictions as *bid monotonicity* constraints.

5.1.2 PSEUDO-AUCTION RULES

Equipped with this formalism, we can specify the rules that govern pseudo-auctions. As in a true auction, the outcome of a pseudo-auction dictates the quantity of each good to exchange, and at what prices, conditional on the agent's bid. The quantity issue is resolved by the *winner determination rule* whereas the price issue is resolved by the *payment rule*.

Definition 5.1 [Pseudo-Auction Winner Determination Rule] Given buyer and seller price-lines P and Π , and bid $\beta = \langle \vec{b}, \vec{a} \rangle$, the agent buys the multiset of goods $\text{Buy}(\beta, P)$ and sells the multiset of goods $\text{Sell}(\beta, \Pi)$, where

$$\text{Buy}_g(\beta, P) = \max_k k \text{ such that } b_{gk} \geq p_{gk}$$

$$\text{Sell}_g(\beta, \Pi) = \max_k k \text{ such that } a_{gk} \leq \pi_{gk}$$

Note that the monotonicity restrictions on bids ensure that the agent’s offer is better than or equal to the price for every unit it exchanges, and that the agent does not simultaneously buy and sell any good.

There are at least two alternative payment rules an agent may face. In a *first-price pseudo-auction*, the agent pays its bid price (for buy offers, or is paid its bid price for sell offers) for each good it wins. In a *second-price pseudo-auction*, the agent pays (or is paid) the prevailing prices, as specified by the realized buyer and seller pricelines. This terminology derives by analogy from the standard first- and second-price sealed bid auctions (Krishna, 2002; Vickrey, 1961). In these mechanisms, the high bidder for a single item pays its bid (the first price), or the highest losing bid (the second price), respectively. The salient property is that in first-price pseudo-auctions, the price is set by the bid of the winner, whereas in second-price pseudo-auctions an agent’s bid price determines whether or not it wins but not the price it pays.

In this paper, we focus on the second-price model. That is, our basic problem definitions presume second-price auctions; however, our bidding heuristics are not tailored to this case. As in true auctions, adopting the second-price model in pseudo-auctions simplifies the problem for the bidder. It also provides a reasonable approximation to the situation faced by TAC agents, as we now argue:

- In TAC entertainment auctions, agents submit bids (i.e., buy and sell offers) of the form specified above. If we interpret an agent’s buyer and seller pricelines as the current order book (not including the agent’s own bid), then the agent’s immediate winnings are as determined by the winner determination rule, and payments are according to the second-price rule (i.e., the order-book prices prevail).
- In TAC hotel auctions, only buy bids are allowed. Assuming once again an order book that reflects all outstanding bids other than the agent’s own, an accurate buyer priceline would indicate that the agent can win k units of a good if it pays—for all k units—a price just above the $(17 - k)$ th existing (other-agent) offer. The actual price it pays will be that of the 16th-highest unit offer (including its own offer). Since the agent’s own bid may affect the price,⁷ this situation lies between the first- and second-price characterizations of pseudo-auctions described above.
- In TAC flight auctions, agents may buy any number of units at the posted price. The situation at any given time is modeled exactly by the second-price pseudo-auction abstraction.

5.2 Bidding Problems

We are now ready to discuss the optimization module repeatedly employed by RoxyBot-06 within its bidding cycle to construct its bids. The key bidding decisions are: what goods to bid on, at what price, and when?

7. It can do so in two ways. First, the agent may submit the 16th-highest unit offer, in which case it sets the price. Second, when it bids for multiple units, the number it wins determines the price-setting unit, thus affecting the price for all winning units. Note that this second effect would be present even if the auction cleared at the 17th-highest price.

Although RoxyBot technically faces an n -stage stochastic optimization problem, it solves this problem by collapsing those n stages into only two relevant stages, “current” and “future,” necessitating only one stochastic model of future prices (current prices are known). This simplification is achieved by ignoring the potentially useful information that hotel auctions close one by one in a random, unspecified order, and instead operating (like most TAC agents) under the assumption that all hotel auctions close at the end of the current stage. Hence, there is only one model of hotel prices: a stochastic model of future prices. Moreover, the only pressing decisions regarding hotels are what goods to bid on now and at what price. There is no need to reason about the timing of hotel bid placement.

In contrast, since flight and entertainment auctions clear continuously, a trading agent should reason about the relevant tradeoffs in timing its placement of bids on these goods. Still, under the assumption that all hotel auctions close at the end of the current stage, in future stages, hotel prices, and hence hotel winnings, are known, so the only remaining decisions are what flight and entertainment tickets to buy. A rational agent will time its bids in these markets to capitalize on the “best” prices. (The best prices are the minima for buying and the maxima for selling.) Hence, it suffices for an agent’s model of future prices in these markets to predict only the best prices (conditioned on current prices). That is, it suffices to consider only one stochastic pricing model. No further information is necessary.

Having established that it suffices for RoxyBot to pose and solve a two-stage, rather than an n -stage, stochastic optimization problem, we now proceed to define an abstract series of such problems that is designed to capture the essence of bidding under uncertainty in TAC-like hybrid markets that incorporate aspects of simultaneous and sequential, one-shot and continuously-clearing, auctions. More specifically, we formulate these problems as two-stage stochastic programs with integer recourse (see the book by Birge & Louveaux, 1997, for an introduction to stochastic programming).

In a two-stage stochastic program, there are two decision-making stages, and hence two sets of variables: first- and second-stage variables. The objective is to maximize the sum of the first-stage objectives (which depend only on the first-stage variables) and the expected value of the ensuing second-stage objectives (which can depend on both the first- and second-stage variables). The objective value in the second stage is called the *recourse* value, and if any of the second-stage variables are integer-valued, then the stochastic program is said to have *integer* recourse.

At a high-level, the bidding problem can be formulated as a two-stage stochastic program as follows: in the first stage, when current prices are known but future prices are uncertain, bids are selected; in the second stage, all uncertainty is resolved, and goods are exchanged. The objective is to maximize the expected value of the second-stage objective, namely the sum of the inherent value of final holdings and any profits earned, less any first-stage costs. Since the second stage involves integer-valued decisions (the bidder decides what goods to buy and sell at known prices), the bidding problem is one with integer recourse.

In this section, we formulate a series of bidding problems as two-stage stochastic programs with integer recourse, each one tailored to a different type of auction mechanism, illustrating a different type of bidding decision. The mechanisms we study, inspired by TAC, are one-shot and continuously-clearing variants of second-price pseudo-auctions. In the former, bids can only be placed in the first stage; in the latter, there is an opportunity

for recourse. Ultimately, we combine all decision problems into one unified problem that captures what we mean by bidding under uncertainty.

In our formal problem statements, we rely on the following notation:

- Variables:

- Q^1 is a multiset of goods to buy now
- Q^2 is a multiset of goods to buy later
- R^1 is a multiset of goods to sell now
- R^2 is a multiset of goods to sell later

- Constants:

- P^1 is a set of current buyer pricelines
- P^2 is a set of future buyer pricelines
- Π^1 is a set of current seller pricelines
- Π^2 is a set of future seller pricelines

Note that P^1 and Π^1 are always known, whereas P^2 and Π^2 are uncertain in the first stage but their uncertainty is resolved in the second stage.

Flight Bidding Problem An agent’s task in bidding in flight auctions is to decide how many flights to buy now at current prices and later at the lowest future prices, given (known) current prices and a stochastic model of future prices. Although in TAC all units of each flight sell for the same price at any one time, we state the flight bidding problem more generally: we allow for different prices for different units of the same flight.

Definition 5.2 [Continuously-Clearing, Buying] Given a set of current buyer pricelines P^1 and a probability distribution f over future buyer pricelines P^2 ,

$$\text{FLT}(f) = \max_{Q^1 \in \mathbb{Z}^n} \mathbb{E}_{P^2 \sim f} \left[\max_{Q^2 \in \mathbb{Z}^n} v(Q^1 \oplus Q^2) - (\text{Cost}(Q^1, P^1) + \text{Cost}(Q^1 \oplus Q^2, P^2) - \text{Cost}(Q^1, P^2)) \right] \quad (11)$$

Note that there are two cost terms referring to future pricelines ($\text{Cost}(\cdot, P^2)$). The first of these terms adds the total cost of the goods bought in the first and second stages. The second term subtracts the cost of the goods bought in just the first stage. This construction ensures that, if an agent buys k units of a good now, any later purchases of that good incur the charges of units $(k+1, k+2, \dots)$ in the good’s future priceline.

Entertainment Bidding Problem Abstractly, the entertainment *buying* problem is the same as the flight bidding problem. An agent must decide how many entertainment tickets to buy now at current prices and later at the lowest future prices. The entertainment *selling* problem is the opposite of this buying problem. An agent must decide how many tickets to sell now at current prices and later at the highest future prices.

Definition 5.3 [Continuously-Clearing, Buying and Selling] Given a set of current buyer and seller pricelines $(P, \Pi)^1$ and a probability distribution f over future buyer and seller pricelines $(P, \Pi)^2$,

$$\begin{aligned} \text{ENT}(f) = \max_{Q^1, R^1 \in \mathbb{Z}^n} & \mathbb{E}_{(P, \Pi)^2 \sim f} \left[\max_{Q^2, R^2 \in \mathbb{Z}^n} v((Q^1 \oplus Q^2) \ominus (R^1 \oplus R^2)) \right. \\ & - (\text{Cost}(Q^1, P^1) + \text{Cost}(Q^1 \oplus Q^2, P^2) - \text{Cost}(Q^1, P^2)) \\ & \left. + (\text{Revenue}(R^1, \Pi^1) + \text{Revenue}(R^1 \oplus R^2, \Pi^2) - \text{Revenue}(R^1, \Pi^2)) \right] \quad (12) \end{aligned}$$

subject to $Q^1 \supseteq R^1$ and $Q^1 \oplus Q^2 \supseteq R^1 \oplus R^2$, for all $(P, \Pi)^2$.

The constraints ensure that an agent does not sell more units of any good than it buys.

Hotel Bidding Problem Hotel auctions close at fixed times, but in an unknown order. Hence, during each iteration of an agent's bidding cycle, one-shot auctions approximate these auctions well. Unlike in the continuous setup, where decisions are made in both the first and second stages, in the one-shot setup, bids can only be placed in the first stage; in the second stage, winnings are determined and evaluated.

Definition 5.4 [One-Shot, Buying] Given a probability distribution f over future buyer pricelines P^2 ,

$$\text{HOT}(f) = \max_{\beta^1 = \langle \vec{b}, 0 \rangle} \mathbb{E}_{P^2 \sim f} [v(\text{Buy}(\beta^1, P^2)) - \text{Cost}(\text{Buy}(\beta^1, P^2), P^2)] \quad (13)$$

Hotel Bidding Problem, with Selling Although it is not possible for agents to sell TAC hotel auctions, one could imagine an analogous auction setup in which it were possible to sell goods as well as buy them.

Definition 5.5 [One-Shot, Buying and Selling] Given a probability distribution f over future buyer and seller pricelines $(P, \Pi)^2$,

$$\max_{\beta^1 = \langle \vec{b}, \vec{a} \rangle} \mathbb{E}_{(P, \Pi)^2 \sim f} [v(\text{Buy}(\beta^1, P^2) \ominus \text{Sell}(\beta^1, \Pi^2)) - \text{Cost}(\text{Buy}(\beta^1, P^2), P^2) + \text{Revenue}(\text{Sell}(\beta^1, \Pi^2), \Pi^2)] \quad (14)$$

subject to $\text{Buy}(\beta^1, P^2) \geq \text{Sell}(\beta^1, \Pi^2)$, for all $(P, \Pi)^2$.

Bidding Problem Finally, we present (a slight generalization of) the TAC bidding problem by combining the four previous stochastic optimization problems into one. This abstract problem models bidding to buy and sell goods both via continuously-clearing and one-shot second-price pseudo-auctions, as follows:

Definition 5.6 [Bidding Under Uncertainty] Given a set of current buyer and seller price-lines $(P, \Pi)^1$ and a probability distribution f over future buyer and seller pricelines $(P, \Pi)^2$,

$$\begin{aligned} \text{BID}(f) = \max_{Q^1, R^1 \in \mathbb{Z}^n, \beta^1 = \langle \vec{b}, \vec{a} \rangle} & \mathbb{E}_{(P, \Pi)^2 \sim f} \left[\max_{Q^2, R^2 \in \mathbb{Z}^n} v((Q^1 \oplus Q^2) \ominus (R^1 \oplus R^2) \oplus \text{Buy}(\beta^1, P^2) \ominus \text{Sell}(\beta^1, P^2)) \right. \\ & - (\text{Cost}(Q^1, P^1) + \text{Cost}(Q^1 \oplus Q^2, P^2) - \text{Cost}(Q^1, P^2) + \text{Cost}(\text{Buy}(\beta^1, P^2), P^2)) \\ & \left. + (\text{Revenue}(R^1, \Pi^1) + \text{Revenue}(R^1 \oplus R^2, \Pi^2) - \text{Revenue}(R^1, \Pi^2) + \text{Revenue}(\text{Sell}(\beta^1, \Pi^2), \Pi^2)) \right] \quad (15) \end{aligned}$$

subject to $Q^1 \supseteq R^1$ and $Q^1 \oplus Q^2 \supseteq R^1 \oplus R^2$ and $\text{Buy}(\beta^1, P^2) \geq \text{Sell}(\beta^1, \Pi^2)$, for all $(P, \Pi)^2$.

Once again, this bidding problem is (i) stochastic: it takes as input a stochastic model of future prices; (ii) global: it seamlessly integrates flight, hotel, and entertainment bidding decisions; and (iii) dynamic: it facilitates simultaneous reasoning about current and future stages of the game.

Next, we describe various heuristic approaches to solving the problem of bidding under uncertainty.

5.3 Bidding Heuristics

In this section, we discuss two heuristic solutions to the bidding problem: specifically, the expected value method (EVM), an approach that collapses stochastic information, and sample average approximation (SAA), an approach that exploits stochastic information and characterizes RoxyBot-06.

5.3.1 EXPECTED VALUE METHOD

The *expected value method* (Birge & Louveaux, 1997) is a standard way of approximating the solution to a stochastic optimization problem. First, the given distribution is collapsed into a point estimate (e.g., the mean); then, a solution to the corresponding deterministic optimization problem is output as an approximate solution to the original stochastic optimization problem. Applying this idea to the problem of bidding under uncertainty yields:

Definition 5.7 [Expected Value Method] Given a probability distribution f over buyer and seller pricelines, with expected values \bar{P}^2 and $\bar{\Pi}^2$, respectively,

$$\begin{aligned} \text{BID_EVM}(\bar{P}^2, \bar{\Pi}^2) = & \\ & \max_{Q^1, R^1 \in \mathbb{Z}^n, \beta^1 = \langle \vec{b}, \vec{a} \rangle, Q^2, R^2 \in \mathbb{Z}^n} v((Q^1 \oplus Q^2) \ominus (R^1 \oplus R^2) \oplus (\text{Buy}(\beta^1, \bar{P}^2) \ominus \text{Sell}(\beta^1, \bar{P}^2)) \\ & - (\text{Cost}(Q^1, P^1) + \text{Cost}(Q^1 \oplus Q^2, \bar{P}^2) - \text{Cost}(Q^1, \bar{P}^2) + \text{Cost}(\text{Buy}(\beta^1, \bar{P}^2), \bar{P}^2)) \\ & + (\text{Revenue}(R^1, \Pi^1) + \text{Revenue}(R^1 \oplus R^2, \bar{\Pi}^2) - \text{Revenue}(R^1, \bar{\Pi}^2) + \text{Revenue}(\text{Sell}(\beta^1, \bar{\Pi}^2), \bar{\Pi}^2)) \end{aligned} \quad (16)$$

subject to $Q^1 \supseteq R^1$ and $Q^1 \oplus Q^2 \supseteq R^1 \oplus R^2$.

In practice, without full knowledge of the distribution f , we cannot implement the expected value method; in particular, we cannot compute \bar{P}^2 or $\bar{\Pi}^2$ so we cannot solve $\text{BID_EVM}(\bar{P}^2, \bar{\Pi}^2)$ exactly. We can, however, solve an approximation of this problem in which the expected buyer and seller pricelines \bar{P}^2 and $\bar{\Pi}^2$ are replaced by an average scenario $(\hat{P}^2, \hat{\Pi}^2)$ (i.e., average buyer and seller pricelines), defined as follows:

$$\hat{P}^2 = \frac{1}{S} \sum_{i=1}^S P_i^2, \quad \hat{\Pi}^2 = \frac{1}{S} \sum_{i=1}^S \Pi_i^2.$$

Algorithm 5 EVM(G, N, f, S)

- 1: sample S scenarios $(P, \Pi)_1^2, \dots, (P, \Pi)_S^2 \sim f$
 - 2: $\beta \Leftarrow \text{BID_EVM} \left(\sum_{i=1}^S P_i^2, \sum_{i=1}^S \Pi_i^2 \right)$
 - 3: **return** β
-

5.3.2 SAMPLE AVERAGE APPROXIMATION

Like the expected value method, *sample average approximation* is an intuitive way of approximating the solution to a stochastic optimization problem. The idea is simple: (i) generate a set of sample scenarios, and (ii) solve an approximation of the problem that incorporates only the sample scenarios. Applying the SAA heuristic (see Algorithm 6) involves solving the following approximation of the bidding problem:

Definition 5.8 [Sample Average Approximation] Given a set of S scenarios, $(P, \Pi)_1^2, \dots, (P, \Pi)_S^2 \sim f$,

$$\begin{aligned} \text{BID_SAA}((P, \Pi)_1^2, \dots, (P, \Pi)_S^2) = \\ \max_{Q^1, R^1 \in \mathbb{Z}^n, \beta^1 = \langle \vec{b}, \vec{a} \rangle} \sum_{i=1}^S \max_{Q^2, R^2 \in \mathbb{Z}^n} v((Q^1 \oplus Q^2) \ominus (R^1 \oplus R^2) \oplus (\text{Buy}(\beta^1, P_i^2) \ominus \text{Sell}(\beta^1, P_i^2))) \\ - (\text{Cost}(Q^1, P^1) + \text{Cost}(Q^1 \oplus Q^2, P_i^2) - \text{Cost}(Q^1, P_i^2) + \text{Cost}(\text{Buy}(\beta^1, P_i^2), P_i^2)) \\ + (\text{Revenue}(R^1, \Pi^1) + \text{Revenue}(R^1 \oplus R^2, \Pi_i^2) - \text{Revenue}(R^1, \Pi_i^2) + \text{Revenue}(\text{Sell}(\beta^1, \Pi_i^2), \Pi_i^2)) \end{aligned} \quad (17)$$

subject to $Q^1 \supseteq R^1$ and $Q^1 \oplus Q^2 \supseteq R^1 \oplus R^2$.

Algorithm 6 SAA(G, N, f, S)

- 1: sample S scenarios $(P, \Pi)_1^2, \dots, (P, \Pi)_S^2 \sim f$
 - 2: $\beta \Leftarrow \text{BID_SAA}((P, \Pi)_1^2, \dots, (P, \Pi)_S^2)$
 - 3: **return** β
-

Using the theory of large deviations, Ahmed and Shapiro (2002) establish the following result: as $S \rightarrow \infty$, the probability that an optimal solution to the sample average approximation of a stochastic program with integer recourse is an optimal solution to the original stochastic optimization problem approaches 1 exponentially fast. Given hard time and space constraints, however, it is not always possible to sample sufficiently many scenarios to infer any reasonable guarantees about the quality of a solution to a sample average approximation. Hence, we propose a modified SAA heuristic, in which SAA is fed some tailor-made “important” scenarios, and we apply this idea to the bidding problem.

5.3.3 MODIFIED SAMPLE AVERAGE APPROXIMATION

The bids that SAA places are sample prices that appear in its scenarios. SAA never bids higher on any good than its highest sampled price, because as far as it knows, bidding that price is enough to win that good in all scenarios. However, there is some chance that the

highest sampled price falls below the clearing price. Let us compute this probability in the case of a single-unit auction, or a *uniform-price* multi-unit auction: i.e., one in which all units of the good being auctioned off clear at the same price.

Let F denote the cumulative distribution function over the predicted prices, let f denote the corresponding density function, and let G denote the cumulative distribution function over the clearing prices. Using this notation, the term $1 - G(x)$ is the probability the clearing price is greater than x . Further, let X be a random variable that represents the highest value among S sample price predictions. Then $P(X \leq x) = F(x)^S$ is the probability that all S samples (and hence the highest among them) are less than x ; and $P(X = x) = (F(x)^S)' = S(F(x))^{S-1}f(x)$ is the probability that the highest value among the S samples equals x . Putting these two terms together—namely, the probability the highest sample price prediction is exactly x , and the probability the clearing price is greater than x —we can express the probability the highest of SAA's sample price predictions is less than the clearing price as follows:

$$\int_{-\infty}^{\infty} S(F(x))^{S-1}f(x)(1 - G(x))dx \quad (18)$$

Assuming perfect prediction (so that $G = F$), this complex expression simplifies as follows:

$$\begin{aligned} & \int_{-\infty}^{\infty} S(F(x))^{S-1}f(x)(1 - F(x))dx \\ &= S \int_{-\infty}^{\infty} (F(x))^{S-1}f(x)dx - S \int_{-\infty}^{\infty} (F(x))^S f(x)dx \\ &= S \left[\frac{(F(x))^S}{S} \right]_{-\infty}^{\infty} - S \left[\frac{(F(x))^{S+1}}{S+1} \right]_{-\infty}^{\infty} \\ &= \frac{1}{S+1} \end{aligned}$$

Hence, the probability that all SAA's sample price predictions are less than the clearing price is $1/(S+1)$. In particular, assuming perfect prediction and that the clearing prices in the TAC hotel auctions are independent, the probability that an SAA agent with 49 scenarios bidding in TAC Travel has any chance of winning all eight hotels (i.e., the probability that a sample price in at least one of its scenarios is greater than the clearing price) is only $\left(1 - \frac{1}{49+1}\right)^8 = 0.98^8 \approx 0.85$.

To remedy this situation, we designed and implemented a simple variant of SAA in RoxyBot-06. The SAA* heuristic (see Algorithm 7) is a close cousin of SAA, the only difference arising in their respective scenario sets. Whereas SAA samples S scenarios, SAA* samples only $S - |N|$ scenarios, where $|N| = \sum_g N_g$. SAA* creates an additional $|N|$ scenarios as follows: for each unit k of each good $g \in G$, it sets the price of the k th unit of good g to the upper limit of its range of possible prices and, after conditioning on this price setting, it sets the prices of the other goods to their mean values. Next, we describe experiments with a test suite of bidding heuristics, including SAA and SAA*, in a controlled testing environment.

Algorithm 7 SAA*(G, N, f, S)**Require:** $S \geq |N|$

- 1: hard-code $|N|$ scenarios $(P, \Pi)_1^2, \dots, (P, \Pi)_{|N|}^2$
- 2: sample $S - |N|$ scenarios $(P, \Pi)_{|N|+1}^2, \dots, (P, \Pi)_S^2 \sim f$
- 3: $\beta \Leftarrow \text{BID_SAA}((P, \Pi)_1^2, \dots, (P, \Pi)_S^2)$
- 4: **return** β

Agent	Predictions	Bids	On
SMU	Average scenario	Marginal utilities	All goods
AMU	S scenarios	Calculates marginal utilities in each scenario Bids average marginal utilities across scenarios	All goods
TMU	Average scenario	Marginal utilities	Goods in a target set
BE	S scenarios	Best of S TMU solutions	Goods in a target set
TMU*	Average scenario	Marginal utilities, assuming only goods in a target set are available	Goods in a target set
BE*	S scenarios	Best of S TMU* solutions	Goods in a target set

Table 3: Marginal-utility-based agents. The marginal utility of a good is defined as the incremental utility that can be achieved by winning that good, relative to the utility of the set of goods already held.

5.4 Summary

In this section, we developed a series of bidding problems, and heuristics solutions to those problems, that captures the essence of bidding in the one-shot and continuously-clearing auctions that characterize TAC. The bulk of our presentation was deliberately abstract, so as to suggest that our problems and their solutions are applicable well beyond the realm of TAC: e.g., to bidding for interdependent goods in separate eBay auctions. Still, it remains to validate our approach in other application domains.

6. Experiments

We close this paper with two sets of experimental results, the first in a controlled testing environment, and the second the results from the final round of the 2006 TAC Travel competition. The combined strategy of hotel price prediction via SimAA and bid optimization via SAA emerged victorious in both settings.

6.1 Controlled Experiments

To some extent at least, our approach to bidding has been validated by the success of RoxyBot-06 in TAC-06. Nonetheless, we ran simulations in a controlled testing environment to further validate our approach. These results are reported by Lee (2007) and Greenwald et al. (2008), but we summarize them here as well.

We built a test suite of agents, all of which predict using RoxyBot-06’s SimAA random mechanism with distribution. The agents differ in their bidding strategies; the possibilities include SAA,⁸ SAA*, and the six marginal-utility-based heuristics studied by Wellman et al. (2007), and summarized in Table 3.

Our experiments were conducted in a TAC Travel-like setting, modified to remove any aspects of the game that would obscure a controlled study of bidding. Specifically, we eliminated flight and entertainment trading, and endowed all agents with eight flights in and eight flights out on each day. Further, we assumed all hotels closed after one round of bidding (i.e., hotel auctions are one-shot, so that the ensuing bid optimization problem adheres to Definition 5.4).

We designed two sets of experiments: one decision-theoretic and one game-theoretic. In the former, hotel clearing prices are the outcome of a simulation of simultaneous ascending auctions, but depend on the actual clients in each game, not a random sampling. (Our simulator is more informed than the individual agents.) In the latter, hotel clearing prices are determined by the bids the agents submit using the same mechanism as in TAC Travel: the clearing price is the 16th highest bid (or zero, if fewer than 16 bids are submitted).

We first ran experiments with 8 agents per game, but found that hotel prices were often zero: i.e., there was insufficient competition. We then changed the setup to include a random number of agents drawn from a binomial distribution with $n = 32$ and $p = 0.5$, with the requisite number of agents sampled uniformly with replacement from the set of possible agents. The agents first sample the number of competitors from the binomial distribution, and then generate scenarios assuming the sampled number of competitors.

Because of the game-theoretic nature of TAC, an individual agent’s performance can depend heavily on the other agents included in the agent pool. In our experiments, we attempted to mitigate any artificial effects of the specific agents we chose to include in our pool by sampling agents from the pool to play each game, with replacement. Thus, an agent’s average score from the games is a measure of the agent’s performance against various combinations of opponents.

In Figures 3(a) and 3(b), we plot the mean scores obtained by each agent type in each setting, along with 95% confidence intervals. These averages were computed based on 1000 independent observations, obtained by playing 1000 games. Scores were averaged across agent types in each game to account for any game dependencies. SAAB and SAAT⁹ are the best performing agents in the game-theoretic experiments and among the best in the decision-theoretic setting.

6.2 TAC 2006 Competition Results

Table 4 lists the agents entered in TAC-06 and Table 5 summarizes the outcome. The TAC-06 finals comprised 165 games over three days, with the 80 games on the last day weighted 1.5 times as much as the 85 over the first two days. On the first day of the finals, RoxyBot finished third, behind Mertacor and Walverine—the top scorers in 2005. As it happens, RoxyBot’s optimization routine, which was designed for stochastic hotel and entertainment

8. The particular implementation details explaining how RoxyBot-06 applied SAA in the TAC domain are relegated to Appendix A.

9. SAAB is SAA, and SAAT is a slight variant of SAA*. See the paper by Greenwald et al. (2008) for details.

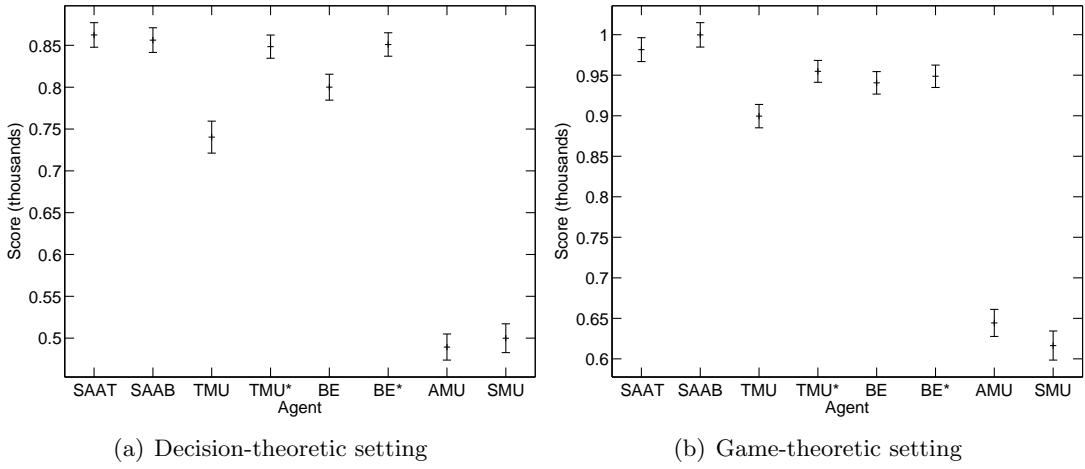


Figure 3: Mean scores and confidence intervals.

price predictions, was accidentally fed deterministic predictions (i.e., point price estimates) for entertainment. Moreover, these predictions were fixed, rather than adapted based on recent game history.

On days 2 and 3, RoxyBot ran properly, basing its bidding in all auctions on stochastic information. Moreover, the agent was upgraded after day 1 to bid on flights not just once, but twice, during each minute. This enabled the agent to delay its bidding somewhat at the end of a game for flights whose prices are decreasing. No doubt this minor modification enabled RoxyBot to emerge victorious in 2006, edging out Walverine by a whisker, below the integer precision reported in Table 5. The actual margin was 0.22—a mere 22 parts in 400,000. Adjusting for control variates (Ross, 2002) spreads the top two finishers a bit further.¹⁰

Agent	Affiliation	Reference
006	Swedish Inst Comp Sci	Aurell et al., 2002
kin_agent	U Macau	
L-Agent	Carnegie Mellon U	Sardinha et al., 2005
Mertacor	Aristotle U Thessaloniki	Toulis et al., 2006; Kehagias et al., 2006
RoxyBot	Brown U	Greenwald et al., 2003, 2004, 2005; Lee et al., 2007
UTTA	U Tehran	
Walverine	U Michigan	Cheng et al., 2005; Wellman et al., 2005
WhiteDolphin	U Southampton	He & Jennings, 2002; Vetsikas & Selman, 2002

Table 4: TAC-06 participants.

10. Kevin Lochner computed these adjustment factors using the method described by Wellman et al. (2007, ch. 8).

Agent	Finals	Adjustment Factor
RoxyBot	4032	-5
Walverine	4032	-17
WhiteDolphin	3936	-2
006	3902	-27
Mertacor	3880	-16
L-Agent	3860	7
kin_agent	3725	0
UTTA	2680	-14

Table 5: TAC-06 final scores, with adjustment factors based on control variates.

Mean scores, utilities, and costs (with 95% confidence intervals) for the last day of the TAC-06 finals (80 games) are plotted in Figure 4 and detailed statistics are tabulated in Table 6. There is no single metric such as low hotel or flight costs that is responsible for RoxyBot’s success. Rather its success derives from the right balance of contradictory goals. In particular, RoxyBot incurs high hotel and mid-range flight costs while achieving mid-range trip penalty and high event profit.¹¹

Let us compare RoxyBot with two closest rivals: Walverine and WhiteDolphin. Comparing to Walverine first, Walverine bids lower prices (by 55) on fewer hotels (49 less), yet wins more (0.8) and wastes less (0.42). It would appear that Walverine’s hotel bidding strategy outperforms RoxyBot’s, except that RoxyBot earns a higher hotel bonus (15 more). RoxyBot also gains an advantage by spending 40 less on flights and earning 24 more in total entertainment profit.

A very different competition takes place between RoxyBot and WhiteDolphin. WhiteDolphin bids lower prices (120 less) on more hotels (by 52) than RoxyBot. RoxyBot spends much more (220) on hotels than WhiteDolphin but makes up for it by earning a higher hotel bonus (by 96) and a lower trip penalty (by 153). It seems that WhiteDolphin’s strategy is to minimize costs even if that means sacrificing utility.

6.3 Summary

As already noted, TAC Travel bidding, viewed as an optimization problem, is an n -stage decision problem. We solve this n -stage decision problem as a sequence of 2-stage decision problems. The controlled experiments reported in this section establish that our bidding strategy, SAA, is the best in our test suite in the setting for which it was designed, with only 2 stages. The TAC competition results establish that this strategy is also effective in an n -stage setting.

7. Collective Behavior

The hotel price prediction techniques described in Section 4.2 are designed to compute (or at least approximate) competitive equilibrium prices without full knowledge of the client pop-

11. An agent suffers trip penalties to the extent that it assigns its clients packages that differ from their preferred.

	Rox	Wal	Whi	SIC	Mer	L-A	kin	UTT
# of Hotel Bids	130	81	182	33	94	58	15	24
Average of Hotel Bids	170	115	50	513	147	88	356	498
# of Hotels Won	15.99	16.79	23.21	13.68	18.44	14.89	15.05	9.39
Hotel Costs	1102	1065	882	1031	902	987	1185	786
# of Unused Hotels	2.24	1.82	9.48	0.49	4.86	1.89	0.00	0.48
Hotel Bonus	613	598	517	617	590	592	601	424
Trip Penalty	296	281	449	340	380	388	145	213
Flight Costs	4615	4655	4592	4729	4834	4525	4867	3199
Event Profits	110	26	6	-6	123	-93	-162	-4
Event Bonus	1470	1530	1529	1498	1369	1399	1619	996
Total Event Profits	1580	1556	1535	1492	1492	1306	1457	992
Average Utility	9787	9847	9597	9775	9579	9604	10075	6607
Average Cost	5608	5693	5468	5765	5628	5605	6213	3989
Average Score	4179	4154	4130	4010	3951	3999	3862	2618

Table 6: 2006 Finals, Last day. Tabulated Statistics. We omit the first two days because agents can vary across days, but cannot vary within. Presumably, the entries on the last day are the teams' preferred versions of the agents.

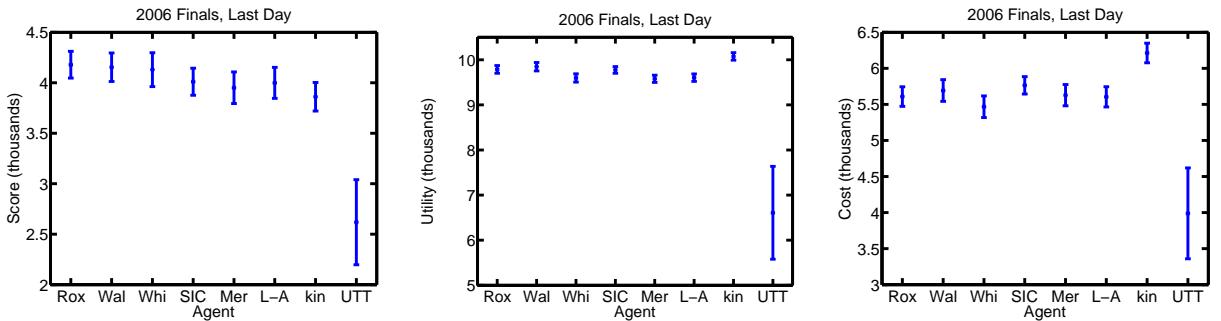


Figure 4: 2006 Finals, Last day. Mean scores, utilities, and costs, and 95% confidence intervals.

ulation. In this section, we assume this knowledge and view the output of the tâtonnement and SimAA calculations not as predictions but as ground truth. We compare the actual prices in the final games to this ground truth in respective years since 2002 to determine whether TAC market prices resemble CE prices. What we find is depicted in Figure 5. Because of the nature of our methods, these calculations pertain to hotel prices only.

The results are highly correlated on both metrics (Euclidean distance and EVPP). We observe that the accuracy of CE price calculations has varied from year to year. 2003 was the year in which TAC Supply Chain Management (SCM) was introduced. Many participants diverted their attention away from Travel towards SCM that year, perhaps leading to degraded performance in Travel. Things seem to improve in 2004 and 2005. We

cannot explain the setback in 2006, except by noting that performance is highly dependent on the particular agent pool, and in 2006 there were fewer agents in that pool.

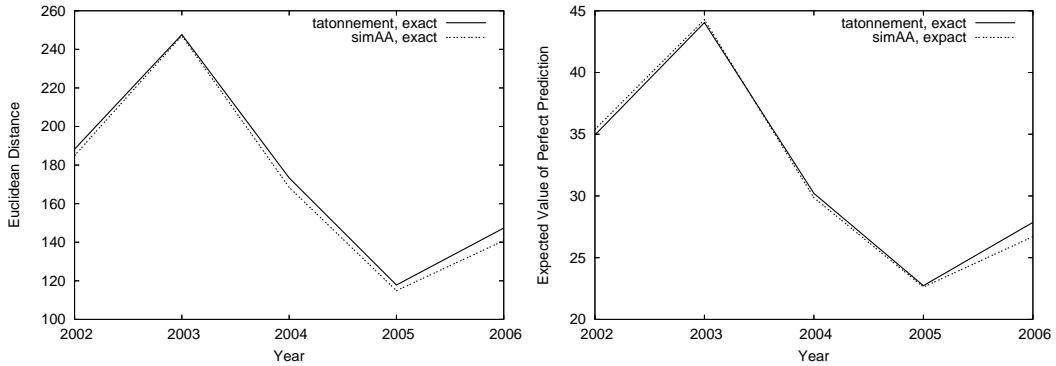


Figure 5: A comparison of the actual (hotel) prices to the output of competitive equilibrium price calculations in the final games since 2002. The label “exact” means: full knowledge of the client population.

8. Conclusion

The foremost aim of trading agent research is to develop a body of techniques for effective design and analysis of trading agents. Contributions to trading agent design include the invention of trading strategies, together with models and algorithms for realizing their computation and methods to measure and evaluate the performance of agents characterized by those strategies. Researchers seek both specific solutions to particular trading problems and general principles to guide the development of trading agents across market scenarios. This paper purports to contribute to this research agenda. We described the design and implementation of RoxyBot-06, an able trading agent as demonstrated by its performance in TAC-06.

Although automated trading in electronic markets has not yet fully taken hold, the trend is well underway. Through TAC, the trading agent community is demonstrating the potential for autonomous bidders to make pivotal trading decisions in a most effective way. Such agents offer the potential to accelerate the automation of trading more broadly, and thus shape the future of commerce.

Acknowledgments

This paper extends the work of Lee et al. (2007). The material in Section 5.1 is based on the book by Wellman et al. (2007). We are grateful to several anonymous reviewers whose constructive criticisms enhanced the quality of this work. This research was supported by NSF Career Grant #IIS-0133689.

Appendix A. TAC Bidding Problem: SAA

The problem of bidding in the simultaneous auctions that characterize TAC can be formulated as a two-stage stochastic program. In this appendix, we present the implementation details of the integer linear program (ILP) encoded in RoxyBot-06 that approximates an optimal solution to this stochastic program.¹²

We formulate this ILP assuming current prices are known, and future prices are uncertain in the first stage but revealed in the second stage. Note that whenever prices are known, it suffices for an agent to make decisions about the quantity of each good to buy, rather than about bid amounts, since choosing to bid an amount that is greater than or equal to the price of a good is equivalent to a decision to buy that good.

Unlike in the main body of the paper, this ILP formulation of bidding in TAC assumes linear prices. Table 7 lists the price constants and decision variables for each auction type. For hotels, the only decisions pertain to buy offers; for flights, the agent decides how many tickets to buy now and how many to buy later; for entertainment events, the agent chooses sell quantities as well as buy quantities.

Hotels	Price	Variable (bid)
bid now	\mathcal{Y}_{as}	ϕ_{apq}

Flights and Events	Price	Variable (qty)
buy now	\mathcal{M}_a	μ_a
buy later	\mathcal{Y}_{as}	v_{as}

Events	Price	Variable (qty)
sell now	\mathcal{N}_a	ν_a
sell later	\mathcal{Z}_{as}	ζ_{as}

Table 7: Auction types and associated price constants and decision variables.

A.1 Index Sets

$a \in A$ indexes the set of goods, or auctions.

$a_f \in A_f$ indexes the set of flight auctions.

$a_h \in A_h$ indexes the set of hotel auctions.

$a_e \in A_e$ indexes the set of event auctions.

$c \in C$ indexes the set of clients.

$p \in P$ indexes the set of prices.

12. The precise formulation of RoxyBot-06's bidding ILP appears in the paper by Lee et al. (2007). The formulation here is slightly simplified, but we expect it would perform comparably in TAC. The key differences are in flight and entertainment bidding.

$q \in Q$ indexes the set of quantities
(i.e., the units of each good in each auction).

$s \in S$ indexes the set of scenarios.

$t \in T$ indexes the set of trips.

A.2 Constants

\mathcal{G}_{at} indicates the quantity of good a required to complete trip t .

\mathcal{M}_a indicates the current buy price of a_f, a_e .

\mathcal{N}_a indicates the current sell price of a_e .

\mathcal{Y}_{as} indicates the future buy price of a_f, a_h, a_e in scenario s .

\mathcal{Z}_{as} indicates the future sell price of a_e in scenario s .

\mathcal{H}_a indicates the hypothetical quantity won of hotel a_h .

\mathcal{O}_a indicates the quantity of good a the agent owns.

\mathcal{U}_{ct} indicates client c 's value for trip t .

A.3 Decision Variables

$\Gamma = \{\gamma_{cst}\}$ is a set of boolean variables indicating whether or not client c is allocated trip t in scenario s .

$\Phi = \{\phi_{apq}\}$ is a set of boolean variables indicating whether to bid price p on the q th unit of a_h .

$M = \{\mu_a\}$ is a set of integer variables indicating how many units of a_f, a_e to buy now.

$N = \{\nu_a\}$ is a set of integer variables indicating how many units of a_e to sell now.

$Y = \{v_{as}\}$ is a set of integer variables indicating how many units of a_f, a_e to buy later in scenario s .

$Z = \{\zeta_{as}\}$ is a set of integer variables indicating how many units of a_e to sell later in scenario s .

A.4 Objective Function

$$\max_{\Gamma, \Phi, M, N, Y, Z} \sum_S \left(\underbrace{\sum_{C,T} \mathcal{U}_{ct} \gamma_{cts}}_{trip value} - \underbrace{\sum_{A_f} \left(\overbrace{\mathcal{M}_a \mu_a}^{current} + \overbrace{\mathcal{Y}_{as} v_{as}}^{future} \right)}_{flight cost} - \underbrace{\sum_{A_h, Q, p \geq \mathcal{Y}_{as}} \mathcal{Y}_{as} \phi_{apq}}_{hotel cost} + \right) \quad (19)$$

$$\sum_{A_e} \left(\begin{array}{cc} \overbrace{\mathcal{N}_a \nu_a + \mathcal{Z}_{as} \zeta_{as}}^{\text{event revenue}} & \overbrace{- \mathcal{M}_a \mu_a - \mathcal{Y}_{as} v_{as}}^{\text{event cost}} \\ \overbrace{\text{current}} & \overbrace{\text{future}} \end{array} \right)$$

A.5 Constraints

$$\sum_T \gamma_{cst} \leq 1 \quad \forall c \in C, s \in S \quad (20)$$

$$\overbrace{\sum_{C,T} \gamma_{cst} \mathcal{G}_{at}}^{\text{allocation}} \leq \overbrace{\mathcal{O}_a}^{\text{own}} + \overbrace{(\mu_a + v_{as})}^{\text{buy}} \quad \forall a \in A_f, s \in S \quad (21)$$

$$\overbrace{\sum_{C,T} \gamma_{cst} \mathcal{G}_{at}}^{\text{allocation}} \leq \overbrace{\mathcal{O}_a}^{\text{own}} + \overbrace{\sum_{Q,p \geq \mathcal{Y}_{as}} \phi_{apq}}^{\text{buy}} \quad \forall a \in A_h, s \in S \quad (22)$$

$$\overbrace{\sum_{C,T} \gamma_{cst} \mathcal{G}_{at}}^{\text{allocation}} \leq \overbrace{\mathcal{O}_a}^{\text{own}} + \left(\overbrace{\mu_a + v_{as}}^{\text{buy}} \right) - \left(\overbrace{\nu_a + \zeta_{as}}^{\text{sell}} \right) \quad \forall a \in A_e, s \in S \quad (23)$$

$$\sum_{P,Q} \phi_{apq} \geq \mathcal{H}_a \quad \forall a \in A_h \quad (24)$$

$$\sum_P \phi_{apq} \leq 1 \quad \forall a \in A_h, q \in Q \quad (25)$$

Equation (20) limits each client to one trip in each scenario. Equation (21) prevents the agent from allocating flights that it does not own or buy. Equation (22) prevents the agent from allocating hotels that it does not own or buy. Equation (23) prevents the agent from allocating event tickets that it does not own or buy and not sell. Equation (24) ensures the agent bids on at least HQW units in each hotel auction. Equation (25) prevents the agent from placing more than one buy offer per unit in each hotel auction.

An agent might also be constrained not to place sell offers on more units of each good than it owns, and/or not to place buy (sell) offers for more units of each good than the market supplies (demands).

Note that there is no need to explicitly enforce the bid monotonicity constraints in this ILP formulation:

- “Buy offers must be nonincreasing in k , and sell offers nondecreasing.”

The ILP does not need this constraint because prices are assumed to be linear. In effect, the only decisions the ILP makes are how many units of each good to bid on. Hence, the bids (10, 15, 20) and (20, 15, 10) are equivalent.

- “An agent may not offer to sell for less than the price it is willing to buy.”

The ILP would not choose to place both a buy offer and a sell offer on a good if the buy price of that good exceeds the sell price, because that would be unprofitable.

References

- Ahmed, S., & Shapiro, A. (2002). The sample average approximation method for stochastic programs with integer recourse. *Optimization Online*, <http://www.optimization-online.org>.
- Arunachalam, R., & Sadegh, N. M. (2005). The supply chain trading agent competition. *Electronic Commerce Research and Applications*, 4(1), 66–84.
- Aurell, E., Boman, M., Carlsson, M., Eriksson, J., Finne, N., Janson, S., Kreuger, P., & Rasmusson, L. (2002). A trading agent built on constraint programming. In *Eighth International Conference of the Society for Computational Economics: Computing in Economics and Finance*, Aix-en-Provence.
- Birge, J., & Louveaux, F. (1997). *Introduction to Stochastic Programming*. Springer, New York.
- Cai, K., Gerding, E., McBurney, P., Niu, J., Parsons, S., & S.Phelps (2009). Overview of CAT: A market design competition. Tech. rep. ULCS-09-005, University of Liverpool.
- Cheng, S., Leung, E., Lochner, K., K.O'Malley, Reeves, D., Schvartzman, L., & Wellman, M. (2003). Walverine: A Walrasian trading agent. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pp. 465–472.
- Cheng, S., Leung, E., Lochner, K., K.O'Malley, Reeves, D., Schvartzman, L., & Wellman, M. (2005). Walverine: A Walrasian trading agent. *Decision Support Systems*, 39(2), 169–184.
- Cramton, P. (2006). Simultaneous ascending auctions. In Cramton, P., Shoham, Y., & Steinberg, R. (Eds.), *Combinatorial Auctions*. MIT Press.
- Fritschi, C., & Dorer, K. (2002). Agent-oriented software engineering for successful TAC participation. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 45–46.
- Greenwald, A. (2003). Bidding marginal utility in simultaneous auctions. In *Workshop on Trading Agent Design and Analysis*.
- Greenwald, A., & Boyan, J. (2004). Bidding under uncertainty: Theory and experiments. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, pp. 209–216.
- Greenwald, A., Naroditskiy, V., & Lee, S. (2008). Bidding heuristics for simultaneous auctions: Lessons from tac travel. In *Workshop on Trading Agent Design and Analysis*.
- Greenwald, A., & Boyan, J. (2005). Bidding algorithms for simultaneous auctions: A case study. *Journal of Autonomous Agents and Multiagent Systems*, 10(1), 67–89.
- He, M., & Jennings, N. (2002). SouthamptonTAC: Designing a successful trading agent. In *Proceedings of the Fifteenth European Conference on Artificial Intelligence*, pp. 8–12.

- Jordan, P. R., & Wellman, M. P. (2009). Designing an ad auctions game for the trading agent competition. In *Workshop on Trading Agent Design and Analysis*.
- Kehagias, D., Toulis, P., & Mitkas, P. (2006). A long-term profit seeking strategy for continuous double auctions in a trading agent competition. In *Fourth Hellenic Conference on Artificial Intelligence*, Heraklion.
- Krishna, V. (2002). *Auction Theory*. Academic Press.
- Lee, S. J. (2007). Comparison of bidding algorithms in simultaneous auctions. B.S. honors thesis, Brown University, <http://list.cs.brown.edu/publications/theses/ugrad/>.
- Lee, S., Greenwald, A., & Naroditskiy, V. (2007). Roxybot-06: An (SAA)² TAC travel agent. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pp. 1378–1383.
- Ross, S. M. (2002). *Simulation* (Third edition). Academic Press.
- Sardinha, J. A. R. P., Milidiú, R. L., Paranhos, P. M., Cunha, P. M., & de Lucena, C. J. P. (2005). An agent based architecture for highly competitive electronic markets. In *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, USA*, pp. 326–332.
- Toulis, P., Kehagias, D., & Mitkas, P. (2006). Mertacor: A successful autonomous trading agent. In *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 1191–1198, Hakodate.
- Vetsikas, I., & Selman, B. (2002). WhiteBear: An empirical study of design tradeoffs for autonomous trading agents. In *Workshop on Game-Theoretic Decision-Theoretic Agents*.
- Vickrey, W. (1961). Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16, 8–37.
- Walras, L. (1874). *Éléments d'économie politique pure*. L. Corbaz, Lausanne.
- Wellman, M. P., Greenwald, A., & Stone, P. (2007). *Autonomous Bidding Agents: Strategies and Lessons from the Trading Agent Competition*. MIT Press.
- Wellman, M. P., Reeves, D. M., Lochner, K. M., & Suri, R. (2005). Searching for Walverine 2005. In *Workshop on Trading Agent Design and Analysis*, No. 3937 in Lecture Notes on Artificial Intelligence, pp. 157–170. Springer.
- Wellman, M., Reeves, D., Lochner, K., & Vorobeychik, Y. (2004). Price prediction in a Trading Agent Competition. *Artificial Intelligence Research*, 21, 19–36.

Positioning to Win: A Dynamic Role Assignment and Formation Positioning System

Patrick MacAlpine, Francisco Barrera, and Peter Stone

Department of Computer Science, The University of Texas at Austin
`{patmac,tank225,pstone}@cs.utexas.edu`

Abstract. This paper presents a dynamic role assignment and formation positioning system used by the 2011 RoboCup 3D simulation league champion UT Austin Villa. This positioning system was a key component in allowing the team to win all 24 games it played at the competition during which the team scored 136 goals and conceded none. The positioning system was designed to allow for decentralized coordination among physically realistic simulated humanoid soccer playing robots in the partially observable, non-deterministic, noisy, dynamic, and limited communication setting of the RoboCup 3D simulation league simulator. Although the positioning system is discussed in the context of the RoboCup 3D simulation environment, it is not domain specific and can readily be employed in other RoboCup leagues as it generalizes well to many realistic and real-world multiagent systems.

1 Introduction

Coordinated movement among autonomous mobile robots is an important research area with many applications such as search and rescue [1] and warehouse operations [2]. The RoboCup 3D simulation competition provides an excellent testbed for this line of research as it requires coordination among autonomous agents in a physically realistic environment that is partially observable, non-deterministic, noisy, and dynamic. While low level skills such as walking and kicking are vitally important for having a successful soccer playing agent, the agents must work together as a team in order to maximize their game performance.

One often thinks of the soccer teamwork challenge as being about where the player with the ball should pass or dribble, but at least as important is where the agents position themselves when they *do not* have the ball [3]. Positioning the players in a formation requires the agents to coordinate with each other and determine where each agent should position itself on the field. While there has been considerable research done in the 2D soccer simulation domain (for example by Stone et al. [4] and Reis et al. [5]), relatively little outside of [6] has been published on this topic in the more physically realistic 3D soccer simulation environment. [6], as well as related work in the RoboCup middle size league (MSL) [7], rank positions on the field in order of importance and then iteratively assign the closest available agent to the most important currently unassigned

position until every agent is mapped to a target location. The work presented in this paper differs from the mentioned previous work in the 2D and 3D simulation and MSL RoboCup domains as it takes into account real-world concerns and movement dynamics such as the need for avoiding collisions of robots.

In UT Austin Villa’s positioning system players’ positions are determined in three steps. First, a full team formation is computed (Section 3); second, each player computes the best assignment of players to role positions in this formation according to its own view of the world (Section 4); and third, a coordination mechanism is used to choose among all players’ suggestions (Section 4.4). In this paper, we use the terms (player) position and (player) role interchangeably.

The remainder of the paper is organized as follows. Section 2 provides a description of the RoboCup 3D simulation domain. The formation used by UT Austin Villa is given in Section 3. Section 4 explains how role positions are dynamically assigned to players. Collision avoidance is discussed in Section 5. An evaluation of the different parts of the positioning system is given in Section 6, and Section 7 summarizes.

2 Domain Description

The RoboCup 3D simulation environment is based on SimSpark,¹ a generic physical multiagent system simulator. SimSpark uses the Open Dynamics Engine² (ODE) library for its realistic simulation of rigid body dynamics with collision detection and friction. ODE also provides support for the modeling of advanced motorized hinge joints used in the humanoid agents.

The robot agents in the simulation are homogeneous and are modeled after the Aldebaran Nao robot,³ which has a height of about 57 cm, and a mass of 4.5 kg. The agents interact with the simulator by sending torque commands and receiving perceptual information. Each robot has 22 degrees of freedom: six in each leg, four in each arm, and two in the neck. In order to monitor and control its hinge joints, an agent is equipped with joint perceptors and effectors. Joint perceptors provide the agent with noise-free angular measurements every simulation cycle (20 ms), while joint effectors allow the agent to specify the torque and direction in which to move a joint. Although there is no intentional noise in actuation, there is slight actuation noise that results from approximations in the physics engine and the need to constrain computations to be performed in real-time. Visual information about the environment is given to an agent every third simulation cycle (60 ms) through noisy measurements of the distance and angle to objects within a restricted vision cone (120°). Agents are also outfitted with noisy accelerometer and gyroscope perceptors, as well as force resistance perceptors on the sole of each foot. Additionally, agents can communicate with each other every other simulation cycle (40 ms) by sending messages limited to 20 bytes.

¹ <http://simspark.sourceforge.net/>

² <http://www.ode.org/>

³ <http://www.aldebaran-robotics.com/eng/>

3 Formation

This section presents the formation used by UT Austin Villa during the 2011 RoboCup competition. The formation itself is not a main contribution of this paper, but serves to set up the role assignment function discussed in Section 4 for which a precomputed formation is required.

In general, the team formation is determined by the ball position on the field. As an example, Figure 1 depicts the different role positions of the formation and their relative offsets when the ball is at the center of the field. The formation can be broken up into two separate groups, an offensive and a defensive group. Within the offensive group, the role positions on the field are determined by adding a specific offset to the ball's coordinates. The *onBall* role, assigned to the player closest to the ball, is always based on where the ball is and is therefore never given an offset. On either side of the ball are two forward roles, *forwardRight* and *forwardLeft*. Directly behind the ball is a *stopper* role as well as two additional roles, *wingLeft* and *wingRight*, located behind and to either side of the ball. When the ball is near the edge of the field some of the roles' offsets from the ball are adjusted so as to prevent them from moving outside the field of play.

Within the defensive group there are two roles, *backLeft* and *backRight*. To determine their positions on the field a line is calculated between the center of the team's own goal and the ball. Both backs are placed along this line at specific offsets from the end line. The goalie positions itself independently of its teammates in order to always be in the best position to dive and stop a shot on goal. If the goalie assumes the *onBall* role, however, a third role is included within the defensive group, the *goalieReplacement* role. A field player assigned to the *goalieReplacement* role is told to stand in front of the center of the goal.

During the course of a game there are occasional stoppages in play for events such as kickoffs, goal kicks, corner kicks, and kick-ins. When one of these events occur UT Austin Villa adjusts its team formation and behavior to assume situational set plays which are detailed in a technical report [8].

Kicking and passing have yet to be incorporated into the team's formation. Instead the *onBall* role always dribbles the ball toward the opponent's goal.

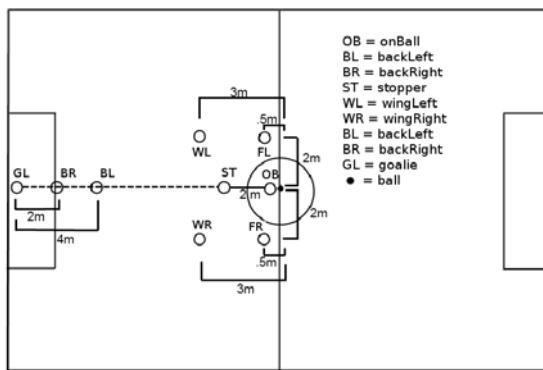


Fig. 1. Formation role positions.

4 Assignment of Agents to Role Positions

Given a desired team formation, we need to map players to roles (target positions on the field). A naïve mapping having each player permanently mapped to one of the roles performs poorly due to the dynamic nature of the game. With such static roles an agent assigned to a defensive role may end up out of position and, without being able to switch roles with a teammate in a better position to defend, allow for the opponent to have a clear path to the goal. In this section, we present a dynamic role assignment algorithm. A role assignment algorithm can be thought of as implementing a role assignment *function*, which takes as input the state of the world, and outputs a one-to-one mapping of players to roles. We start by defining three properties that a role assignment function must satisfy (Section 4.1). We then construct a role assignment function that satisfies these properties (Section 4.2). Finally, we present a dynamic programming algorithm implementing this function (Section 4.3).

4.1 Desired Properties of a Valid Role Assignment Function

Before listing desired properties of a role assignment function we make a couple of assumptions. The first of these is that no two agents and no two role positions occupy the same position on the field. Secondly we assume that all agents move toward fixed role positions along a straight line at the same constant speed. While this assumption is not always completely accurate, the omnidirectional walk used by the agent, and described in [9], gives a fair approximation of constant speed movement along a straight line.

We call a role assignment function *valid* if it satisfies three properties:

1. *Minimizing longest distance* - it minimizes the maximum distance from a player to target, with respect to all possible mappings.
2. *Avoiding collisions* - agents do not collide with each other as they move to their assigned positions.
3. *Dynamically consistent* - a role assignment function f is dynamically consistent if, given a *fixed* set of target positions, if f outputs a mapping m of players to targets at time T , and the players are moving toward these targets, f would output m for every time $t > T$.

The first two properties are related to the output of the role assignment function, namely the mapping between players and positions. We would like such a mapping to minimize the time until all players have reached their target positions because quickly doing so is important for strategy execution. As we assume all players move at the same speed, we start by requiring a mapping to minimize the maximum distance any player needs to travel. However, paths to positions might cross each other, therefore we additionally require a mapping to guarantee that when following it, there are no collisions. The third property guarantees that once a role assignment function f outputs a mapping, f is committed to it as long as there is no change in the target positions. This guarantee is necessary as otherwise agents might unduly thrash between roles thus impeding progress. In the following section we construct a valid role assignment function.

4.2 Constructing a Valid Role Assignment Function

Let M be the set of all one-to-one mappings between players and roles. If the number of players is n , then there are $n!$ possible such mappings. Given a state of the world, specifically n player positions and n target positions, let the *cost* of a mapping m be the n -tuple of distances from each player to its target, sorted in decreasing order. We can then sort all the $n!$ possible mappings based on their costs, where comparing two costs is done lexicographically. Sorted costs of mappings from agents to role positions for a small example are shown in Figure 2.

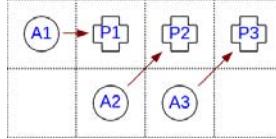


Fig. 2. Lowest lexicographical cost (shown with arrows) to highest cost ordering of mappings from agents (A_1, A_2, A_3) to role positions (P_1, P_2, P_3). Each row represents the cost of a single mapping.

- 1: $\sqrt{2}$ ($A_2 \rightarrow P_2$), $\sqrt{2}$ ($A_3 \rightarrow P_3$), 1 ($A_1 \rightarrow P_1$)
- 2: 2 ($A_1 \rightarrow P_2$), $\sqrt{2}$ ($A_3 \rightarrow P_3$), 1 ($A_2 \rightarrow P_1$)
- 3: $\sqrt{5}$ ($A_2 \rightarrow P_3$), 1 ($A_1 \rightarrow P_1$), 1 ($A_3 \rightarrow P_2$)
- 4: $\sqrt{5}$ ($A_2 \rightarrow P_3$), 2 ($A_1 \rightarrow P_2$), $\sqrt{2}$ ($A_3 \rightarrow P_1$)
- 5: 3 ($A_1 \rightarrow P_3$), 1 ($A_2 \rightarrow P_1$), 1 ($A_3 \rightarrow P_2$)
- 6: 3 ($A_1 \rightarrow P_3$), $\sqrt{2}$ ($A_2 \rightarrow P_2$), $\sqrt{2}$ ($A_3 \rightarrow P_1$)

Denote the role assignment function that always outputs the mapping with the lexicographically smallest cost as f_v . Here we provide an informal proof sketch that f_v is a valid role assignment; we provide a longer, more thorough derivation in a technical report [8].

Theorem 1 f_v is a valid role assignment function.

It is trivial to see that f_v minimizes the longest distance traveled by any agent (Property 1) as the lexicographical ordering of distance tuples sorted in descending order ensures this. If two agents in a mapping are to collide (Property 2) it can be shown, through the triangle inequality, that f_v will find a lower cost mapping as switching the two agents' targets reduces the maximum distance either must travel. Finally, as we assume all agents move toward their targets at the same constant rate, the distance between any agent and target will not decrease any faster than the distance between an agent and the target it is assigned to. This observation serves to preserve the lowest cost lexicographical ordering of the chosen mapping by f_v across all timesteps thereby providing dynamic consistency (Property 3). Section 4.3 presents an algorithm that implements f_v .

4.3 Dynamic Programming Algorithm for Role Assignment

In UT Austin Villa's basic formation, presented in Section 3, there are nine different roles for each of the nine agents on the field. The goalie always fills the *goalie* role and the *onBall* role is assigned to the player closest to the ball. The

other seven roles must be mapped to the agents by f_v . Additionally, when the goalie is closest to the ball, the goalie takes on both the *goalie* and *onBall* roles causing us to create an extra *goalieReplacement* role positioned right in front of the team's goal. When this occurs the size of the mapping increases to eight agents mapped to eight roles. As the total number of mapping permutations is $n!$, this creates the possibility of needing to evaluate $8!$ different mappings.

Clearly f_v could be implemented using a brute force method to compare all possible mappings. This implementation would require creating up to $8! = 40,320$ mappings, then computing the cost of each of the mappings, and finally sorting them lexicographically to choose the smallest one. However, as our agent acts in real time, and f_v needs to be computed during a decision cycle (20 ms), a brute force method is too computationally expensive. Therefore, we present a dynamic programming implementation shown in Algorithm 1 that is able to compute f_v within the time constraints imposed by the decision cycle's length.

Algorithm 1 Dynamic programming implementation

```

1: HashMap bestRoleMap =  $\emptyset$ 
2: Agents =  $\{a_1, \dots, a_n\}$ 
3: Positions =  $\{p_1, \dots, p_n\}$ 
4: for  $k = 1$  to  $n$  do
5:   for each  $a$  in Agents do
6:      $S = \binom{n-1}{k-1}$  sets of  $k - 1$  agents from Agents –  $\{a\}$ 
7:     for each  $s$  in  $S$  do
8:       Mapping  $m_0 = \text{bestRoleMap}[s]$ 
9:       Mapping  $m = (a \rightarrow p_k) \cup m_0$ 
10:      bestRoleMap[ $\{a\} \cup s$ ] = mincost( $m$ , bestRoleMap[ $\{a\} \cup s$ ])
11: return bestRoleMap[Agents]

```

Theorem 2 Let A and P be sets of n agents and positions respectively. Denote the mapping $m := f_v(A, P)$. Let m_0 be a subset of m that maps a subset of agents $A_0 \subset A$ to a subset of positions $P_0 \subset P$. Then m_0 is also the mapping returned by $f_v(A_0, P_0)$.

A key recursive property of f_v that allows us to exploit dynamic programming is expressed in Theorem 2. This property stems from the fact that if within any subset of a mapping a lower cost mapping is found, then the cost of the complete mapping can be reduced by augmenting the complete mapping with that of the subset's lower cost mapping. The savings from using dynamic programming comes from only evaluating mappings whose subset mappings are returned by f_v . This is accomplished in Algorithm 1 by iteratively building up optimal mappings for position sets from $\{p_1\}$ to $\{p_1, \dots, p_n\}$, and using optimal mappings of $k - 1$ agents to positions $\{p_1, \dots, p_{k-1}\}$ (line 8) as a base when constructing each new mapping of k agents to positions $\{p_1, \dots, p_k\}$ (line 9), before saving the lowest cost mapping for the current set of k agents to positions $\{p_1, \dots, p_k\}$ (line 10).

An example of the mapping combinations evaluated in finding the optimal mapping for three agents through the dynamic programming approach of Algorithm 1 can be seen in Table 1. In this example we begin by computing the

distance of each agent to our first role position. Next we compute the cost of all possible mappings of agents to both the first and second role positions and save off the lowest cost mapping of every pair of agents to the the first two positions. We then proceed by sequentially assigning every agent to the third position and compute the lowest cost mapping of all agents mapped to all three positions. As all subsets of an optimal (lowest cost) mapping will themselves be optimal, we need only evaluate mappings to all three positions which include the previously calculated optimal mapping agent combinations for the first two positions.

$\{P_1\}$	$\{P_2, P_1\}$	$\{P_3, P_2, P_1\}$
$A_1 \rightarrow P_1$	$A_1 \rightarrow P_2, f_v(A_2 \rightarrow P_1)$	$A_1 \rightarrow P_3, f_v(\{A_2, A_3\} \rightarrow \{P_1, P_2\})$
$A_2 \rightarrow P_1$	$A_1 \rightarrow P_2, f_v(A_3 \rightarrow P_1)$	$A_2 \rightarrow P_3, f_v(\{A_1, A_3\} \rightarrow \{P_1, P_2\})$
$A_3 \rightarrow P_1$	$A_2 \rightarrow P_2, f_v(A_1 \rightarrow P_1)$ $A_2 \rightarrow P_2, f_v(A_3 \rightarrow P_1)$ $A_3 \rightarrow P_2, f_v(A_1 \rightarrow P_1)$ $A_3 \rightarrow P_2, f_v(A_2 \rightarrow P_1)$	$A_3 \rightarrow P_3, f_v(\{A_1, A_2\} \rightarrow \{P_1, P_2\})$

Table 1. All mappings evaluated during dynamic programming using Algorithm 1 when computing an optimal mapping of agents A_1 , A_2 , and A_3 to positions P_1 , P_2 , and P_3 . Each column contains the mappings evaluated for the set of positions listed at the top of the column.

Recall that during the k th iteration of the dynamic programming process to find a mapping for n agents, where k is the current number of positions that agents are being mapped to, each agent is sequentially assigned to the k th position and then all possible subsets of the other $n - 1$ agents are assigned to positions 1 to $k - 1$ based on computed optimal mappings to the first $k - 1$ positions from the previous iteration of the algorithm. These assignments result in a total of $\binom{n-1}{k-1}$ agent subset mapping combinations to be evaluated for mappings of each agent assigned to the k th position. The total number of mappings computed for each of the n agents across all n iterations of dynamic programming is thus equivalent to the sum of the $n - 1$ binomial coefficients. That is,

$$\sum_{k=1}^n \binom{n-1}{k-1} = \sum_{k=0}^{n-1} \binom{n-1}{k} = 2^{n-1}$$

Therefore the total number of mappings that must be evaluated using our dynamic programming approach is $n2^{n-1}$. For $n = 8$ we thus only have to evaluate 1024 mappings which takes about 3.3 ms for each agent to compute compared to upwards of 50 ms using a brute force approach to evaluate all possible mappings.⁴

4.4 Voting Coordination System

In order for agents on a team to assume correct positions on the field they all must coordinate and agree on which mapping of agents to roles to use. If every agent had perfect information of the locations of the ball and its teammates this would not be a problem as each could independently calculate the optimal mapping to use. Agents do not have perfect information, however, and are limited to

⁴ As measured on an Intel Core 2 Duo CPU E8500 @3.16GHz.

noisy measurements of the distance and angle to objects within a restricted vision cone (120°). Fortunately agents can share information with each other every other simulation cycle (40 ms). The bandwidth of this communication channel is very limited, however, as only one agent may send a message at a time and messages are limited to 20 bytes.

We utilize the agents' limited communication bandwidth in order to coordinate role mappings as follows. Each agent is given a rotating time slice to communicate information, as in [4], which is based on the uniform number of an agent. When it is an agent's turn to send a message it broadcasts to its teammates its current position, the position of the ball, and also what it believes the optimal mapping should be. By sending its own position and the position of the ball, the agent provides necessary information for computing the optimal mapping to those of its teammates for which these objects are outside of their view cones. Sharing the optimal mapping of agents to role positions enables synchronization between the agents, as follows.

First note that just using the last mapping received is dangerous, as it is possible for an agent to report inconsistent mappings due to its noisy view of the world. This can easily occur when an agent falls over and accumulates error in its own localization. Additionally, messages from the server are occasionally dropped or received at different times by the agents preventing accurate synchronization. To help account for inconsistent information, a sliding window of received mappings from the last n time-slots is kept by each agent where n is the total number of agents on a team. Each of these kept messages represents a single vote by each of the agents as to which mapping to use. The mapping chosen is the one with the most votes or, in the case of a tie, the mapping tied for the most votes with the most recent vote cast for it. By using a voting system, the agents on a team are able to synchronize the mapping of agents to role positions in the presence of occasional dropped messages or an agent reporting erroneous data. As a test of the voting system the number of cycles all nine agents shared a synchronized mapping of agents to roles was measured during 5 minutes of gameplay (15,000 cycles). The agents were synchronized 100% of the time when using the voting system compared to only 36% of the time when not using it.

5 Collision Avoidance

Although the positioning system discussed in Section 4 is designed to avoid assigning agents to positions that might cause them to collide, external factors outside of the system's control, such as falls and the movement of the opposing team's agents, still result in occasional collisions. To minimize the potential for these collisions the agents employ an active collision avoidance system. When an obstacle, such as a teammate, is detected in an agent's path the agent will attempt to adjust its path to its target in order to maneuver around the obstacle. This adjustment is accomplished by defining two thresholds around obstacles: a *proximity* threshold at 1.25 meters and a *collision* threshold at .5 meters from an obstacle. If an agent enters the *proximity* threshold of an obstacle it will

adjust its course to be tangent to the obstacle thereby choosing to circle around to the right or left of said obstacle depending on which direction will move the agent closer to its desired target. Should the agent get so close as to enter the *collision* proximity of an obstacle it must take decisive action to prevent an otherwise imminent collision from occurring. In this case the agent combines the corrective movement brought about by being in the *proximity* threshold with an additional movement vector directly away from the obstacle. Figure 3 illustrates the adjusted movement of an agent when attempting to avoid a collision.

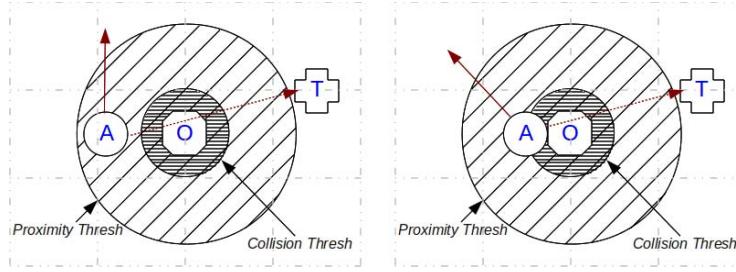


Fig. 3. Collision avoidance examples where agent A is traveling to target T but wants to avoid colliding with obstacle O. The left diagram shows how the agent’s path is adjusted if it enters the *proximity* threshold of the obstacle while the right diagram depicts the agent’s movement when entering the *collision* threshold. The dotted arrow is the agent’s desired path while the solid arrow is the corrected path to avoid a collision.

6 Formation Evaluation

To test how our formation and role positioning system⁵ affects the team’s performance we created a number of teams to play against by modifying the base positioning system and formation of UT Austin Villa.

UT Austin Villa Base agent using the dynamic role positioning system described in Section 4 and formation in Section 3.

NoCollAvoid No collision avoidance.

AllBall No formations and every agent except for the goalie goes to the ball.

NoTeamwork Similar to AllBall except that collision avoidance is also turned off.

NoCommunication Agents do not communicate with each other.

Static Each role is statically assigned to an agent based on its uniform number.

Defensive Defensive formation in which only two agents are in the offensive group.

Offensive Offensive formation in which all agents except for the goalie are positioned in a close symmetric formation behind the ball.

Boxes Field is divided into fixed boxes and each agent is dynamically assigned to a home position in one of the boxes. Similar to system used in [4].

⁵ Video demonstrating our positioning system can be found online at
[http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/
 AustinVilla3DSimulationFiles/2011/html/positioning.html](http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2011/html/positioning.html)

NearestStopper The *stopper* role position is mapped to nearest agent.

PathCost Agents add in the cost of needing to walk around known obstacles (using collision avoidance from Section 5), such as the ball and agent assuming the *onBall* role, when computing distances of agents to role positions.

PositiveCombo Combination of *Offensive*, *PathCost*, and *NearestStopper* attributes.

Table 2. Full game results, averaged over 100 games. Each row corresponds to an agent with varying formation and positioning systems as described in Section 6. Entries show the goal difference (row – column) from 10 minute games versus our base agent, using the dynamic role positioning system described in Section 4 and formation in Section 3, as well as the Apollo3D and CIT3D agents from the 2011 RoboCup China Open. Values in parentheses are the standard error.

	UTAustinVilla	Apollo3D	CIT3D
PositiveCombo	0.33 (.07)	2.16 (.11)	4.09 (.12)
Offensive	0.21 (.09)	1.80 (.12)	3.89 (.12)
AllBall	0.09 (.08)	1.69 (.13)	3.56 (.13)
PathCost	0.07 (.07)	1.27 (.11)	3.25 (.11)
NearestStopper	0.01 (.07)	1.26 (.11)	3.21 (.11)
UTAustinVilla	—	1.05 (.12)	3.10 (.12)
Defensive	-0.05 (.05)	0.42 (.10)	1.71 (.11)
Static	-0.19 (.07)	0.81 (.13)	2.87 (.11)
NoCollAvoid	-0.21 (.08)	0.82 (.12)	2.84 (.12)
NoCommunication	-0.30 (.06)	0.41 (.11)	1.94 (.10)
NoTeamwork	-1.10 (.11)	0.33 (.15)	2.43 (.12)
Boxes	-1.38 (.11)	-0.82 (.13)	1.52 (.11)

Results of UT Austin Villa playing against these modified versions of itself are shown in Table 2. The UT Austin Villa agent is the same agent used in the 2011 competition, except for a bug fix,⁶ and so the data shown does not directly match with earlier released data in [9]. Also shown in Table 2 are results of the modified agents playing against the champion (Apollo3D) and runner-up (CIT3D) of the 2011 RoboCup China Open. These agents were chosen as reference points as they are two of the best teams available with CIT3D and Apollo3D taking second and third place respectively at the main RoboCup 2011 competition. The China Open occurred after the main RoboCup event during which time both teams improved (Apollo3D went from losing by an average of 1.83 to 1.05 goals and CIT3D went from losing by 3.75 to 3.1 goals on average when playing 100 games against our base agent).

Several conclusions can be made from the game data in Table 2. The first of these is that it is really important to be aggressive and always have agents near the ball. This finding is shown in the strong performance of the *Offensive* agent. In contrast to an offensive formation, we see that a very defensive formation used by the *Defensive* agent hurts performance likely because, as the saying goes, the best defense is a good offense. The poor performance of the *Boxes* agent, in which the positions on the field are somewhat static and not calculated as relative offsets to the ball, underscores the importance of being around the ball and adjusting positions on the field based on the current state of the game.

⁶ A bug in collision avoidance present in the 2011 competition agent where it always moved in the direction away from the ball to avoid collisions was fixed.

The likely reason for the success of offensive and aggressive formations grouped close to the ball is because few teams in the league have managed to successfully implement advanced passing strategies, and thus most teams primarily rely on dribbling the ball. Should a team develop good passing skills then a spread out formation might become useful.

The *NearestStopper* agent was created after noticing that the *stopper* role is a very important position on the field so as to always have an agent right behind the ball to prevent breakaways and block kicks toward the goal. Ensuring that the *stopper* role is filled as quickly as possible improved performance slightly. This result is another example of added aggression improving game performance.

Another factor in team performance that shows up in the data from Table 2 is the importance of collision avoidance. Interestingly the *AllBall* agent did almost as well as the *Offensive* agent even though it does not have a set formation. While this result might come as a bit of surprise, collision avoidance causes the *AllBall* agent to form a clumped up mass around the ball which is somewhat similar to that of the *Offensive* agent’s formation. For the strategy of all the agents running to the ball to work well it is imperative to have good collision avoidance. This conclusion is evident from the poor performance of the *NoTeamwork* agent where collision avoidance is turned off with everyone running to the ball, as well as from a result in [9] where the *AllBall* agent lost to the base agent by an average of .43 goals when both agents had a bug in their collision avoidance systems. Turning off collision avoidance, but still using formations, hurts performance as seen in the results of the *NoCollAvoid* agent. Additionally the *PathCost* agent showed an improvement in gameplay by factoring in known obstacles that need to be avoided when computing the distance required to walk to each target.

Another noteworthy observation from the data in Table 2 is that dynamically assigning roles is better than statically fixing them. This finding is clear in the degradation in performance of the *Static* agent. It is important that the agents are synchronized in their decision as to which mapping of agents to roles to use, however, as is noticeable by the dip in performance of the *NoCommunication* agent which does not use the voting system presented in Section 4.4 to synchronize mappings. The best performing agent, that being the *PositiveCombo* agent, demonstrates that the most successful agent is one which employs an aggressive formation coupled with synchronized dynamic role switching, path planning, and good collision avoidance. While not shown in Table 2, the *PositiveCombo* agent beat the *AllBall* agent (which only employs collision avoidance and does not use formations or positioning) by an average of .31 goals across 100 games with a standard error of .09. This resulted in a record of 43 wins, 20 losses, and 37 ties for the *PositiveCombo* agent against the *AllBall* agent.

7 Summary and Discussion

We have presented a dynamic role assignment and formation positioning system for use with autonomous mobile robots in the RoboCup 3D simulation domain — a physically realistic environment that is partially observable, non-deterministic,

noisy, and dynamic. This positioning system was a key component in UT Austin Villa⁷ winning the 2011 RoboCup 3D simulation league competition.

For future work we hope to add passing to our strategy and then develop formations for passing, possibly through the use of machine learning. Additionally we intend to look into ways to compute f_v more efficiently as well as explore other potential functions for mapping agents to role positions.

Acknowledgments

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. Thanks especially to UT Austin Villa 2011 team members Daniel Urieli, Samuel Barrett, Shivararam Kalyanakrishnan, Michael Quinlan, Nick Collins, Adrian Lopez-Mobilia, Art Richards, Nicolae Știurcă, and Victor Vu. LARG research is supported in part by grants from the National Science Foundation (IIS-0917122), ONR (N00014-09-1-0658), and the Federal Highway Administration (DTFH61-07-H-00030). Patrick MacAlpine is supported by a NDSEG fellowship.

References

1. Kitano, H., Tadokoro, S., Noda, I., Matsubara, H., Takahashi, T., Shinjou, A., Shimada, S.: Robocup rescue: search and rescue in large-scale disasters as a domain for autonomous agents research. In: Proc. of 1999 IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC). Volume 6. (1999) 739 –743 vol.6
2. Wurman, P.R., D’Andrea, R., Mountz, M.: Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine* **29** (2008) 9–20
3. Kalyanakrishnan, S., Stone, P.: Learning complementary multiagent behaviors: A case study. In: RoboCup 2009: Robot Soccer World Cup XIII, Springer (2010) 153–165
4. Stone, P., Veloso, M.: Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence* **110** (1999) 241–273
5. Reis, L., Lau, N., Oliveira, E.: Situation based strategic positioning for coordinating a team of homogeneous agents. In Hannebauer, M., Wendler, J., Pagello, E., eds.: Balancing Reactivity and Social Deliberation in Multi-Agent Systems. Volume 2103 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2001) 175–197
6. Chen, W., Chen, T.: Multi-robot dynamic role assignment based on path cost. In: 2011 Chinese Control and Decision Conference (CCDC). (2011) 3721 –3724
7. Lau, N., Lopes, L., Corrente, G., Filipe, N.: Multi-robot team coordination through roles, positionings and coordinated procedures. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2009). (2009) 5841 –5848
8. MacAlpine, P., Urieli, D., Barrett, S., Kalyanakrishnan, S., Barrera, F., Lopez-Mobilia, A., Știurcă, N., Vu, V., Stone, P.: UT Austin Villa 2011 3D Simulation Team report. Technical Report AI11-10, The Univ. of Texas at Austin, Dept. of Computer Science, AI Laboratory (2011)
9. MacAlpine, P., Urieli, D., Barrett, S., Kalyanakrishnan, S., Barrera, F., Lopez-Mobilia, A., Știurcă, N., Vu, V., Stone, P.: UT Austin Villa 2011: A champion agent in the RoboCup 3D soccer simulation competition. In: Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2012). (2012)

⁷ More information about the UT Austin Villa team, as well as video highlights from the 2011 competition, can be found at the team’s website:
<http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/>

Selectively Reactive Coordination for a Team of Robot Soccer Champions

Juan Pablo Mendoza, Joydeep Biswas, Philip Cooksey, Richard Wang,
 Steven Klee, Danny Zhu and Manuela Veloso

School of Computer Science. Carnegie Mellon University
 5000 Forbes Avenue. Pittsburgh, PA, 15213

Abstract

CMDragons 2015 is the champion of the RoboCup Small Size League of autonomous robot soccer. The team won all of its six games, scoring a total of 48 goals and conceding 0. This unprecedented dominant performance is the result of various features, but we particularly credit our novel offense multi-robot coordination. This paper thus presents our Selectively Reactive Coordination (SRC) algorithm, consisting of two layers: A coordinated opponent-agnostic layer enables the team to create its own plans, setting the pace of the game in offense. An individual opponent-reactive action selection layer enables the robots to maintain reactivity to different opponents. We demonstrate the effectiveness of our coordination through results from RoboCup 2015, and through controlled experiments using a physics-based simulator and an automated referee.

1 Introduction

The RoboCup 2015 robot soccer Small-Size League (SSL) consists of teams of six autonomous robots playing on a field of 9m×6m, with overhead cameras that observe the pose of the twelve players and of the orange golf ball used to play. These observations (gathered at 60Hz) are passed to each team’s computer (Zickler et al. 2010), which runs the planning algorithms to choose actions for each individual team robot. Such actions are sent by radio (at 60Hz) to the robots for execution. The RoboCup SSL is a very complex multi-robot planning problem with clear goals to achieve, in a fast-paced adversarial environment, with inevitably non-deterministic real physical sensing and execution.

Many researchers, present authors included, have worked on this research problem (Veloso, Stone, and Han 2000; Veloso, Bowling, and Stone 2000; D’Andrea 2005; Bruce et al. 2008; Sukvichai, Ariyachartphadungkit, and Chaiso 2012; Li et al. 2015), making contributions in real-time sensing (Bruce and Veloso 2003) and control (Behnke et al. 2004), planning (Zickler and Veloso 2009), and teamwork (Stone and Veloso 1999), which have enabled the current RoboCup SSL games to be a fascinating demonstration of effective AI multi-robot planning algorithms under significant uncertainty. Every year, teams improve and the

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

game conditions and rules change to increase the difficulty of the problem and challenge the autonomous planning algorithms (Weitzenfeld et al. 2015). This year, the CMDragons (see Figure 1), composed of the same robot hardware for the last 10 years, won the competition, scoring 48 goals and suffering 0 goals in 6 games. This level of performance had never been reached before in the league. While various defense and offense algorithms contributed to our result (Mendoza et al. 2015), we strongly credit our new coordinated, aggressive, and continuous attack, which we present here.



Figure 1: All the CMDragons 2015 robots (6 play at a time), champions of the RoboCup Small Size League of robot soccer. (*Hardware designed and built by Mike Licitra, and carefully maintained by Joydeep Biswas and Richard Wang.*)

In this paper, we contribute our Selectively Reactive Coordination (SRC) approach to tractably and effectively solve the coordinated soccer offense problem. Our SRC algorithm is composed of two layers: The *coordinated opponent-agnostic* layer enables the team to conduct potentially expensive optimizations offline to find multi-robot team plans that generally perform well. The *individual opponent-reactive action selection* layer is highly reactive to opponents within the constraints imposed by the coordination layer plans, and thus enables the team to adapt appropriately to opponent behavior.

While robot soccer is a specific planning problem, we believe many of the ideas we present generalize to other dynamic multi-robot domains –e.g., capture-the-flag (Atkin, Westbrook, and Cohen 1999), keepaway (Stone et al. 2006), rescue planning (Jennings, Whelan, and Evans 1997), and team patrolling (Agmon et al. 2008)– in which robots balance executing an agreed-upon team plan with reacting to changes in the environment. We hope that this paper inspires others to pursue the robot soccer problem, or to apply or extend our algorithms to other dynamic multi-robot domains.

2 Multi-Robot Offense Coordination: Statement and Overview

One of the core challenges of planning for a team of soccer robots consists of representing and reasoning about the opponent team. The level of opponent-reactivity of team plans can vary, as exemplified by two extremes: (i) The *purely reactive* team, which positions its robots completely in reaction to the adversary, is unable to carry out plans of its own and is susceptible to coercion (Biswas et al. 2014); (ii) The *open loop* team, which positions its robots ignoring the opponent’s state, is unable to appropriately react to opponent behavior. In this work, we introduce a novel intermediate Selectively Reactive Coordination (SRC) algorithm that creates team plans of its own while also responding to the opponent. SRC combines an *opponent-agnostic team coordination* layer with an *opponent-reactive individual action evaluation and selection* layer. This section provides an overview of our SRC algorithm, while Sections 3 and 4 describe in detail the two layers in the context of the CMDragons.

Coordination via Zones and Guard Locations. The SRC creates a skeleton of an opponent-agnostic multi-robot plan \mathcal{P} to be followed by the team, composed by a set of *roles* $R = \{r_1, \dots, r_n\}$ for a team of n robots. The roles capture *what* the team members should be doing and constrains *how* the robots should do it to adhere to the plan. The coordination layer performs two main functions: (i) selects a plan skeleton based on the state of the game, and then (ii) matches each robot ρ_i to a role r_j .

The CMDragons 2015 offense has two types of roles: one *Primary Attacker* (PA), and $(n-1)$ *Support Attackers* (SAs). The PA role is completely opponent-and-situation-driven, and is thus unconstrained by \mathcal{P} . The SAs move to maximize the estimated probability of the team scoring. Plan \mathcal{P} constrains the behavior of each SA_i by (i) bounding its motion a *zone* $z_i \subset \mathbb{R}^2$, and (ii) assigning it a default target *guard location* $p_i^0 \in z_i$. Each element of a zone set $Z = \{(z_1, p_1^0), \dots, (z_{n-1}, p_{n-1}^0)\}$ is assigned to a SA in the team. A plan can consist of a single zone assignment $\mathcal{P} = Z$, or of a sequence of such steps $\mathcal{P} = [Z_1, \dots, Z_k]$. These plan-skeletons encode strategies that work well generally against various opponents. We search for effective plans offline, using extensive data and human knowledge.

Individual Action Selection. SRC considers the opponents and ball in the positioning of the PA and the SAs, leading to an opponent-reactive action selection layer for each robot, that maximizes the estimated probability of scoring a goal. Formally, we have our n offense robots $\mathcal{R} = \{\rho_1, \dots, \rho_n\}$ and a team of adversary robots $\mathcal{R}^o = \{\rho_1^o, \dots, \rho_m^o\}$. We assume full knowledge of the observable state of the world $\mathbf{x} \in X$ consisting of: Each of our robots’ pose and velocity state \mathbf{x}_i^o , the opponent robots’ pose and velocity state \mathbf{x}_i^o , the ball state \mathbf{x}^b , and the state \mathbf{x}^g of the game, with information such as time left and score.

Table 1 shows the higher-level actions of the individual robots (as opposed to low-level actions, such as apply current to the wheel motors, or to the kicker), as used in the coordination approach. In our formulation, the PA executes

Action	Effect	Type
move(p)	Move to location p	Passive
getBall	Move to intercept ball	Active
shoot	Shoot ball to opponent’s goal	Active
pass(p)	Pass ball to location p	Active
dribble	Dribble ball to hold possession	Active

Table 1: Actions available to each robot. Active actions manipulate the ball, while Passive actions do not.

active actions that manipulate the ball, while the SAs perform *passive* actions that do not involve ball manipulation.

Each Support Attacker SA_i moves either to its guard location $p_i^0 \in z_i$, or to a location $p_i^* \in z_i$ to receive a pass from the PA, depending on whether the PA is ready to pass to SA_i . Pass location p^* is computed via optimization as the location within z_i that maximizes the probability of successfully receiving a pass from the PA and then scoring a goal by taking a shoot action.

The PA selects the optimal action a^* among the set of possible active actions A^a . The PA only considers passing to the $(n-1)$ locations p_i^* that the SAs have chosen. Therefore, the action space for the PA is (getBall, shoot, dribble, pass), which can be fully explored and evaluated to estimate the probability of scoring a goal for each action, and choose the optimal action a^* .

Complete Overview of SRC algorithm Algorithm 1 presents the complete algorithm and refers to the rest of the paper. First, the algorithm *jointly* computes $(n-1)$ zones to assign to each SA. Then, each of the n fully-instantiated roles (one PA and $(n-1)$ SAs with zones) is *jointly* optimally assigned to each robot. Finally, each robot plans its actions *individually* within the constraints of its role.

Algorithm 1 Selective Reactive Coordination for Offense.

Input: State of the world \mathbf{x} .

Output: Individual robot actions.

```

function PlanAction( $\mathbf{x}$ )
  Instantiate roles  $r_i$  with zones  $z_i$  (Section 3)
   $\{(z_i, p_i^0)\}_{i=1}^{n-1} \leftarrow \text{ComputeZones}(\mathbf{x})$ 
   $\{r_i\}_{i=1}^n \leftarrow [\text{SA}(z_1, p_1^0), \dots, \text{SA}(z_{n-1}, p_{n-1}^0), \text{PA}]$ 
  Optimally assign roles (Section 3)
   $\{(\rho_i, r_i)\}_{i=1}^n \leftarrow \text{OptAssign}(\{r_i\}_{i=1}^n, \mathbf{x})$ 
  Choose actions individually (Section 4)
  for  $i$  in  $[1, 2, \dots, n]$  do
     $a_i^* \leftarrow \text{IndividualAction}(\rho_i, r_i, \mathbf{x})$ 
  end for
end function

```

This layered joint-individual algorithm maintains tractability: ComputeZones is $O(n)$, OptAssign is $O(n^3)$, and IndividualAction is $O(n + m)$ for each robot, where m is the number of opponents. As the size of the team grows, the OptAssign step might need to be modified to maintain real-time planning.



Figure 2: Coordinated zone assignments for Support Attacker robots. White dashed lines show the zone boundaries; white and orange circles show our SAs and PA respectively. A pass from the PA in (b) triggers a change in zones to those in (c).

3 Role Assignment to Zones

This section addresses the problem of selecting zones and assigning robots to be PA or SAs based on such zones. We explore two selection approaches: *coverage-zones* and *dynamic-zones*. Throughout this paper, we assume that the number n of offense robots is known. Balancing offense and defense, a complex problem on its own, is beyond the scope of this paper, which focuses on offense coordination.

Coverage-zone Selection Our coverage-zone approach is based on an offline definition of zone sets Z_i , each of which cover the opponent’s half of the field. Online, the team chooses the right coverage set Z based on features of the state of the game, such as possession and ball position. This approach follows a long tradition in robot soccer of building upon human knowledge of the game of soccer to reason about zones and formations (Stone and Veloso 1999).

Figure 2a shows a set of Coverage-zones for a four-robot offense used at RoboCup 2015. These predefined zone sets partition the offensive half of the field, giving much freedom to the individual SAs to search for the optimal positioning within these large zones, while ensuring a well-distributed opponent-agnostic formation along the field.

Dynamic-zone Selection The other approach we employed in RoboCup 2015 relies more heavily on coordinated zone selection to determine the flow of actions, leaving little positioning choice to the individual robots. The algorithm coordinates the team of robots to move, as the play progresses, in *sequences of zones* $\mathcal{P} = [Z_1, Z_2, \dots, Z_k]$, each of a smaller size than the coverage-zones.

Each plan \mathcal{P}_i is selected from a set \mathbb{P} of possible plans. To create \mathbb{P} , we first created a set \mathbb{P}_0 of candidate plans, leveraging human knowledge and intuition about the game. Then, we ran extensive simulation tests to find the best-performing plans from \mathbb{P}_0 , according to various performance metrics, such as goals scored and pass completion. Automated plan generation and selection is a subject for future work.

Each of the plans in \mathbb{P} has a set of applicability conditions. For instance, in RoboCup 2015, the set \mathbb{P} was divided into defense, midfield, and offense plans. In general, the goal of these plans is to move the ball toward the opponent’s goal. Transitions from Z_i to Z_{i+1} are triggered when either a robot kicks the ball or a timeout period expires. Figures 2b shows a set of dynamic zones for a 3-robot offense, which evolves to that of Figure 2c when the PA passes the ball.

Optimal role assignment Once the set of roles has been fully instantiated with zones, our algorithm assigns each robot to a role. This assignment is performed optimally, given an appropriate cost function $C_i(\rho_j)$ of assigning role $r_i \in R$ to robot $\rho_j \in \mathcal{R}$: The optimal assignment is a bijection $f : R \rightarrow \mathcal{R}$, such that the total assignment cost $\sum_i C_i(f(r_i))$ is minimized. This optimal assignment can be computed in $O(n^3)$ time (Bertsekas 1981).

The definition of the cost function C_i is crucial to achieve the right assignments. The optimal assignment is the one that maximizes the probability of scoring a goal. In our algorithm, we approximate this by a cost function that represents the *time* $t_i(\rho_j)$ that it would take robot ρ_j to fulfill role r_i , multiplied by an importance factor w_i :

$$C_i(\rho_j) = w_i t_i(\rho_j) \quad (1)$$

Our robots are homogeneous, and thus there is no intrinsic benefit of choosing one over another for a specific role. Thus, the assignment that maximizes the probability of scoring is the one that minimizes the probability of the opponent disrupting our plans. This probability highly correlates with the time taken to perform a plan, and thus we minimize the total completion time, giving a higher importance w_i to the PA than to the SAs.

The cost of assigning the role of Primary Attacker to robot ρ_j is thus the time that it will take for ρ_j to drive to location p_{PA}^* computed to be the best ball interception location for ρ_j (or 0 if ρ_j is already in possession of the ball). Similarly, the cost for the Support Attackers is computed as the time it will take for robot ρ_j to drive to location p_{ij}^* evaluated to be the best location within zone z_i to support the PA.

4 Opponent-Reactive Individual Action Evaluation and Selection

Once the coordinated team has assigned each robot a fully instantiated role (see Section 3), each robot performs its role individually, with limited communication with the rest of the team (Browning et al. 2005). This individualization enables our algorithms to scale tractably with the number of robots.

4.1 Primary Attacker (PA)

The PA is the most complex role, as it requires the robot to make various decisions and use several different skills. For clarity of presentation, we present the PA at a level of

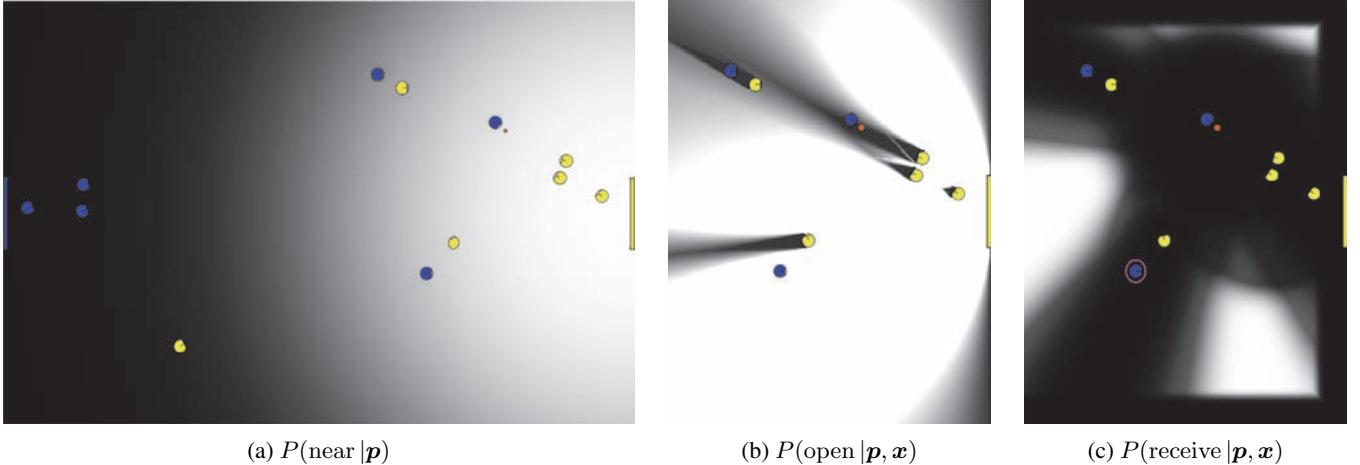


Figure 3: Estimated probabilities for individual decision-making of the blue robots. Lighter gray indicates higher probability points. Probability that p (a) is near enough to the yellow goal and (b) has a wide enough angle on it, to shoot and score, and (c) Probability that the highlighted SA can receive a pass at different locations p from the PA holding the orange ball.

abstraction suitable for this paper; for example, even though the PA has various skills for intercepting a moving ball, here we join them into a single getBall action.

The goal of the PA is to manipulate the ball to maximize the probability of scoring a goal. When the PA does not have possession of the ball, it executes the getBall action. When the PA has possession, it chooses among three action types: shoot on goal (shoot), pass to a Support Attacker SA_{*i*} (pass_{*i*}), or individually dribble the ball (dribble). We briefly explain the dribble action, new to the CMDragons 2015.

Individual Dribbling. To hold possession of the ball, the PA uses a rotating dribbler bar that imparts back-spin on the ball, making it roll toward the robot. Furthermore, the PA drives with the ball to keep it away from opponents, while driving to the location p_t of the most promising pass or shoot option (as computed in Equation 4 or 3 below). Thus, the PA, with location p_{PA} , balances the goal of driving in the direction $\vec{u}_t = p_t - p_{PA}$ of its target, and avoiding the closest opponent with direction $\vec{u}_o = p^o - p_{PA}$, if its distance $d_o = |p^o - p_{PA}|$ is smaller than a threshold D_{\min} . Figure 4 shows a diagram of these quantities.

The robot aligns with \vec{u}_t by rotating its heading \vec{u}_b towards \vec{u}_t along the smaller angle ϕ^- unless there exists a *turning threat* threat($\vec{u}_b, \vec{u}_o, d_o$) within ϕ^- , in which case it rotates along the larger angle ϕ^+ . A *turning threat* exists if the opponent is within ϕ^- , and closer than D_{\min} :

$$\text{threat}(\vec{u}_b, \vec{u}_o, d_o) = (d_o \leq D_{\min}) \wedge ((\vec{u}_b \times \vec{u}_o)(\vec{u}_b \times \vec{u}_t) \geq 0) \wedge ((\vec{u}_t \times \vec{u}_o)(\vec{u}_t \times \vec{u}_b) \geq 0) \quad (2)$$

Once aligned, the robot dribbles that ball towards p_t , while avoiding obstacles along the way (Bruce and Veloso 2006).

Primary Attacker Algorithm. Algorithm 2 shows the procedure for choosing the optimal PA action. The PA estimates the probability that each one of these actions will lead

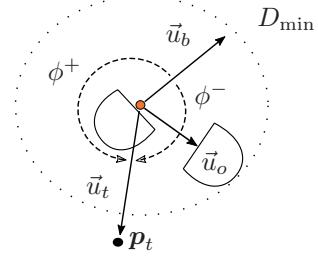


Figure 4: Variables used to execute the dribble action. The PA holding the orange ball decides how to drive to p_t while avoiding the opponent in direction \vec{u}_o .

to a goal, given the location of the ball p_b and the state of the world¹ x . The probability of scoring a goal by shooting is estimated as the probability that the ball is close enough to the opponent's goal for a shot to be effective *and* that the robot has a wide enough angle on the goal:

$$P(\text{goal} | \text{shoot}, p_b, x) = P(\text{near} | p_b) P(\text{open} | p_b, x). \quad (3)$$

Figures 3a and 3b illustrate these two functions, treated as independent for simplicity.

The probability of scoring a goal by first passing is estimated as the probability that the pass will successfully reach its target robot *and* that the robot will subsequently successfully shoot on the goal from the estimated world state x'_i after the pass, obtained from forward predictions of our own robots, and assuming the opponents are static. This probability is highly dependent on the location p_i^* at which robot SA_{*i*} decides to receive the pass:

$$P(\text{goal} | \text{pass}_i, p_i^*, x) = P(\text{receive}_i | p_i^*, x) \times P(\text{goal} | \text{shoot}, p_i^*, x'_i). \quad (4)$$

¹While the location of the ball is part of x , we state it explicitly for clarity of explanation in the remainder of the paper.

Algorithm 2 Primary Attacker action-selection algorithm.
Input: Robot ρ_i , instantiated role PA, world state \mathbf{x}
Output: Chosen individual action \mathbf{a}^* .

```

1: function INDIVIDUALACTION( $\rho_i$ , PA,  $\mathbf{x}$ )
2:   if not in possession of the ball then
3:      $\mathbf{a}^* \leftarrow \text{getBall}$ 
4:   else
5:      $A \leftarrow \{\text{shoot, dribble, pass}_1, \dots, \text{pass}_{n-1}\}$ 
6:      $\mathbf{a}^* \leftarrow \arg \max_{\mathbf{a} \in A} [P(\text{goal} | \mathbf{a}, \mathbf{x})]$ 
7:   end if
8:   return  $\mathbf{a}^*$ 
9: end function
```

This approximation is a one-step lookahead that assumes the receiving robot will shoot on the goal. Estimating further pass success probabilities is computationally expensive and inaccurate in a highly dynamic adversarial domain (Zickler and Veloso 2010; Trevizan and Veloso 2012). However, as these probabilities are estimated at every timestep, multiple passes emerge naturally. The estimated pass success probability $P(\text{receive}_i | \mathbf{p}_i^*, \mathbf{x})$ itself is a composition of various probabilities, as described in Section 4.2.

The probability of scoring by dribbling (to be followed by a pass or shot) is estimated by a constant value k_d :

$$P(\text{goal} | \text{dribble}, \mathbf{p}_b, \mathbf{x}) = k_d \quad (5)$$

This simple estimate provided significant results during RoboCup 2015: Our PA used dribble 42 times in the semi-final and 60 times in the final, giving our team 47 and 146 seconds of additional ball possession time, respectively.

4.2 Support Attackers (SAs).

The task of each Support Attacker SA_i is to maximize the probability of the team scoring by supporting the PA from within its assigned zone z_i . Each SA_i thus (i) searches for the location \mathbf{p}_i^* that maximizes the probability of receiving a pass and then scoring a goal (Equation 4) and then (ii) moves to \mathbf{p}_i^* at the right time to receive a pass from the PA. We now describe these two components and the resulting algorithm.

Optimal Pass Location Search. To find the location \mathbf{p}_i^* that maximizes Equation 4, we must compute estimates of the two factors in it. Section 4.1 addresses the computation of $P(\text{goal} | \text{shoot}, \mathbf{p}, \mathbf{x}')$. To estimate the probability $P(\text{receive}_i | \mathbf{p}, \mathbf{x})$ of successfully receiving a pass at location \mathbf{p} , we compile a set of conditions c_k that must all be true for a pass to be received, and combine their individual probabilities assuming independence (Biswas et al. 2014):

$$P(\text{receive}_i | \mathbf{p}, \mathbf{x}) = \prod_k P(c_k | \mathbf{p}, \mathbf{x}). \quad (6)$$

Examples of these individual factors include the probability of the pass not being intercepted by any opponent, and the probability of not losing the ball by attempting to receive too close to the field boundary. Figure 3c shows an example of the resulting map from location on the field to probability of success. Because the optimization space is only 2-dimensional, we effectively employ random sampling to find the optimal location \mathbf{p}_i^* that maximizes Equation 4.

Pass-ahead Computation After choosing \mathbf{p}_i^* , SA_i and PA execute the pass using a *pass-ahead* procedure: Only once SA_i has calculated that its own navigation time $t_p(\mathbf{p}_i^*)$ is comparable to the time $t_b(\mathbf{p}_i^*)$ that the pass will take to get there, SA_i moves to \mathbf{p}_i^* . Until then, SA_i moves to (or stays at) its guard location \mathbf{p}_i^0 . Thus, we combine an opponent-agnostic default location \mathbf{p}_i^0 that enables our robots to move the world to a less dynamic and thus more predictable state, with an opponent-reactive pass location \mathbf{p}_i^* that enables our robots to adapt appropriately. This pass-ahead coordination (Biswas et al. 2014) has been crucial in the high pass success rate of the CMDragons, as it makes the task of marking our SAs significantly more difficult. Figure 5 illustrates a pass-ahead maneuver leading to a goal in RoboCup 2015.

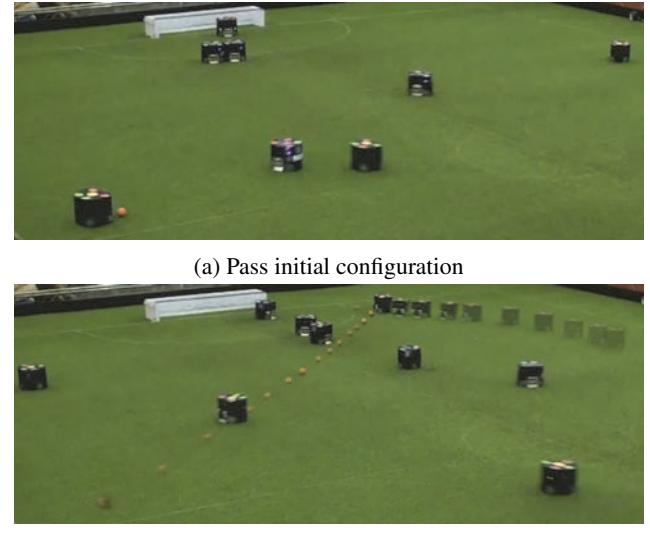


Figure 5: Pass-ahead maneuver leading to a goal in RoboCup 2015. The figure shows the initial and final world configurations, and the motion of the ball and pass receiver.

Secondary Attacker Algorithm. Algorithm 3 describes the procedure for choosing the optimal SA action. While robots choose their individual actions independently, we enable limited communication to avoid computation redundancy. For example, SA robots communicate to the PA robots their computed values for \mathbf{p}_i^* , $P(\text{goal} | \text{pass}_i, \mathbf{p}_i^*, \mathbf{x})$, $t_b(\mathbf{p}_i^*)$ and $t_p(\mathbf{p}_i^*)$.

5 RoboCup 2015 and Simulation Results

Our layered coordinated offense approach proved extremely successful during the RoboCup 2015 competition. Here, we discuss the CMDragons’ performance in the tournament, and conduct simulation experiments to further support SRC.

CMDragons Performance in RoboCup 2015 The RoboCup 2015 SSL tournament consisted of 17 teams from various universities in the world. The CMDragons played three games during the Round Robin stage (RR1, RR2, RR3), one Quarter-Final (QF), one Semi-Final (SF),

Algorithm 3 Support Attacker action-selection algorithm.
Input: Robot ρ_i , instantiated role SA(z_i, p_i^0), world state x
Output: Chosen individual action a^* .

```

1: function INDIVIDUALACTION( $\rho_i$ , SA( $z_i, p_i^0$ ),  $x$ )
2:    $p_i^* \leftarrow \text{FindOptLoc}(\rho_i, z_i, x)$ 
3:   if  $t_b(p_i^*) \leq t_\rho(p_i^*)$  then
4:      $a^* \leftarrow \text{move}(p_i^*)$ 
5:   else
6:      $a^* \leftarrow \text{move}(p_i^0)$ 
7:   end if
8:   return  $a^*$ 
9: end function

```

and the Final (F). We won all 6 games, scoring 48 goals, and conceding 0. Table 2 summarizes goal, shot, and pass statistics for our team in each game.

Game	Shots			Passes		
	Scored	Missed	Succ.%	Compl.	Missed	Succ.%
RR1	6	11	35.3	32	9	78.1
RR2	10	5	66.7	14	4	77.8
RR3	10	15	40.0	30	7	81.1
QF	15	25	37.5	38	4	90.5
SF	2	29	6.5	51	12	81.0
F	5	15	25.0	29	15	65.9
Total	48	100	32.4	194	51	79.2
Avg	8	16.7	32.4	32.3	8.5	79.2

Table 2: Statistics for each CMDragons game in RoboCup 2015. RR2 and RR3 ended early due to a 10-goal mercy rule, after 10:20 and 18:25 minutes respectively (normally, games last 20 minutes).

Our team demonstrated an effective level of coordination: with an average of 32.3 passes completed per game, our team had a 79.2% pass completion rate². Furthermore, most of the team’s goals were collective efforts: 22 goals were scored directly after 1 pass, 11 directly after 2 consecutive passes, and 1 after 3 consecutive passes.

Simulation Validation We complement our real-world results from RoboCup with experiments run on a PhysX-based simulator, with an automated referee (Zhu, Biswas, and Veloso 2015) that enabled extensive testing without human intervention. We test our novel SRC algorithm against two alternate highly competitive versions of our team described below. We tested each condition for over 500 minutes of regular game-play, with no free kicks, fitting the focus of this paper. While it is always difficult to clearly dominate over other versions of our team by changing a single aspect of the team, the SRC algorithm showed a significant improvement over the alternatives.

First, we tested our team using SRC against a team in which each SA_i moves to its Individually-Optimal Location p_i^* computed over the entire field (team IndivOpt). Thus,

²For anecdotal reference, the pass completion rates of the human teams in the 2014 World Cup final were 71% and 80%.

IndivOpt has the advantage of globally-optimal individual positioning, while SRC has the advantage of team coordination. Table 3 shows that the SRC offense outperforms team IndivOpt in terms of offensive statistics.

Team	Goals Scored	Blocked (Goalie)	Blocked (Other)	Total Shots
SRC	59	128	1636	1823
IndivOpt	35	85	1381	1501
ExactPlan	53	142	2453	2648
SRC	67	165	2045	2227

Table 3: Results of simulation experiments. Our SRC algorithm outperforms a team that positions robots in their Individually-Optimal Location (IndivOpt), and a team that positions them according to an Exact Team Plan (ExactPlan)

Then, we tested SRC against a team in which the coordination layer, instead of creating only a skeleton of a team plan, creates an Exact Team Plan (team ExactPlan); this plan assigns robots to specific locations, rather than zones as in SRC. Thus, ExactPlan has the advantage that effective complete plans can be specified in advance, but it lacks the reactivity of SRC. Similar exact plan strategies have been successfully applied by highly competitive teams in SSL (Zhao et al. 2014). Table 3 shows that SRC outperforms ExactPlan: even though ExactPlan shoots more, SRC shoots past the defense more and scores more. We hypothesize that ExactPlan shoots more in general because its PA is less likely to find good passing options, and decides to shoot instead.

6 Conclusion

This paper presents a Selectively Reactive Coordination (SRC) approach to the problem of offensive team coordination in the context of robot soccer. This approach achieves a tradeoff between the ability to create team plans independently of the opponents, and the ability to react appropriately to different opponent behaviors.

An opponent-agnostic coordination layer creates skeletons of team plans that are generally effective against various opponents. These plans are encoded by zones to be assigned to the Support Attacker robots, while the Primary Attacker robot is unconstrained by the plan. Team plans can be generated offline, and can therefore leverage human knowledge or extensive computation. During the game, plans are selected based on efficiently-computable conditions.

Given the constraints imposed by the selected team plan, each robot plans its actions individually. At this level, decisions are highly reactive to the behavior of the opponents.

We present empirical support for our approach through statistics of our unprecedented performance in RoboCup 2015, and through controlled experimental results obtained using a physics-based simulator and an automated referee.

Acknowledgments This research was partially supported by ONR grant number N00014-09-1-1031 and AFRL grant number FA87501220291. The views and conclusions contained in this document are those solely of the authors.

References

- Agmon, N.; Kraus, S.; Kaminka, G.; et al. 2008. Multi-robot perimeter patrol in adversarial settings. In *Proceedings of the International Conference of Robotics and Automation (ICRA)*, 2339–2345.
- Atkin, M. S.; Westbrook, D. L.; and Cohen, P. R. 1999. Capture the flag: Military simulation meets computer games. In *Proceedings of AAAI Spring Symposium Series on AI and Computer Games*, 1–5.
- Behnke, S.; Egorova, A.; Gloye, A.; Rojas, R.; and Simon, M. 2004. Predicting away robot control latency. In *RoboCup 2003: Robot Soccer World Cup VII*, 712–719. Springer.
- Bertsekas, D. P. 1981. A new algorithm for the assignment problem. *Mathematical Programming* 21(1):152–171.
- Biswas, J.; Mendoza, J. P.; Zhu, D.; Choi, B.; Klee, S.; and Veloso, M. 2014. Opponent-driven planning and execution for pass, attack, and defense in a multi-robot soccer team. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 493–500.
- Browning, B.; Bruce, J.; Bowling, M.; and Veloso, M. 2005. STP: Skills, tactics, and plays for multi-robot control in adversarial environments. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering* 219(1):33–52.
- Bruce, J., and Veloso, M. 2003. Fast and Accurate Vision-Based Pattern Detection and Identification. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*.
- Bruce, J. R., and Veloso, M. M. 2006. Safe multirobot navigation within dynamics constraints. *Proceedings of the IEEE, Special Issue on Multi-Robot Systems* 94(7):1398–1411.
- Bruce, J.; Zickler, S.; Licita, M.; and Veloso, M. 2008. CMDragons: Dynamic passing and strategy on a champion robot soccer team. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 4074–4079.
- D’Andrea, R. 2005. The Cornell RoboCup Robot Soccer Team: 1999–2003. In *Handbook of Networked and Embedded Control Systems*. Springer. 793–804.
- Jennings, J.; Whelan, G.; and Evans, W. 1997. Cooperative search and rescue with a team of mobile robots. In *Proceedings of the International Conference on Advanced Robotics (ICAR)*, 193–200.
- Li, C.; Xiong, R.; Ren, Z.; Tang, W.; and Zhao, Y. 2015. ZJUNlict: RoboCup 2014 Small Size League Champion. In *RoboCup 2014: Robot World Cup XVIII*. Springer. 47–59.
- Mendoza, J. P.; Biswas, J.; Zhu, D.; Wang, R.; Cooksey, P.; Klee, S.; and Veloso, M. 2015. CMDragons2015: Coordinated Offense and Defense of the SSL Champions. In *Proceedings of the International RoboCup Symposium*.
- Stone, P., and Veloso, M. 1999. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence* 110(2):241–273.
- Stone, P.; Kuhlmann, G.; Taylor, M. E.; and Liu, Y. 2006. Keepaway soccer: From machine learning testbed to benchmark. In *RoboCup 2005: Robot Soccer World Cup IX*. Springer. 93–105.
- Sukvichai, K.; Ariyachartphadungkit, T.; and Chaiso, K. 2012. Robot hardware, software, and technologies behind the SKUBA robot team. In *RoboCup 2011: Robot Soccer World Cup XV*. Springer. 13–24.
- Trevizan, F., and Veloso, M. 2012. Trajectory-Based Short-Sighted Probabilistic Planning. In *Proceedings of NIPS-12*.
- Veloso, M.; Bowling, M.; and Stone, P. 2000. The CMUnited-98 Champion Small-Robot Team. *Advanced Robotics* 13(8).
- Veloso, M.; Stone, P.; and Han, K. 2000. The CMUnited-97 Robotic Soccer Team: Perception and Multiagent Control. *Robotics and Autonomous Systems* 29 (2-3):133–143.
- Weitzenfeld, A.; Biswas, J.; Akar, M.; and Sukvichai, K. 2015. RoboCup Small-Size League: Past, Present and Future. In *RoboCup 2014: Robot World Cup XVIII*. Springer. 611–623.
- Zhao, Y.; Xiong, R.; Tong, H.; Li, C.; and Fang, L. 2014. ZJUNlict: RoboCup 2013 Small Size League Champion. In *RoboCup 2013: Robot World Cup XVII*.
- Zhu, D.; Biswas, J.; and Veloso, M. 2015. AutoRef: Towards Real-Robot Soccer Complete Automated Refereeing. In *RoboCup 2014: Robot World Cup XVIII*. Springer. 419–430.
- Zickler, S., and Veloso, M. 2009. Efficient Physics-Based Planning: Sampling Search Via Non-Deterministic Tactics and Skills. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*.
- Zickler, S., and Veloso, M. 2010. Variable Level-of-Detail Motion Planning in Environments with Poorly Predictable Bodies. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*.
- Zickler, S.; Laue, T.; Birbach, O.; Wongphati, M.; and Veloso, M. 2010. SSL-vision: The shared vision system for the RoboCup Small Size League. In *RoboCup 2009: Robot Soccer World Cup XIII*. Springer. 425–436.

The CMUnited-98 Champion Simulator Team *

Peter Stone, Manuela Veloso, and Patrick Riley

Computer Science Department, Carnegie Mellon University
Pittsburgh, PA 15213

{pstone,veloso}@cs.cmu.edu, priley@andrew.cmu.edu

<http://www.cs.cmu.edu/~pstone>, <http://www.andrew.cmu.edu/~priley>

A shortened version of this paper appears in
“RoboCup-98: Robot Soccer World Cup II.” M. Asada and H. Kitano eds.
Springer Verlag, Berlin, 1999.

Abstract. The CMUnited-98 simulator team became the 1998 RoboCup simulator league champion by winning all 8 of its games, outscoring opponents by a total of 66–0. CMUnited-98 builds upon the successful CMUnited-97 implementation, but also improves upon it in many ways. This article describes the complete CMUnited-98 software, emphasizing the recent improvements. Coupled with the publicly-available CMUnited-98 source code, it is designed to help other RoboCup and multi-agent systems researchers build upon our success.

1 Introduction

The CMUnited-98 simulator team became the 1998 RoboCup [4] simulator league champion by winning all 8 of its games, outscoring opponents by a total of 66–0. CMUnited-98 builds upon the successful CMUnited-97 implementation [8], but also improves upon it in many ways.

The most notable improvements are the individual agent skills and the strategic agent positioning in anticipation of passes from teammates. While the success of CMUnited-98 also depended on our previous research innovations including layered learning [9], a flexible teamwork structure [10], and a novel communication paradigm [10], these techniques are all described elsewhere. The purpose of this article is to clearly and fully describe the low-level CMUnited-98 agent architecture as well as the key improvements over the previous implementation.

Coupled with the publicly-available CMUnited-98 source code [11], this article is designed to help researchers involved in the RoboCup software challenge [5] build upon our success. Throughout the article, we assume that the reader is familiar with the soccer server [1].

* This research is sponsored in part by the DARPA/RL Knowledge Based Planning and Scheduling Initiative under grant number F30602-95-1-0018. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the U. S. Government.

The rest of the article is organized as follows. Section 2 gives an overview of the entire agent architecture. Section 3 describes the agents' method of keeping an accurate and precise world model. Section 4 details the low-level skills available to the agents. Section 5 presents the CMUnited-98 collaborative coordination mechanisms. Section 6 summarizes the RoboCup-98 results and Section 7 concludes.

2 Agent Architecture Overview

CMUnited-98 agents are capable of perception, cognition, and action. By perceiving the world, they build a model of its current state. Then, based on a set of behaviors, they choose an action appropriate for the current world state.

A driving factor in the design of the agent architecture is the fact that the simulator operates in fixed cycles of length 100 msec. As presented in Section [1], the simulator accepts commands from clients throughout a cycle and then updates the world state all at once at the end of the cycle. Only one action command (dash, kick, turn, or catch) is executed for a given client during a given cycle.

Therefore, agents (simulator clients) should send exactly one action command to the simulator in every simulator cycle. If more than one command is sent in the same cycle, a random one is executed, possibly leading to undesired behavior. If no command is sent during a simulator cycle, an action opportunity has been lost: opponent agents who have acted during that cycle may gain an advantage.

In addition, since the simulator updates the world at the end of every cycle, it is advantageous to try to determine the state of the world at the end of the previous cycle when choosing an action for the current cycle. As such, the basic agent loop during a given cycle t is as follows:

- Assume the agent has consistent information about the state of the world at the end of cycle $t - 2$ and has sent an action during cycle $t - 1$.
- While the server is still in cycle $t - 1$, upon receipt of a sensation (see, hear, or sense_body), store the new information in temporary structures. Do not update the current state.
- When the server enters cycle t (determined either by a running clock or by the receipt of a sensation with time stamp t), use all of the information available (temporary information from sensations and predicted effects of past actions) to **update the world model** to match the server's world state (the "real world state") at the end of cycle $t - 1$. Then **choose and send an action** to the server for cycle t .
- Repeat for cycle $t + 1$.

While the above algorithm defines the overall agent loop, much of the challenge is involved in updating the world model effectively and choosing an appropriate action. The remainder of this section goes into these processes in detail.

3 World Modeling

When acting based on a world model, it is important to have as accurate and precise a model of the world as possible at the time that an action is taken. In order to achieve this goal, CMUnited-98 agents gather sensory information over time, and process the information by incorporating it into the world model immediately prior to acting.

3.1 Object Representation

There are several objects in the world, such as the goals and the field markers which remain stationary and can be used for self-localization. Mobile objects are the agent itself, the ball, and 21 other players (10 teammates and 11 opponents). These objects are represented in a type hierarchy as illustrated in Figure 1.

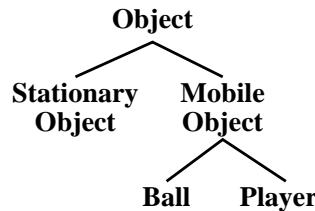


Fig. 1. The agent's object type hierarchy.

Each agent's world model stores an instantiation of a stationary object for each goal, sideline, and field marker; a ball object for the ball; and 21 player objects. Since players can be seen without their associated team and/or uniform number, the player objects are not identified with particular individual players. Instead, the variables for team and uniform number can be filled in as they become known.

Mobile objects are stored with confidence values within [0,1] indicating the confidence with which their locations are known. The confidence values are needed because of the large amount of hidden state in the world: no object is seen consistently. While it would be a mistake to only remember objects that are currently in view, it is also wrong to assume that a mobile object will stay still (or continue moving with the same velocity) indefinitely. By decaying the confidence in unseen objects over time, agents can determine whether or not to rely on the position and velocity values [2].

All information is stored as global coordinates even though both sensor and actuator commands are specified in relative coordinates (angles and distances relative to the agent's position on the field). Global coordinates are easier to store and maintain as the agent moves around the field because the global coordinates of stationary objects do not change as the agent moves, while the relative

coordinates do. It is a simple geometric calculation to convert the global coordinates to relative coordinates on demand as long as the agent knows its own position on the field.

The variables associated with each object type are as follows:

Object :

- Global (x, y) position coordinates
- Confidence within $[0,1]$ of the coordinates' accuracy

Stationary Object : nothing additional

Mobile Object :

- Global (dx, dy) velocity coordinates
- Confidence within $[0,1]$ of the coordinates' accuracy

Ball : nothing additional

Player :

- Team
- Uniform number
- Global θ facing angle
- Confidence within $[0,1]$ of the angle's accuracy

3.2 Updating the World Model

Information about the world can come from

- Visual information;
- Audial information;
- Sense_body information; and
- Predicted effects of previous actions.

Visual information arrives as relative distances and angles to objects in the player's view cone. Audial information could include information about global object locations from teammates. Sense_body information pertains to the client's own status including stamina, view mode, and speed.

Whenever new information arrives, it is stored in temporary structures with time stamps and confidences (1 for visual information, possibly less for audial information). Visual information is stored as relative coordinates until the agent's exact location is determined.

When it is time to act during cycle t , all of the available information is used to best determine the server's world state at the end of cycle $t - 1$. If no new information arrived pertaining to a given object, the velocity and actions taken are used by the predictor to predict the new position of the object and the confidence in that object's position and velocity are both decayed.

When the agent's world model is updated to match the end of simulator cycle $t - 1$, first the agent's own position is updated to match the time of the last sight; then those of the ball and players are updated.

The Agent Itself: Since visual information is given in coordinates relative to the agent's position, it is important to determine the agent's exact position at the time of the sight. When updating the world model to match the end of simulator cycle $t - 1$, there may have been visual information with time stamp $t - 1$ and/or t (anything earlier would have been incorporated into the previous update of the world model at the end of cycle $t - 1$).

If the latest visual information has time stamp $t - 1$, then the agent's own position is not updated until after the other objects have been updated as their coordinates are stored relative to the old agent position. On the other hand, if the latest visual information has time stamp t , or if there has been no new visual information since the last world-state update, the agent's status can be updated immediately.

In either case, the following process is used to update the information about the agent:

- If new visual information has arrived:
 - The agent's position can be determined accurately by using the relative coordinates of one seen line and the closest stationary object.
- If no visual information has arrived:
 - Bring the velocity up to date, possibly incorporating the predicted effects of any actions (a dash) taken during the previous cycle.
 - Using the previous position and velocity, predict the new position and velocity.
- If available, reset the agent's speed as per the sense_body information.
Assume velocity is in the direction the agent is facing.
- Bring the player's stamina up to date either via the sense_body information or from the predicted action effects.

The Ball: As the key focus of action initiative in the domain, the ball's position and velocity drives a large portion of the agents' decisions. As such, it is important to have as accurate and up-to-date information about the ball as possible.

The ball information is updated as follows:

- If there was new visual information, use the agent's absolute position at the time (determined above), and the ball's temporarily stored relative position to determine the ball's absolute position at the time of the sight.
- If velocity information is given as well, update the velocity. Otherwise, check if the old velocity is correct by comparing the new ball position with the expected ball position.
- If no new visual information arrived or the visual information was from cycle $t - 1$, estimate its position and velocity for cycle t using the values from cycle $t - 1$. If the agent kicked the ball on the previous cycle, the predicted resulting ball motion is also taken into account.
- If the ball should be in sight (i.e. its predicted position is in the player's view cone), but isn't (i.e. visual information arrived, but no ball information was included), set the confidence to 0.

- Information about the ball may have also arrived via communication from teammates. If any heard information would increase the confidence in the ball’s position or velocity at this time, then it should be used as the correct information. Confidence in teammate information can be determined by the time of the information (did the teammate see the ball more recently?) and the teammate’s distance to the ball (since players closer to the ball see it more precisely).

Ball velocity is particularly important for agents when determining whether or not (or how) to try to intercept the ball, and when kicking the ball. However, velocity information is often not given as part of the visual information string, especially when the ball is near the agent and kickable. Therefore, when necessary, the agents attempt to infer the ball’s velocity indirectly from the current and previous ball positions.

Teammates and Opponents: The biggest challenge in keeping track of player positions is that the visual information often does not identify the player that is seen [1]. One might be tempted to ignore all ambiguously-specified players. However, for strategic planning it is very useful to have a complete picture of the player positions around the field.

In general, player positions and velocities are determined and maintained in the same way as in the case of the ball. A minor addition is that the direction a player is facing is also available from the visual information.

When a player is seen without full information about its identity, previous player positions can be used to help disambiguate the identity. Knowing the maximum distance a player can move in any given cycle, it is possible for the agent to determine whether a seen player could be the same as a previously identified player. If it is physically possible, the agent assumes that they are indeed the same player.

Since different players can see different regions of the field in detail, communication can play an important role in maintaining accurate information about player locations.

From the complete set of player locations, an agent can determine both the defensive and offensive off-sides lines. It is particularly important for forwards to stay in front of the last opponent defender in order to avoid being called off-sides. Forwards periodically look towards the opponent defenders in order to increase the accuracy of their location information.

4 Agent Skills

Once the agent has determined the server’s world state for cycle t as accurately as possible, it can choose and send an action to be executed at the end of the cycle. In so doing, it must choose its local goal within the team’s overall strategy. It can then choose from among several low-level skills which provide it with basic capabilities. The output of the skills are primitive movement commands.

The skills available to CMUnited-98 players include kicking, dribbling, ball interception, goaltending, defending, and clearing. The implementation details

of these skills are described in this section.

The common thread among these skills is that they are all *predictive, locally optimal skills* (PLOS). They take into account predicted world models as well as predicted effects of future actions in order to determine the optimal primitive action from a local perspective, both in time and in space.

One simple example of PLOS is each individual agent's stamina management. The server models stamina as having a replenishable and a non-replenishable component. Each is only decremented when the current stamina goes below a fixed threshold. Each player monitors its own stamina level to make sure that it never uses up any of the non-replenishable component of its stamina. No matter how fast it should move according to the behavior the player is executing, it slows down its movement to keep itself from getting too tired. While such behavior might not be optimal in the context of the team's goal, it is locally optimal considering the agent's current tired state.

Even though the skills are predictive, the agent *commits* to only one action during each cycle. When the time comes to act again, the situation is completely reevaluated. If the world is close to the anticipated configuration, then the agent will act similarly to the way it predicted on previous cycles. However, if the world is significantly different, the agent will arrive at a new sequence of actions rather than being committed to a previous plan. Again, it will only execute the first step in the new sequence.

4.1 Kicking

There are three points about the kick model of the server that should be understood before looking at our kicking style. First, a kick changes the ball's velocity by vector addition. That is, a kick accelerates the ball in a given direction, as opposed to setting the velocity. Second, an agent can kick the ball when it is in the "kickable area" which is a circle centered on the player (see Figure 2). Third, the ball and the player can collide. The server models a collision when the ball and player are overlapping at the end of a cycle. If there is a collision, the two bodies are separated and their velocities multiplied by -0.1 .

As a first level of abstraction when dealing with the ball, all reasoning is done as a desired trajectory for the ball for the next cycle. Before a kick is actually sent to the server, the difference between the ball's current velocity and the ball's desired velocity is used to determine the kick to actually perform. If the exact trajectory can not be obtained, the ball is kicked such that the direction is correct, even if the speed is not.

In order to effectively control the ball, a player must be able to kick the ball in any direction. In order to do so, the player must be able to move the ball from one side of its body to the other without the ball colliding with the player. This behavior is called the *turnball* behavior. It was developed based on code released by the PaSo'97 team[7]. The desired trajectory of a turnball kick is calculated by getting the ray from the ball's current position that is tangent to a circle around the player (see Figure 3). Note that there are two possible such rays which correspond to the two directions that the ball can be turned around

the player. Care is taken to ensure that the ball stays well within the kickable area from kick to kick so that the player keeps control of the ball.

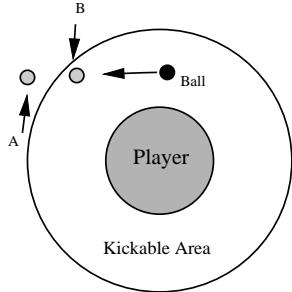


Fig. 2. Basic kicking with velocity prediction.

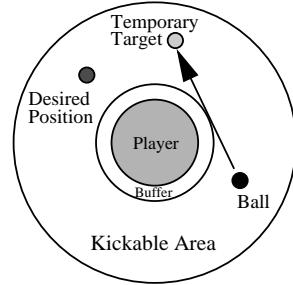


Fig. 3. The turnball skill.

The next important skill is the ability to kick the ball in a given direction, either for passing or shooting. The first step is to figure out the target speed of the ball. If the agent is shooting, the target speed is the maximum ball speed, but for a pass, it might be better to kick the ball slower so that the receiving agent can intercept the ball more easily. In this case, the agent must take into account the ball's deceleration over time when determining how hard to kick the ball.

In order to get the ball to the desired speed, several kicks in succession are usually required. By putting the ball to the side of the player (relative to the desired direction of the kick) the agent can kick the ball several times in succession. If a higher ball speed is desired, the agent can use the turnball kicks to back the ball up so that enough kicks can be performed to accelerate the ball.

This skill is very predictive in that it looks at future velocities of the ball given slightly different possible kicks. In some cases, doing a weaker kick one cycle may keep the ball in the kickable area so that another kick can be executed the following cycle. In Figure 2, the agent must choose between two possible kicks. Kicking the ball to position A will result in the ball not being kickable next cycle; if the ball is already moving quickly enough, this action may be correct. However, a kick to position B followed by a kick during the next cycle may result in a higher overall speed. Short term velocity prediction is the key to these decisions.

4.2 Dribbling

Dribbling is the skill which allows the player to move down the field while keeping the ball close to the player the entire time. The basic idea is fairly simple: alternate kicks and dashes so that after one of each, the ball is still close to the player.

Every cycle, the agent looks to see that if it dashes this cycle, the ball will be in its kickable area (and not be a collision) at the next cycle. If so, then the agent dashes, otherwise it kicks. A kick is always performed assuming that on the next cycle, the agent will dash. As an argument, the low-level dribbling code takes the angle relative to the direction of travel at which the player should aim the ball (see Figure 4). This is called the “dribble angle” and its valid values are $[-90, 90]$. Deciding what the dribble angle should be is discussed in Section 4.3.

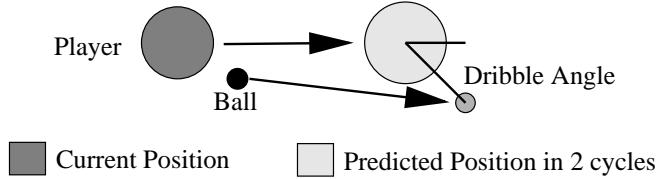


Fig. 4. The basic dribbling skill.

First the predicted position of the agent (in 2 cycles) is calculated:

$$p_{new} = p_{current} + v + (v * pdecay + a)$$

where p_{new} is the predicted player position, $p_{current}$ is the current position of the player, v is the current velocity of the player, $pdecay$ is the server parameter `player_decay`, and a is the acceleration that a dash gives. The a value is usually just the dash power times the `dash_power_rate` in the direction the player is facing, but stamina may need to be taken into account.

Added to p_{new} is a vector in the direction of the dribble angle and length such that the ball is in the kickable area. This is the target position p_{target} of the ball. Then the agent gets the desired ball trajectory by the following formula:

$$traj = \frac{p_{target} - p_{ball}}{1 + bdecay}$$

where $traj$ is the target trajectory of the ball, p_{ball} is the current ball position, and $bdecay$ is the server parameter `ball_decay`. This process is illustrated in Figure 4.

If for some reason this kick can not be done (it would be a collision for example), then a turnball kick is done to get the ball in the right position. Then the next cycle, a normal dribble kick should work.

As can be seen from these calculations, the basic dribbling is highly predictive of the positions and velocities of the ball and player. It is also quite local in that it only looks 2 cycles ahead and recomputes the best action every cycle.

4.3 Smart Dribbling

The basic dribbling takes one parameter that was mentioned above: the dribble angle. Smart dribbling is a skill layered on the basic dribbling skill that decides

the best dribble angle based on opponent positions. Intuitively, the agent should keep the ball away from the opponents, so that if an opponent is on the left, the ball is kept on the right, and vice versa.

The agent considers all nearby opponents that it knows about. Each opponent is given a “vote” about what the dribble angle should be; each opponent votes for the valid angle $[-90, 90]$ that is farthest from itself. For example, an opponent at 45 degrees, would vote for -90, while an opponent at -120 degrees would vote for 60. Each opponent’s vote is weighted by the distance and angle relative to the direction of motion. Closer opponents and opponents more in front of the agent are given more weight (see Figure 5).

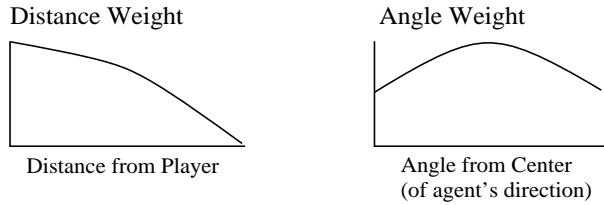


Fig. 5. The weights for smart dribbling.

4.4 Ball Interception

There are two types of ball interception, referred to as active and passive interception. The passive interception is used only by the goaltender in some particular cases, while the rest of the team uses only the active interception. Each cycle, the interception target is recomputed so that the most up to date information about the world is used.

The *active interception* is similar to the one used by the Humboldt ’97 team[3]. The active interception predicts the ball’s position on successive cycles, and then tries to predict whether the player will be able to make it to that spot before the ball does, taking into account stamina and the direction that the player is facing. The agent aims for the earliest such spot.

This process can be used for teammates as well as for the agent itself. Thus, the agent can determine which player should go for the ball, and whether it can get there before the opponents do.

The *passive interception* is much more geometric. The agent determines the closest point along the ball’s current trajectory that is within the field. By prediction based on the ball’s velocity, the agent decides whether it can make it to that point before the ball. If so, then the agent runs towards that point.

4.5 Goaltending

The assumption behind the movement of the goaltender is that the worst thing that could happen to the goaltender is to lose sight of the ball. The sooner

the goaltender sees a shot coming, the greater chance it has of preventing a goal. Therefore, the goaltender generally uses the widest view mode and uses backwards dashing when appropriate to keep the ball in view to position itself in situations that are not time-critical.

Every cycle that the ball is in the defensive zone, the goaltender looks to see if the ball is in the midst of a shot. It does this by extending the ray of the ball's position and velocity and intersecting that with the baseline of the field. If the intersection point is in the goaltender box and the ball has sufficient velocity to get there, the ball is considered to be a shot (though special care is used if an opponent can kick the ball this cycle). Using the passive interception if possible (see Section 4.4), the goaltender tries to get in the path of the ball and then run at the ball to grab it. This way, if the goaltender misses a catch or kick, the ball may still collide with the goaltender and thus be stopped.

When there is no shot coming the goaltender positions itself in anticipation of a future shot. Based on the angle of the ball relative to the goal, the goaltender picks a spot in the goal to guard; call this the "guard point." The further the ball is to the side of the field, the further the goaltender guards to that side. Then, a rectangle is computed that shrinks as the ball gets closer (though it never shrinks smaller than the goaltender box). The line from the guard point to the ball's current position is intersected with the rectangle, and that is the desired position of the goaltender.

4.6 Defending

CMUnited-98 agents are equipped with two different defending modes: opponent tracking and opponent marking. In both cases, a particular opponent player is selected as the target against which to defend. This opponent can either be selected individually or as a defensive unit via communication (the latter is the case in CMUnited-98).

In either case, the agent defends against this player by observing its position over time and position itself strategically so as to minimize its usefulness to the other team. When *tracking*, the agent stays between the opponent and the goal at a generous distance, thus blocking potential shots. When *marking*, the agent stays close to the opponent on the ball-opponent-goal angle bisector, making it difficult for the opponent to receive passes and shoot towards the goal. Defensive marking and tracking positions are illustrated in Figure 6.

When marking and tracking, it is important for the agent to have accurate knowledge about the positions of both the ball and the opponent (although the ball position isn't strictly relevant for tracking, it is used for the decision of whether or not to be tracking). Thus, when in the correct defensive position, the agent always turns to look at the object (opponent or ball) in which it is least confident of the correct position. The complete algorithm, which results in the behavior of doggedly following a particular opponent and glancing back and forth between the opponent and ball, is as follows:

- If the ball position is unknown, look for the ball.

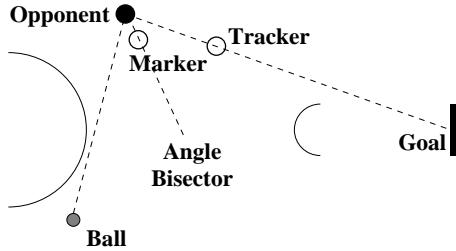


Fig. 6. Positioning for defensive tracking and marking.

- Else, if the opponent position is unknown, look for the opponent.
- Else, if not in the correct defensive position, move to that position.
- else, look towards the object, ball or opponent, which has been seen less recently (lower confidence value).

This defensive behavior is locally optimal in that it defends according to the opponent's current position, following it around rather than predicting its future location. However, in both cases, the defensive positioning is chosen in anticipation of the opponent's future possible actions, i.e. receiving a pass or shooting.

4.7 Clearing

Often in a defensive position, it is advantageous to just send the ball upfield, clearing it from the defensive zone. If the agent decides that it cannot pass or dribble while in a defensive situation, it will clear the ball. The important decision in clearing the ball is where to clear it to. The best clears are upfield, but not to the middle of the field (you don't want to center the ball for the opponents), and also away from the opponents.

The actual calculation is as follows. Every angle is evaluated with respect to its usefulness, and the expected degree of success. The usefulness is a sine curve with a maximum of 1 at 30 degrees, .5 at 90 degrees, and 0 at -90, where a negative angle is towards the middle of the field. The actual equation is (Θ is in degrees):

$$\text{usefulness}(\Theta) = \frac{\sin(\frac{3}{2}\Theta + 45) + 1}{2} \quad (1)$$

The expected degree of success is evaluated by looking at an isosceles triangle with one vertex where the ball is, and congruent sides extending in the direction of the target being evaluated. For each opponent in the triangle, its distance from the center line of the triangle is divided by the distance from the player on that line. For opponent C in Figure 7, these values are w and d respectively. The expected success is the product of all these quotients. In Figure 7, opponent A would not affect the calculation, being outside the triangle, while opponent

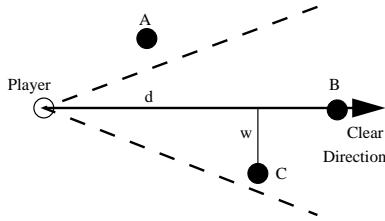


Fig. 7. Measuring the expected success of a clear.

B would lower the expected success to 0, since it is on the potential clear line ($w = 0$).

By multiplying the usefulness and expected success together for each possible clear angle, and taking the maximum, the agent gets a crude approximation to maximizing the expected utility of a clear.

There is a closely related behavior of offensive “sending.” Rather than trying to clear the ball to the sides, the agent sends the ball to the middle of the offensive zone with hopes that a teammate will catch up to the ball before the defenders. This is useful if the agent is too tired or unable to dribble for some reason. It is especially useful to beat an offside trap because it generally requires the defenders to run back to get the ball.

The only difference with defensive clearing is the usefulness function. For sending, the usefulness function is linear, with slope determined by the agent’s Y position on the field. The closer the agent is to the sideline, the steeper the slope, and the more that it favors sending to the middle of the field.

5 Coordination

Given all of the individual skills available to the CMUnited-98 clients, it becomes a significant challenge to coordinate the team so that the players are not all trying to do the same thing at the same time. Of course one and only one agent should execute the goaltending behavior. But it is not so clear how to determine when an agent should move towards the ball, when it should defend, when it should dribble, or clear, etc.

If all players act individually — constantly chase the ball and try to kick towards the opponent goal — they will all get tired, there will be nowhere to pass, and the opponents will have free reign over most of the field. Building upon the innovations of the CMUnited-97 simulator team [8], the CMUnited-98 team uses several complex coordination mechanisms, including reactive behavior modes, pre-compiled multi-agent plans and strategies, a flexible teamwork structure, a novel anticipatory offensive positioning scheme, and a sophisticated communication paradigm.

5.1 Behavior Modes

A player's top-level behavior decision is its behavior mode. Implemented as a rule-based system, the behavior mode determines the abstract behavior that the player should execute. For example, there is a behavior mode for the set of states in which the agent can kick the ball. Then, the decision of what to do with the ball is made by way of a more involved decision mechanism. On each action cycle, the first thing a player does is re-evaluate its behavior mode.

The behavior modes include:

Goaltend: Only used by the goaltender.

Localize: Find own field location if it's unknown.

Face Ball: Find the ball and look at it.

Handle Ball: Used when the ball is kickable.

Active Offense: Go to the ball as quickly as possible. Used when no teammate could get there more quickly.

Auxiliary Offense: Get open for a pass. Used when a nearby teammate has the ball.

Passive Offense: Move to a position likely to be useful offensively in the future.

Active Defense: Go to the ball even though another teammate is already going. Used in the defensive end of the field.

Auxiliary Defense: Mark an opponent.

Passive Defense: Track an opponent or go to a position likely to be useful defensively in the future.

The detailed conditions and effects of each behavior mode are beyond the scope of this article. However, they will become more clear in subsequent sections as the role-based flexible team structure is described in Section 5.3.

5.2 Locker-Room Agreement

At the core of the CMUnited-98 coordination mechanism is what we call the Locker-Room Agreement [10]. Based on the premise that agents can periodically meet in safe, full-communication environments, the locker-room agreement specifies how they should act when in low-communication, time-critical, adversarial environments.

The locker-room agreement includes specifications of the flexible teamwork structure (Section 5.3) and the inter-agent communication paradigm (Section 5.5). A good example of the use of the locker-room agreement is CMUnited-98's ability to execute pre-compiled multi-agent plans after dead-ball situations. While it is often difficult to clear the ball from the defensive zone after goal kicks, CMUnited-98 players move to pre-specified locations and execute a series of passes that successfully move the ball out of their half of the field. Such "set plays" exist in the locker-room agreement for all dead-ball situations.

A new addition to CMUnited-98's locker-room agreement is a defensive off-sides strategy. Since the rules of the soccer server prohibit an opponent from

receiving a pass when located behind the last defender on the opponent's attacking half of the field², it is an effective defensive strategy to move all the defenders forward towards midfield. However, if only one defender is farther back than the rest of the team, the strategy can back-fire horribly.

To take advantage of this rule using the locker-room agreement, the team agrees on a formula based on the location of the ball and the opponent's furthest-back defender. This strategy relies on relatively consistent sensing by all of the defensive players, but it does not require any communication. Independently, the players can dynamically adjust their positions as the ball and opponents move so that the team's defenders stay in a coordinated line.

The CMUnited-98 offside line was always at least 15 meters behind the current ball position to prevent opponents from dribbling through to goal and at least 40 meters behind the opponents last defender to allow enough room in the midfield to pass the ball amongst teammates.

5.3 Roles and Formations

Like CMUnited-97, CMUnited-98 is organized around the concept of flexible formations consisting of flexible roles. Roles are defined independently of the agents that fill them: homogeneous agents (all except the goalie) can freely switch roles as time progresses. Each role specifies the behavior of the agent filling the role, both in terms of positioning on the field and in terms of the behavior modes that should be considered. For example, forwards never go into auxiliary defense mode and defenders never go into auxiliary offense mode.

A formation is a collection of roles, again defined independently from the agents. Just as agents can dynamically switch roles within a formation, the entire team can dynamically switch formations. After testing about 10 formations, the CMUnited-98 team ended up selecting from among 3 different formations. A standard formation with 4 defenders, 3 midfielders, and 3 forwards (4-3-3) was used at the beginnings of the games. If losing by enough goals relative to the time left in the game (as determined by the locker-room agreement), the team would switch to an offensive 3-3-4 formation. When winning by enough, the team switched to a defensive 5-3-2 formation.

Formations also include sub-formations, or units, for dealing with issues of local importance. For example, the defensive unit can be concerned with marking opponents while not involving the midfielders or forwards. A player can be a part of more than one unit.

For a detailed presentation of roles, formations, and units, see [10].

5.4 SPAR

The flexible roles defined in the CMUnited-97 software were an improvement over the concept of rigid roles. Rather than associating fixed (x, y) coordinates with

² As in real soccer, the offside rule is more complicated than that. But for the purposes of this article, the above definition is sufficient.

each position, an agent filling a particular role was given a range of coordinates in which it could position itself. Based on the ball's position on the field, the agent would position itself so as to increase the likelihood of being useful to the team in the future.

However, by taking into account the positions of other agents as well as that of the ball, an even more informed positioning decision can be made. The idea of *strategic position by attraction and repulsion* (SPAR) is one of the novel contributions of the CMUnited-98 research which has been applied to both the simulator and the small robot teams [12].

When positioning itself using SPAR, the agent uses a multi-objective function with attraction and repulsion points subject to several constraints. To formalize this concept, we introduce the following variables:

- P - the desired position for the passive agent in anticipation of a passing need of its active teammate;
- n - the number of agents on each team;
- O_i - the current position of each opponent, $i = 1, \dots, n$;
- T_i - the current position of each teammate, $i = 1, \dots, (n - 1)$;
- B - the current position of the active teammate and ball;
- G - the position of the opponent's goal.

SPAR extends similar approaches of using potential fields for highly dynamic, multi-agent domains [6]. The probability of collaboration in the robotic soccer domain is directly related to how “open” a position is to allow for a successful pass. Thus, SPAR maximizes the distance from other robots and minimizes the distance to the ball and to the goal, namely:

- *Repulsion* from opponents, i.e., maximize the distance to each opponent: $\forall i, \max dist(P, O_i)$
- *Repulsion* from teammates, i.e., maximize the distance to other passive teammates: $\forall i, \max dist(P, T_i)$
- *Attraction* to the active teammate and ball: $\min dist(P, B)$
- *Attraction* to the opponent's goal: $\min dist(P, G)$

This formulation is a multiple-objective function. To solve this optimization problem, we restate the problem as a single-objective function. As each term may have a different relevance (e.g. staying close to the goal may be more important than staying away from opponents), we want to apply a different weighting function to each term, namely f_{O_i} , f_{T_i} , f_B , and f_G , for opponents, teammates, the ball, and the goal, respectively. Our anticipation algorithm then maximizes a weighted single-objective function with respect to P :

$$\max \left(\sum_{i=1}^n f_{O_i}(dist(P, O_i)) + \sum_{i=1}^{n-1} f_{T_i}(dist(P, T_i)) - f_B(dist(P, B)) - f_G(dist(P, G)) \right)$$

In our case, we use $f_{O_i} = f_{T_i} = x$, $f_B = 0$, and $f_G = x^2$. For example, the last term of the objective function above expands to $(dist(P, G))^2$.

One constraint in the simulator team relates to the position, or role, that the passive agent is playing relative to the position of the ball. The agent only considers locations that within one of the four rectangles, illustrated in Figure 5.4: the one closest to the position home of the position that it is currently playing. This constraint helps ensure that the player with the ball will have several different passing options in different parts of the field. In addition, players don't need to consider moving too far from their positions to support the ball.

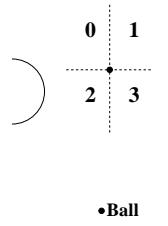


Fig. 8. The four possible rectangles, each with one corner at the ball's location, considered for positioning by simulator agents when using SPAR.

Since this position-based constraint already encourages players to stay near the ball, we set the ball-attraction weighting function f_B to the constant function $y = 0$. In addition to this first constraint, the agents observe three additional constraints. In total, the constraints in the simulator team are:

- Stay in an area near home position;
- Stay within the field boundaries;
- Avoid being in an off-sides position;
- Stay in a position in which it would be possible to receive a pass.

This last constraint is evaluated by checking that there are no opponents in a cone with vertex at the ball and extending to the point in consideration.

In our implementation, the maximum of the objective function is estimated by sampling its values over a fine-grained mesh of points that satisfy the above constraints.

Using this SPAR algorithm, agents are able to *anticipate* the collaborative needs of their teammates by positioning themselves in such a way that the player with the ball would have several useful passing options.

5.5 Communication

The soccer server provides a challenging communication environment for teams of agents. With a single, low-bandwidth, unreliable communication channel for all 22 agents and limited communication range and capacity, agents must not rely on any particular message reaching any particular teammate. Nonetheless,

when a message does get through, it can help distribute information about the state of the world as well as helping to facilitate team coordination.

All CMUnited-98 messages include a certain amount of state information from the speaker's perspective. Information regarding object position and teammate roles are all given along with the confidence values associated with this data. All teammates hearing the message can then use the information to augment their visual state information.

The principle functional uses of communication in CMUnited-98 are

- To ensure that all participants in a set play are ready to execute the multi-step plan. In this case, since the ball is out of play, time is not a critical issue.
- To assign defensive marks. The captain of the defensive unit (the goaltender in most formations) determines which defenders should mark or track which opponent forwards. The captain then communicates this information periodically until receiving a confirmation message.

For a detailed specification of the communication paradigm as it was first developed for CMUnited-97, see [10].

5.6 Ball Handling

One of the most important decisions in the robotic soccer domain arises when the agent has control of the ball. In this state, it has the options of dribbling the ball in any direction, passing to any teammate, shooting the ball, clearing the ball, or simply controlling the ball.

In CMUnited-98, the agent uses a complex heuristic decision mechanism, incorporating a machine learning module, to choose its action. The best teammate to receive a potential pass (called *potential receiver* below) is determined by a decision tree trained off-line [9]. Following is a rough sketch of the decision-making process without all of the parametric details.

To begin with, since kicks (i.e. shots, passes, and clears) can take several cycles to complete (Section 4.1), the agent remembers the goal of a previously started kick and continues executing it. When no kick is in progress (do the first that applies):

- If close to the opponent's goal and no defenders are blocking the path to the goal (defined as a cone with vertex at the ball): shoot or dribble based on the goaltender's position, the position of the closest opponent, and the distance to the goal.
- At the other extreme, if close to the agent's own goal and there is an opponent nearby: clear the ball.
- If approaching the line of the last opponent defender: dribble the ball forward if possible; otherwise send the ball (clear) past the defender.
- If the potential receiver is closer to the goal and has a clear shot: pass to the potential receiver.
- If no opponents are in the direct path to the goal: dribble to the goal.

- If fairly close to the opponent’s goal and there is at most one opponent in front of the goal: shoot.
- If no opponents are in the way of one of the corner flags: dribble towards the corner flag.
- If there is a potential receiver: pass.
- If it’s possible to hold onto the ball without moving (at most one opponent is nearby): hold the ball.
- Otherwise: Kick the ball away (clear).

6 Results

In order to test individual components of the CMUnited-98 team, it is best to compile performance results for the team with and without these components as we have done elsewhere [10]. However, competition against other, independently-created teams is useful for evaluating the system as a whole.

At the RoboCup-98 competition, CMUnited-98 won all 8 of its games by a combined score of 66–0, finishing first place in a field of 34 teams. Table 1 details the game results.

Opponent	Affiliation	Score (CMU-Opp.)
UU	Utrecht University, The Netherlands	22 – 0
TUM / TUMSA	Technical University Munich, Germany	2 – 0
Kasuga-Bitos II	Chubu University, Japan	5 – 0
Andhill’98	NEC, Japan	8 – 0
ISIS	Information Sciences Institute (USC), USA	12 – 0
Rolling Brains	Johannes Gutenberg-University Mainz, Germany	13 – 0
Windmill Wanderers	University of Amsterdam, The Netherlands	1 – 0
AT-Humboldt’98	Humboldt University of Berlin, Germany	3 – 0
TOTAL		66 – 0

Table 1. The scores of CMUnited-98’s games in the simulator league of RoboCup-98. CMUnited-98 won all 8 games, finishing in 1st place out of 34 teams.

From observing the games, it was apparent that the CMUnited-98 low-level skills were superior in the first 6 games: CMUnited-98 agents were able to dribble around opponents, had many scoring opportunities, and suffered few shots against.

However, in the last 2 games, the CMUnited-98 strategic formations, communication, and ball-handling routines were put more to the test as the Windmill

Wanderers (3rd place) and AT-Humboldt'98 (2nd place) also had similar low-level capabilities. In these games, CMUnited-98's abilities to use set-plays to clear the ball from its defensive zone, to get past the opponents' off-sides traps, and to maintain a cohesive defensive unit became very apparent. Many of the goals scored by CMUnited-98 were a direct result of the opponent team being unable to clear the ball from its own end after a goal kick: a CMUnited-98 player would intercept the clearing pass and quickly shoot it into the goal. In particular, two of the goals in the final game against AT-Humboldt'98 were scored in this manner. On the other hand, the CMUnited-98 simulator team was able to clear the ball successfully from its own zone using its ability to execute set-plays, or pre-compiled multi-agent plans. Rather than kicking the ball up the middle of the field, one player would pass out to the sideline to a second player that would then clear the ball up the field. After a series of 3 or 4 passes, the ball was usually safely in the other half of the field.

Another strategic advantage that was clear throughout CMUnited-98's games was the players' abilities to maintain a coherent defensive unit exploiting the off-sides rule, and conversely, its ability to get through the defense of other teams. Often, the opposing teams were unable to get anywhere near the CMUnited-98 goal because of the defenders' ability to stay in front of some of the opposing forwards, thus rendering them off-sides and prohibiting them from ever successfully receiving the ball.

In order to deal with opposing teams that tried to use a similar technique, the CMUnited-98 forwards would kick the ball towards the offensive corners of the field (the “sending” skill described in Section 4.7) and then either get to the ball before the defenders or intercept defenders' clearing passes. CMUnited-98 scored several nice goals after such kicks to the corners.

In addition to the strategic reasoning that helped the team win its final two games, the fine points of the dribbling and goaltending skills also came into play. Using their predictive, locally optimal skills (PLOS—see Section 4), the CMUnited-98 players were occasionally able to dribble around opponents for shots. At a crucial moment against the Windmill Wanderers, the CMUnited-98 goaltender made a particularly important save: winning 1–0 near the end of the game, a shot got past the goaltender, but it was able to turn and catch the ball before the ball entered the goal.

7 Conclusion

The success of CMUnited-98 at RoboCup-98 was due to several technical innovations ranging from predictive locally optimal skills (PLOS) to strategic positioning using attraction and repulsion (SPAR). Building on the innovations of CMUnited-97, including flexible formation, a novel communication paradigm, and machine learning modules, CMUnited-98 successfully combines low-level individual and high-level strategic, collaborative reasoning in a single multi-agent architecture.

For a more thorough understanding of the implementation details involved, the reader is encouraged to study the algorithms described here in conjunction with the CMUnited-98 source code [11]. Other RoboCup researchers and multi-agent researchers in general should be able to benefit and build from the innovations represented therein.

References

1. David Andre, Emiel Corten, Klaus Dorer, Pascal Gugenberger, Marius Joldos, Johan Kummeneje, Paul Arthur Navratil, Itsuki Noda, Patrick Riley, Peter Stone, Romoichi Takahashi, and Travlex Yeap. Soccer server manual, version 4.0. Technical Report RoboCup-1998-001, RoboCup, 1998. At URL <http://ci.etl.go.jp/~noda/soccer/server/Documents.html>.
2. Mike Bowling, Peter Stone, and Manuela Veloso. Predictive memory for an inaccessible environment. In *Proceedings of the IROS-96 Workshop on RoboCup*, pages 28–34, Osaka, Japan, November 1996.
3. Hans-Diter Burkhard, Markus Hannebauer, and Jan Wendler. AT humboldt — development, practice and theory. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 357–372. Springer Verlag, Berlin, 1998.
4. Hiroaki Kitano, Yasuo Kuniyoshi, Itsuki Noda, Minoru Asada, Hitoshi Matsubara, and Eiichi Osawa. RoboCup: A challenge problem for AI. *AI Magazine*, 18(1):73–85, Spring 1997.
5. Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The RoboCup synthetic agent challenge 97. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 24–29, San Francisco, CA, 1997. Morgan Kaufmann.
6. Jean-Claude Latombe. *Robot Motion Planning*. Kluwer, 1991.
7. E. Pagello, F. Montesello, A. D’Angelo, and C. Ferrari. A reactive architecture for RoboCup competition. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 434–442. Springer Verlag, Berlin, 1998.
8. Peter Stone and Manuela Veloso. The CMUnited-97 simulator team. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 387–397. Springer Verlag, Berlin, 1998.
9. Peter Stone and Manuela Veloso. A layered approach to learning client behaviors in the RoboCup soccer server. *Applied Artificial Intelligence*, 12:165–188, 1998.
10. Peter Stone and Manuela Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 1999. To appear.
11. Peter Stone, Manuela Veloso, and Patrick Riley. CMUnited-98 source code, 1998. Accessible from <http://www.cs.cmu.edu/~pstone/RoboCup/CMUnited98-sim.html>.
12. Manuela Veloso, Michael Bowling, Sorin Achim, Kwun Han, and Peter Stone. The CMUnited-98 champion small robot team. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin, 1999.

StarCraft AI Competitions, Bots, and Tournament Manager Software

Michal Čertíký , David Churchill, Kyung-Joong Kim , Martin Čertíký, and Richard Kelly

Abstract—Real-time strategy games have become an increasingly popular test bed for modern artificial intelligence (AI) techniques. With this rise in popularity has come the creation of several annual competitions, in which AI agents (bots) play the full game of *StarCraft: Broodwar* by Blizzard Entertainment. The three major annual StarCraft AI Competitions are the Student StarCraft AI Tournament, the Computational Intelligence in Games competition, and the Artificial Intelligence and Interactive Digital Entertainment competition. In this paper, we will give an overview of the current state of these competitions, describe the bots that compete in them, and describe the underlying open-source Tournament Manager software that runs them.

Index Terms—Artificial intelligence, computational intelligence, education, computer science education, educational programs, learning, machine learning.

I. INTRODUCTION

REAL-TIME strategy (RTS) games are a genre of video games in which players manage economic and strategic tasks by gathering resources and building bases, increase their military power by researching new technologies and training units, and lead them into battle against their opponent(s). They serve as an interesting domain for artificial intelligence (AI) research and education, since they represent well-defined, complex adversarial systems [1] that pose a number of interesting AI challenges in the areas of planning, dealing with uncertainty, domain knowledge exploitation, task decomposition, spatial reasoning, and machine learning [2].

Manuscript received January 8, 2018; revised April 11, 2018, July 11, 2018, August 27, 2018, and November 9, 2018; accepted November 13, 2018. Date of publication November 26, 2018; date of current version September 13, 2019. This work was supported by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Science, ICT and Future Planning (2017R1A2B4002164). (Corresponding author: Michal Čertíký.)

M. Čertíký is with the Artificial Intelligence Center, Czech Technical University in Prague, Prague 16000, Czech Republic (e-mail: certicky@agents.fel.cvut.cz).

D. Churchill and R. Kelly are with the Department of Computer Science, Memorial University of Newfoundland, St. John's, NF A1C 5S7, Canada (e-mail: dave.churchill@gmail.com; richard.kelly@mun.ca).

K.-J. Kim was with the Department of Computer Science and Engineering, Sejong University, Seoul 143-747, South Korea. He is now with the School of Integrated Technology, Gwangju Institute of Science and Technology (GIST), Gwangju 61005, South Korea (e-mail: kjkim@gist.ac.kr).

M. Čertíký is with the Department of Cybernetics and Artificial Intelligence, Technical University in Košice, Košice 04001, Slovakia (e-mail: martin.certicky@stuk.sk).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TG.2018.2883499

Unlike turn-based abstract board games, such as chess and go, which can already be played by AI at super-human skill levels, RTS games are played in *real time*, meaning the state of the game will continue to progress even if the player takes no action, and so actions must be decided in fractions of a second. In addition, individual turns in RTS games (game frames) can consist of issuing simultaneous actions to hundreds of units at any given time [3]. This, together with their partially observable and nondeterministic nature, makes the RTS game genre one of the hardest game AI challenges today, attracting the attention of the academic research community, as well as commercial companies. For example, Facebook AI Research, Microsoft, and Google DeepMind have all recently expressed interest in using the most popular RTS game of all time: *StarCraft* as a test environment for their AI research [4].

Meanwhile, the academic community has been using *StarCraft* as a domain for AI research since the advent of the Brood War Application Programming Interface (BWAPI) in 2009 [5]. BWAPI allows programs to interact with the game engine directly to play autonomously against human players or against other programs (bots). The introduction of BWAPI gave rise to many scientific publications over the last eight years, addressing many subproblems inherent to RTS games. A comprehensive overview can be found in [2], [6], and [7].

In addition to AI research, *StarCraft* and BWAPI are often used for educational purposes as part of AI-related courses at universities, including the University of California at Berkeley (USA), Washington State University (USA), University of Alberta (Canada), Comenius University (Slovakia), Czech Technical University (Czech Republic), and most recently Technical University Delft (The Netherlands), where a new course titled “Multi-agent systems in *StarCraft*” has been opened for more than 200 students. The educational potential of *StarCraft* has recently been extended even further, when Blizzard Entertainment released the game entirely for free in April 2017.

Widespread use of *StarCraft* in research and education has led to a creation of three annual StarCraft AI competitions. The first competition was organized at the University of California, Santa Cruz in 2010 as part of the AAAI Artificial Intelligence and Interactive Digital Entertainment (AIIDE) conference program. The following year gave rise to two other annual competitions—the Student StarCraft AI Tournament (SSCAIT), organized as a standalone long-term event at Comenius University and Czech Technical University, and the CIG StarCraft AI competition colocated with the IEEE Computational Intelligence in Games (CIG) conference.

In this paper, we will talk about these three major StarCraft AI competitions and provide the latest updates on each of them, with the following three sections detailing the SSCAIT, AIIDE, and CIG StarCraft AI Competitions. We will then describe the state-of-the-art bots under active development that compete in these competitions, and the AI methods they use. Finally, we will introduce the open-source Tournament Manager software powering the competitions.

II. STUDENT STARCRAFT AI TOURNAMENT

The SSCAIT is the StarCraft AI competition with the highest number of total participants. There are three fundamental differences between SSCAIT and the remaining two competitions.

- 1) SSCAIT is an online-only event. Unlike AIIDE or CIG, it is not colocated with a scientific conference/event.
- 2) There are two phases of SSCAIT each year: a competitive *tournament phase*, lasting for up to four weeks and a *ladder phase* that runs for the rest each year. In other words, SSCAIT is live at all times with only a few short interruptions for maintenance.
- 3) Games are played one at a time and are publicly streamed live on Twitch.tv¹ and SmashCast.tv. The AIIDE and CIG competitions instead play as many games as possible at maximum speed, with no broadcast.

SSCAIT's *tournament phase* takes place every winter in late December and early January.

A. SSCAIT History

The first SSCAIT was organized in 2011 by Michal Čertický, as a part of the “Fundamentals of Artificial Intelligence” course at Comenius University, Bratislava, Slovakia. It started as a closed event, with 50 local students competing for extra points for their course evaluation. Since the event received a lot of positive feedback from the participants, the organizers decided to open it for the international public and for nonstudents next year (although the word “Student” remained in the competition name for historic reasons).

SSCAIT changed significantly over the course of 2012—both in terms of the format and technology behind it. The organizers implemented a collection of simple Python and AHK scripts that were able to run and evaluate the bot games automatically. This allowed for the creation of 24/7 bot *ladder* with online live stream, similar to the one available today.² The live-streamed ladder simplifies the bot debugging process (since bot authors can watch their creations play all kinds of AI opponents), encourages continuous development over the whole year and accelerates the growth of StarCraft AI research community.

The registration of new bots on the ladder was simplified in 2013 with the introduction of web-based user interface for bot creators. They can now upload new versions of their bots to the ladder at any time. In 2014, the custom automation scripts were replaced by Tournament Manager software, developed originally for AIIDE competition (details in Section VI), which



Fig. 1. SSCAIT live stream running in HD resolution and controlled by the custom observer script [9].

needed to be heavily modified in order to work with SSCAIT user and game databases and to support the ladder format. Further modifications and the introduction of the Dockerized multiplatform version of *StarCraft* [8] are planned soon.

Currently, SSCAIT is organized by the *Games and Simulations Research Group*³—part of Artificial Intelligence Center, Czech Technical University, Prague.

B. SSCAIT 2017/2018 Tournament & Ladder

The activity of bot programmers and the general public surrounding SSCAIT has grown considerably over the course of the past year, with new members to the organizing team making a number of improvements to the live stream, and better community engagement during the Ladder phase.

First, the ladder phase was updated, with SSCAIT introducing so-called “weekly reports.” Every weekend, there is a 1–2-h long segment of curated AI versus AI matches with insightful commentary on the live stream. Second, a voting system was implemented, allowing bot programmers and viewers to select which bots will play the next ladder match on live stream. This not only supports viewer engagement, but also greatly simplifies bot debugging process. Bot programmers can now quickly test their newest updates against specific opponents. This change might have contributed to the significant increase in bot update frequency. Approximately 5–6 bots are updated every day, in contrast to 0–2 updates per week in 2015.

Another update was the introduction of “minitournaments” to SSCAIT. These are easily configurable, irregular, and unofficial short competitions, taking up to one day. The format of these minitournaments and the selection of participants is usually up to the stream viewers and moderators. Visual quality of the stream was improved by updating the custom observer script [9], which now moves the camera fluently to the most interesting parts of the game in real time and displays SSCAIT-related information on top of the game. The stream was also upgraded to HD using a “resolution hack” (see Fig. 1). The overall number of stream views has increased to 376 920 views on Twitch.tv and additional 434 216 views on SmashCast.tv over the past 12 months. Two additional metrics were added to the ladder ranking system due to popular demand: ELO rating [10], which is used in adversarial games, such as chess, and “SSCAIT rank,” based on

¹<http://www.twitch.tv/sscait>

²The SSCAIT bot ladder was inspired by an older automated StarCraft bot ladder, available at that time at <http://bots-stats.krasio.com/>

³<http://gas.fel.cvut.cz/>

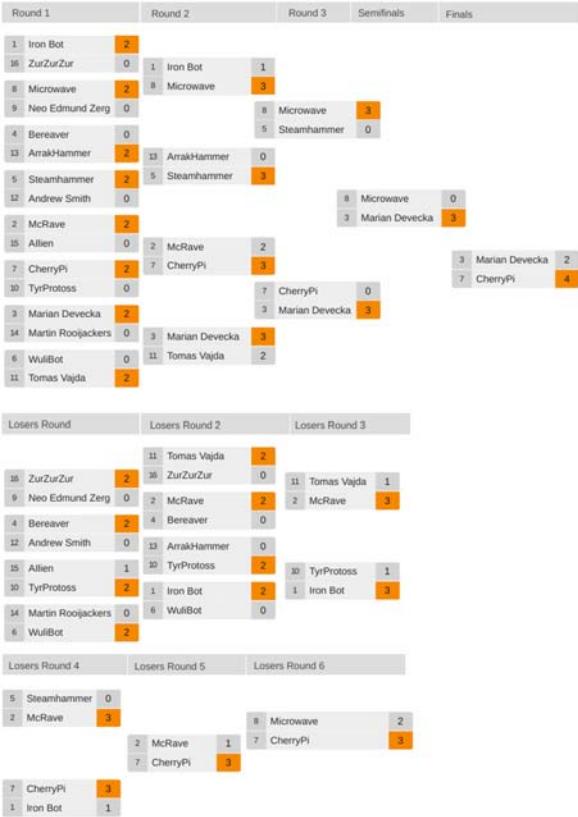


Fig. 2. SSCAIT 2017/18 mixed division double-elimination bracket.

the so-called “ICCUP ranking system,” typical for competitive *StarCraft*.

1) *SSCAIT Tournament Phase Updates*: The 2017/18 installment of SSCAIT’s tournament phase took place during four weeks at the end of December 2017 and beginning of January 2018 and sported 78 participants. The tournament was divided into the following two divisions.

a) *Student Division*: Round Robin tournament of 6006 games, where every bot played two games against every opponent. Only the bots created by individual students were considered “student” bots and were eligible for victory in this division. Other bots were tagged as “mixed-division” bots (they played the games, but could not win the student division title). Winners of the student division in 2017/2018 were as follows.

- 1) Wulibot, University of Southern California (USA) with 124 wins.
- 2) LetaBot (Martin Rooijackers), University of Maastricht (The Netherlands) with 109 wins.
- 3) Carsten Nielsen, Technical University of Denmark (Denmark) with 101 wins.

The student division of SSCAIT exists so that the students stand a chance of winning in the presence of more experienced, nonstudent participants, and team-created bots.

b) *Mixed Division*: After the student division ended, 16 bots with the most wins among all the participants were selected for the additional mixed division double elimination bracket, consisting of 30 matches (best of three, five, or seven games), which is shown in Fig. 2.

CherryPi, created by the Facebook AI Research team won the mixed division by beating KillerBot by Marian Devecka 4-2 in the finals. Interestingly, *CherryPi* encountered KillerBot earlier in the tournament (in winner’s round 3) and lost that match 0-3, dropping down to the losers bracket. The bot then managed to win the whole losers bracket, meet Killerbot again in the finals, and win by exploiting its experience from their previous games. More information about *CherryPi* can be found in Section V.

All the elimination bracket games were published as videos with commentary on SSCAIT YouTube channel⁴ and as replay files on SSCAIT website.⁵

III. ARTIFICIAL INTELLIGENCE AND INTERACTIVE DIGITAL ENTERTAINMENT

The AIIDE StarCraft AI Competition is the longest running annual StarCraft competition, and has been held every year since 2010 along with the AAAI Artificial Intelligence and Interactive Digital Entertainment conference. Unlike the CIG and SSCAIT competitions, the AIIDE competition requires (since 2011) that bot source code be submitted, and that the code will be published for download after the competition has finished. Running 24 hours a day for two weeks with games played at super-human speed, the competition is a single round-robin format with the winner being the bot with the highest win percentage when the time limit has been reached.

A. AIIDE StarCraft AI Competition History

The AIIDE StarCraft AI Competition was first run in 2010 by Ben Weber at the Expressive Intelligence Studio at the University of California, Santa Cruz, as a part of the AIIDE conference. In total, 26 entrants competed in four different game modes that varied from simple combat battles to the full game of *StarCraft*. As this was the first year of the competition, and little infrastructure had been created, each game of the tournament was run manually on two laptop computers and monitored by hand to record the results. Also, no persistent data were kept for bots to learn about opponents between matches. The 2010 competition had four different tournament categories in which to compete. Tournament 1 was a flat-terrain unit micromanagement battle consisting of four separate unit composition games. Tournament 2 was another microfocused game with nontrivial terrain. Tournament 3 was a tech-limited *StarCraft* game on a single known map with no fog-of-war enforced. Players were only allowed to choose the Protoss race, with no late game units allowed.

Tournament 4 was considered the main event, which involved playing the complete game of *StarCraft: Brood War* with fog-of-war enforced. The tournament was run with a random pairing double-elimination format with each match being best of five games. A map pool of five well-known professional maps were announced to competitors in advance, with a random map being chosen for each game. Tournament 4 was won by Overmind—a Zerg bot created by a large team from the University of

⁴<https://goo.gl/icbYdK>

⁵<http://sscaitournament.com/index.php?action=2017>

TABLE I
TOURNAMENT SETTINGS AND TOP 3 FINISHERS IN THE AIIDE, CIG, AND SSCAIT (STUDENT DIVISION) COMPETITIONS

Competition	Year	TM Software	Open-Source	Maps	1st Place	2nd Place	3rd Place
AIIDE	2010	Manual	Optional	Unannounced	Overmind	Krasi0	Chronos
AIIDE	2011	AIIDE TM	Forced	Announced	SkyNet	UAlbertaBot	Aiur
AIIDE	2012	AIIDE TM	Forced	Announced	SkyNet	Aiur	UAlbertaBot
AIIDE	2013	AIIDE TM	Forced	Announced	UAlbertaBot	SkyNet	Aiur
AIIDE	2014	AIIDE TM	Forced	Announced	IceBot	Ximp	LetaBot
AIIDE	2015	AIIDE TM	Forced	Announced	tscmoo	ZZZKBot	Overkill
AIIDE	2016	AIIDE TM	Forced	Announced	Iron	ZZZKBot	tscmoo
AIIDE	2017	AIIDE TM	Forced	Announced	ZZZKBot	PurpleWave	Iron
CIG	2011	Manual	Optional	Unannounced	SkyNet	UAlbertaBot	Xelnaga
CIG	2012	AIIDE TM	Optional	Unannounced	SkyNet	UAlbertaBot	Xelnaga
CIG	2013	Java-based TM	Optional	Unannounced	SkyNet	UAlbertaBot	Aiur
CIG	2014	AIIDE TM	Forced	Unannounced	IceBot	Ximp	LetaBot
CIG	2015	AIIDE TM	Optional	Unannounced	ZZZKBot	tscmoo	Overkill
CIG	2016	AIIDE TM	Forced	Announced	tscmoo	Iron	LetaBot
CIG	2017	AIIDE TM	Forced	Announced	ZZZKBot	tscmoo	PurpleWave
SSCAIT	2011	Custom	Optional	Announced	Roman Danielis	N/A	N/A
SSCAIT	2012/13	Custom	Optional	Announced	DementorBot	Marcin Bartnicki	UAlbertaBot
SSCAIT	2013/14	Custom	Optional	Announced	Ximp	WOPR	UAlbertaBot
SSCAIT	2014/15	AIIDE TM + Mod	Optional	Announced	LetaBot	WOPR	UAlbertaBot
SSCAIT	2015/16	AIIDE TM + Mod	Optional	Announced	LetaBot	Carsten Nielsen	UAlbertaBot
SSCAIT	2016/17	AIIDE TM + Mod	Optional	Announced	LetaBot	Wulibot	Zia Bot
SSCAIT	2017/18	AIIDE TM + Mod	Optional	Announced	Wulibot	LetaBot	Carsten Nielsen

California, Berkeley, who defeated the Terran bot Krasi0 by Krasimir Krastev in the finals.

From 2011 to 2016, the AIIDE competition was hosted by the University of Alberta, and was organized and run each year by David Churchill and Michael Buro. Due to the low number of entries to Tournaments 1, 2, and 3 from the 2010 AIIDE competition, it was decided that the AIIDE competition for 2011 would only consist of the full game of *StarCraft* (with the same rules as the 2010 Tournament 4), with no smaller micromanagement tournaments. The 2011 tournament rules were also updated so that all entrants must submit the source code of their bot and allow it to be published after the competition is over, which was done for several reasons. The first reason was to lower the barrier to entry for future competitions—since programming a StarCraft AI bot was very time consuming, future entrants could download and modify the source code of previous bots to save considerable effort. Another reason was to more easily prevent cheating—with thousands of games being played in the tournament, no longer could each game be manually inspected to detect if any cheating tactics were being employed, which would be more easily detected by inspecting the source code. The final reason was to help advance the state of the art in StarCraft AI by allowing future bots to borrow strategies and techniques of previous bots by inspecting their source code—ideally, all bots in future competitions should be at least as strong as the bots from the previous year.

Since the first competition was run by a single person on two laptops, games were played by manually starting the *StarCraft* game and creating and joining games by hand. As the physical demand was quite high, a simple random-pairing double-elimination tournament was played with approximately 60 games in total. This caused some negative feedback that this elimination-style tournament was quite dependent on pairing luck, so for the 2011 competition all chance was eliminated from the tournament by playing a round-robin style format. Playing a round-robin format requires far more games to be played, and it would no longer be possible to run each game manually. In the summer of 2011, the StarCraft AI Tournament Manager

software was written (see Section VI) that could automatically schedule and play round-robin tournaments of StarCraft on an arbitrary number of locally networked computers. The initial version of this software allowed for a total of 2340 games to be played in the same time period as the 2010 competition’s 60 games, with each bot playing each other bot a total of 30 times. There were ten total maps in the competition, chosen from expert human tournaments that were known to be balanced for each race, which were available for download several months in advance on the competition website. The AIIDE competition was modeled on human tournaments where the map pool and opponents are known in advance in order to allow for some expert knowledge and opponent modeling.

The 2012 AIIDE competition brought a major change to the functionality of the StarCraft AI Competitions: persistent file storage, which allowed the bots to learn throughout the course of the competition. The tournament managing software was updated so that each bot had access to a read folder and a write folder contained on a shared folder that was accessible to all the client machines. During each round bots could read from their “read” folder and write to their “write” folder, and at the end of each round robin (one game between each bot pairing on a single map) the contents of the write folder were copied to the read folder, giving access to all information written about previous rounds. This new functionality was used by several bots to implement strategy selection, in which their bot selected which of several strategies to use based on the results of previous rounds versus the same opponent, which typically increased their win rates over time during the competitions.

The AIIDE competitions between 2013 and 2016 did not have any major rule changes, and continued to use the same pool of ten maps for each competition. Competition appeared to stagnate between 2011 and 2013, with a relatively low number of entrants, and saw the same three bots (Aiur, SkyNet, and UAlbertaBot) trading first, second, and third place during these years. The 2014 to 2016 competitions, however, saw many new entries to the competition, with new bots taking the top three positions each year. Top three finishers of each year’s

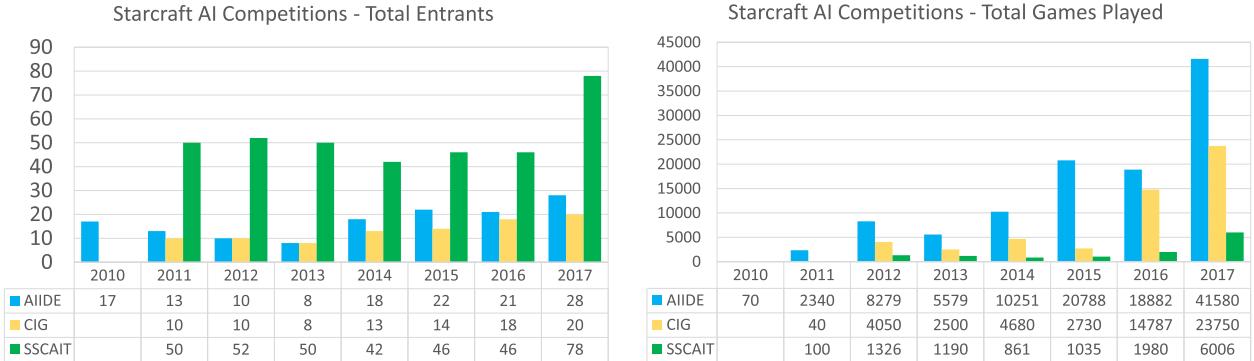


Fig. 3. Statistics for each of the three major annual StarCraft AI Competitions: AIIDE, CIG, and SSCAIT, since the first competition in 2010. Shown on the left is the number of total entrants for each competition, and on the right are the total number of games played in each competition.

TABLE II
RESULTS OF THE TOP EIGHT FINISHERS IN THE 2017 AIIDE COMPETITION

Bot	Race	Games	Win	Loss	Win %
ZZZKBot	Zerg	2966	2465	501	83.11
PurpleWave	Protoss	2963	2440	523	82.53
Iron	Terran	2965	2417	548	81.52
cpac	Zerg	2963	2104	859	71.01
Microwave	Zerg	2962	2099	863	70.86
CherryPi	Zerg	2966	2049	917	69.08
McRave	Protoss	2964	1988	976	67.07
Arrakhammer	Zerg	2963	1954	1009	65.95

competition are shown in Table I. Improvements to the tournament software and hardware infrastructure allowed for more games to be played each year are shown in Fig. 3.

B. 2017 AIIDE Competition

The 2017 AIIDE competition⁶ had a total of 28 competitors, and the round-robin games ran on 14 virtual machines for two weeks. In total, 110 rounds of round-robin play were completed, with each bot playing 2970 games for a total of 41 580 games. Any bot that achieved a win rate of 30% or higher in the 2016 competition that did not receive a new submission was automatically entered into the 2017 competition. No new rules or maps were used for the 2017 tournament that were not in place for the 2016 tournament. The AIIDE Tournament Manager software had been updated with new features, such as support for BWAPI version 4.2.0, and the ability for client machines to be listed with special properties, such as GPU computation ability. In combination with this update, a hardware upgrade for the tournament allowed for GPU computation support for any bots that required it, however, no 2017 bots used the feature. The 2017 competition had the closest top three finish of any competition yet, with the top three bots separated by less than 2% win rate, and 3rd–6th place bots also separated by less than 2% win rate. Statistics for the top eight finishers are shown in Table II.

The win percentage over time of the top three bots of the competition is shown in Fig. 4, and demonstrates the importance of implementing some form of opening modeling/learning over time. Although Iron (shown in green) led for the vast majority of the competition, it did not implement any form of learning over the course of the competition, and its win rate slowly

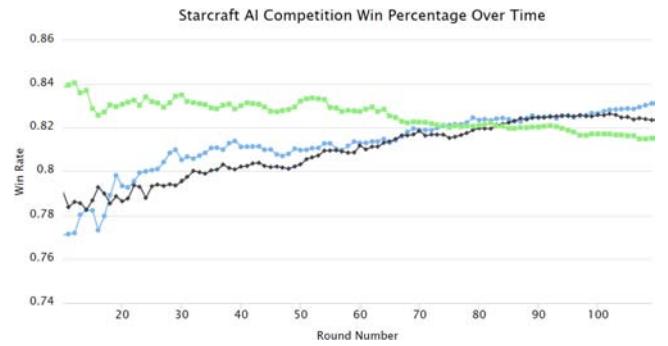


Fig. 4. Win percentage over time for the top three bots of the 2017 AIIDE StarCraft AI Competition. First place ZZZKBot shown in blue, second place PurpleWave in black, and third place Iron in green.

dropped over time. ZZZKBot (blue) and PurpleWave (black) implemented strategy selection learning, and their win rates slowly climbed to the point where they overtook Iron near round 85 of 110.

IV. COMPUTATIONAL INTELLIGENCE IN GAMES

The CIG StarCraft AI Competition has been a part of the program of the IEEE Computational Intelligence in Games conference since August 2011. Since the date of the CIG competition was usually just before the AIIDE competition, many of the bots submitted to both competitions ended up being nearly identical, therefore, the CIG competition has several rule differences with AIIDE in order to keep the results interesting. The biggest rule difference was that the CIG competition did not disclose which maps would be used for the competition, meaning that the bots could not use any hard-coded map information like they could for AIIDE.

A. CIG Competition History

The CIG conference is well known for hosting and organizing many AI-related competitions, such as the Mario AI Competition and the General Video Game Playing Competition, and in 2010, the first CIG StarCraft AI Competition was held. Organized by Johan Hagelback, Mike Preuss, and Ben Weber, the CIG 2010 competition was to have a single game mode similar to the tech-limited Tournament 3 from the AIIDE 2010

⁶<http://www.cs.mun.ca/~dchurchill/starcraftaicomp/2017/>

competition, but using the Terran race instead of the Protoss race. Unfortunately, the first year of the CIG competition had several technical problems, and no winner could be announced for the competition. Mike Preuss and his team members then successfully organized the CIG competition each year from 2011 to 2013. Since 2014, the Sejong University team (led by Kyung-Joong Kim) has been organizing the CIG competition at the IEEE CIG conference. In order to provide a more diverse competition, CIG rules and settings have changed each year, as shown in Table I.

Throughout its history, CIG has had multiple changes in the selection of tournament management software, open-source policy, and map pool announcement policy. The tournament management software (see Section VI) is used to distribute the matches over multiple machines on the network and to automate the competition operation. Although CIG organizers developed their own JAVA-based TM software, the AIIDE TM has been used for the competition since 2014 (see details in Section VI). Since 2016, CIG has enforced an open-source code policy, and all of the bots' source code are published after the competition. Unlike the AIIDE competition, the CIG map pool was not known to the participants before the competition to promote generalization ability of the entries. However, it was found that participants usually did not exploit map knowledge, and so since 2016, maps in the CIG competition have been announced in advance.

In the 2016 competition, the organizers introduced a second stage to the competition such that half of the entries advance to the second stage based on the win ratio of the first stage. This was inspired by the Simulated Car Racing Competition [11] that adopted a two-stage competition divided into a qualification stage and a main race. Since the single-pool round-robin format is based on win percentage only, it is important to get high average win ratio against all the opponents. The CIG organizers introduced two pools with the intent to reduce the chance that the top ranking bots win just by exploiting lower ranked bots to boost their win ratio. Bots were randomly split into two groups, and then the top half of bots from each group were brought together for a final stage group, to be played as round robin, with all learned information being deleted before the beginning of the final stage.

B. 2017 CIG Competition

In the 2017 CIG competition, the two-stage format was changed back to the single group format. The reason was that the participants did not seem to change their strategy to consider the two-stage tournament, and it did not seem to have much of an effect on the final results. In the future, CIG organizers consider adopting the SWISS-system widely used in the board game community. In this setting, a player does not play against all the other opponents. Instead, the participants are paired with the opponents with similar scores. Such systems usually produce final outcomes similar to round-robin while playing fewer total games.

In 2017, CIG organizers tried to play as many games as possible, and reached 125 rounds with 190 games per round, which

TABLE III
RESULTS OF THE 2017 CIG COMPETITION FINAL STAGE

Bot	Race	Games	Win	Loss	Win %
ZZZKBot	Zerg	1984	1628	356	82.06
tscmoo	Random	1992	1541	451	77.36
PurpleWave	Protoss	2021	1360	661	67.29
LetaBot	Terran	2026	1363	663	67.28
UAlbertaBot	Random	2005	1315	690	65.59
Overkill	Zerg	2024	1270	754	62.75

resulted in 23 750 games in the two-round format. Currently, AI bots often use multiple preprepared strategies and adapt them or their selection against specific opponents. The more games played during a tournament, the more experience allows them to learn which strategies are good against which opponents. As in the 2017 AIIDE competition, many bots implemented learning strategies that dramatically increased their win rates over time. Detailed results of the top 6 bots in the competition can be seen in Table III.

After the 2017 CIG competition, Sejong University organized a special event where human players were matched against the AI bots. The human players included one novice player (ladder rating around 1100), one middle-level player (around 1500), and a professional gamer: Byung-Gu Song. AI bots in the event were ZZZKBOT (winner of CIG 2017), tscmoo bot (2nd place in CIG 2017), and MJBOT, an AI bot specially designed against human players. MJBOT has been developed since June 2017 by Cognition Intelligence Laboratory to beat novice/middle-level human players. Each human player played a single game against each AI bot (nine games). The novice human player lost two games against ZZZKBOT and TSCMOO, but won the game against MJBOT, which was not able to finish the game due to a programming bug. In the next session, the middle-level human player lost all three games against the AI bots. Finally, the professional human player Byung-Gu Song won against all the AI bots.⁷ This suggests that the AI bots have a potential to compete against novice and middle-level players, but not professionals.

V. CURRENT STARCRAFT BOTS

Over the years, StarCraft AI competitions have motivated many individuals and groups to implement a variety of bots capable of playing complete *StarCraft* 1v1 games. The structure of most current bots emerges from the attempts to decompose the game into a hierarchy of smaller subproblems, such as higher level strategy, tactics, combat unit control, terrain analysis, and intelligence gathering. For more information about the individual challenges, refer to [2] and [6].

Bots vary in complexity, and while many are rule based, top-performing bots are now employing more sophisticated AI techniques, such as real-time search/planning, pretrained neural network controllers, and online learning during the competition games. In this section, we provide an overview of a selection of bots and discuss some of the AI approaches they implement. We only mention those bots that were active in one of the 2017

⁷<http://cilab.sejong.ac.kr/>

TABLE IV
OVERVIEW OF THE TECHNIQUES USED IN 2017 BOTS DESCRIBED IN SECTION V

Bot	Created	Rules	ML	HS	IO	SIM
CherryPi	2017	Yes	No	No	Yes	Yes
cpac	2017	Yes	No	Yes	No	Yes
ForceBot	2017	Yes	No	No	No	No
Iron	2016	Yes	No	No	No	No
KillAll	2017	Yes	Yes	Yes	No	Yes
Krasi0bot	2010	Yes	Yes	Yes	Yes	Yes
LetaBot	2014	Yes	No	Yes	Yes	Yes
McRave	2017	Yes	No	No	No	Yes
MegaBot	2016	Yes	No	No	Yes	Yes
PurpleWave	2017	Yes	No	Yes	Yes	Yes
StarcraftGP	2015	Yes	Yes	No	No	No
Steamhammer	2016	Yes	No	Yes	Yes	Yes
tscmoo	2015	Yes	Yes	Yes	Yes	Yes
UAlbertaBot	2010	Yes	No	Yes	Yes	Yes
ZZZKbot	2015	Yes	Yes	No	Yes	No

Shown are bots' creation year, and whether the bot uses any: "Rules" = rule-based systems, "ML" = machine learning, "HS" = heuristic search, "IO" = file I/O for learning during competitions, and "SIM" = simulations.

competitions, have recently been updated, and employ some more complex AI techniques (see Table IV).

- 1) *CherryPi*: CherryPi is a TorchCraft [12] Zerg bot developed by Facebook AI Research team. It is implemented as a collection of heterogeneous modules that can be added, removed, or configured via the command line. This design allows individual modules to be easily replaced with learning-powered modules, or to do narrow experiments using only a subset of them. The modules communicate by exchanging two kinds of data elements via the blackboard architecture: Key-value pairs and so-called UPC objects (Unit, Position, Command), which have a generic enough meaning to be loosely interpreted by other modules. In general, a UPC represents a probability distribution of units, a probability distribution of positions, and a single command. Build orders are represented as a set of prioritized requests pushed into a queue that fills in requirements and applies optimizations (like allocating resources for just-in-time construction). Fight-or-flight decisions are made by clustering units and running a combat simulation. CherryPi's combat simulation works similarly to the SparCraft simulation package (see UAlbertaBot) with a naive "attack-the-closest-target" policy for enemy behavior. A threat-aware path-finding is used to find the least dangerous routes to a safety. The selection of high-level strategy across multiple games is based on UCB1 algorithm, selecting from a fixed list of strategies against each race. The bot uses TorchCraft to communicate with BWAPI over TCP, which allows it to run on a different host than *StarCraft*.
- 2) *cpac*: A Zerg bot created by a 13-person team from China, cpac combines hard-coded rules with a multilayer perceptron network for unit production. The network is trained on state-action pairs extracted from a large data set of BroodWar games.⁸ The core of cpac bot is based on the bots UAlbertaBot and Steamhammer (see ahead).

⁸http://www.starcraftai.com/wiki/StarCraft_Brood_War_Data_Mining

- 3) *ForceBot*: ForceBot is a Zerg bot written in GOAL—an agent-based programming language designed on top of BWAPI for programming cognitive agents.⁹ Since the GOAL language is designed to implement multiagent systems, all of ForceBot's units have their own corresponding agent with specific beliefs and goals. Each agent more or less follows a rule-based AI pattern.
- 4) *Iron*¹⁰: Iron bot won the 2016 AIIDE competition, and is a decentralized multiagent system, with each unit controlled by a highly autonomous individual agent, able to switch between 25 behaviors. All its units share one simple aim: go to the main enemy base and destroy it. It often seems like a harasser bot, which is due to its units having mainly individual behavior. There are also so-called "expert" agents who autonomously recommend how resources should be spent and what units should be trained, based on heuristics.
- 5) *KillAll*: KillAll is a Zerg bot based on the Overkill bot by Sijia Xu, and most of its functionality is rule based. However, its production module uses Q-learning to select unit types to produce based on the current situation.
- 6) *Krasi0bot*: Krasi0bot has competed every year since 2010, and is still being actively developed. According to the author, it originally started as a rule-based bot, and currently makes some use of genetic algorithms, neural networks, and potential fields. As the bot is not open source, these details cannot be verified. Krasi0bot plays the Terran race, and is known for its strong defensive capabilities, and wide variety of strategies implemented.
- 7) *LetaBot*¹¹: LetaBot won the 2014, 2015, and 2016 SS-CAIT tournaments. It uses the Monte Carlo Tree Search to plan the movement of groups of units around the map. A similar approach has previously been used by the author of Nova bot, Alberto Uriarte [13]. It employs cooperative pathfinding for resource gathering and text mining to extract build orders from Liquipedia articles.
- 8) *McRave*: All the decisions of McRave bot are based on current enemy unit composition—there are no hardcoded tech choices. The bot also builds an opponent model and uses it to select build orders.
- 9) *MegaBot*¹²: For every game, MegaBot [14] chooses one of three approaches, each of which is implemented as a different bot (Skynet, Xelnaga, or NUSBot). Algorithm selection is modeled as a multiarmed bandit. At the beginning of the game, an algorithm is selected using epsilon-greedy strategy. After the game, the reward is perceived (+1, 0, and -1 for victory, draw, and loss, respectively) and the value of the selected algorithm is updated via an incremental version of recency-weighted exponential average (Q-learning update rule).

⁹<http://goalapl.atlassian.net/wiki/spaces/GOAL/>

¹⁰<http://bwem.sourceforge.net/Iron.html>

¹¹<https://github.com/MartinRooijackers/LetaBot>

¹²<https://github.com/andertavares/MegaBot>

- 10) *PurpleWave*¹³: The decision making of the PurpleWave bot is mainly based on hierarchical task networks. For micromanagement, it uses a hybrid squad/multiagent approach and nearest neighbors clustering. The bot then simulates the outcomes of battles and suggests tactics for squads by min–maxing tactical approaches by each side (e.g., “charge in,” “run away,” or “fight with workers”). In the end, each unit takes the tactical suggestion under advisement, but behaves independently. The units choose between approximately two dozen simple, reusable stateless behaviors. The bot heuristics include using potential fields for unit movement. Strategies are chosen based on results of previous games, race, map, and number of starting positions. It has a graph of strategy selections, like opening build orders paired with midgame transitions and late-game compositions.
- 11) *StarCraftGP*: StarcraftGP is the first StarCraft metabot—a program that autonomously creates a program that autonomously plays *StarCraft* [15]. Currently, StarcraftGP v0.1 is using (Linear) Genetic Programming and it is able to directly write C++ code. Its first creations: Salsa and Tequila, have been the first bots not directly written by a human to participate in international competitions.
- 12) *Steamhammer*¹⁴: The Zerg bot Steamhammer, developed by Jay Scott, and its random-race version Randomhammer are based on UAlbertaBot (see ahead), employing sophisticated combat simulation to predict the outcome of battles. The bots also use hierarchical reactive control for the units. For Protoss and Terran production, Randomhammer uses branch-and-bound search, whereas Zerg is currently rule based.
- 13) *tscmoo*¹⁵: tscmoo won the 2015 AIIDE and 2016 CIG competitions. The bot uses no external libraries: it has its own combat simulation code to predict the outcome of battles, it does not use BWTA¹⁶ to analyze the terrain and it even has its own threat-aware path-finding for individual units. The bot is one of the most strategically diverse, and selects among its many strategies based on their success in previous games. Recent versions of the bot experimented with recurrent neural networks for high-level strategy and build-order decisions.
- 14) *UAlbertaBot*¹⁷: UAlbertaBot has competed in every major StarCraft AI Competition since 2010, and won the 2013 AIIDE competition. UAlbertaBot uses a dynamic heuristic-search-based Build-Order Search System to plan all its build orders in real time, as well as a *StarCraft* combat simulation system called SparCraft for estimating the outcome of in-game battles. The bot uses the results of previous games against specific opponents to choose a strategy to implement at the beginning of each game, with each strategy being defined in an external

¹³<https://github.com/dgant/PurpleWave>

¹⁴<http://satirist.org/ai/starcraft/steamhammer/>

¹⁵<https://github.com/tscmoo>

¹⁶<https://bitbucket.org/auriarte/bwta2>

¹⁷<https://github.com/davechurchill/ualbertabot>

- JSON configuration file. Its development has focused on its ease of use and modification, and as such has become the basis of more than ten other bots in current competitions, including LetaBot, Overkill, and Steamhammer. In 2017, UAlbertaBot became CommandCenter,¹⁸ the first bot capable of playing both *BroodWar* and *StarCraft 2*.
- 15) *ZZZKbot*¹⁹: ZZZKBot, a Zerg bot developed by Chris Coxe, was the winner of the 2017 AIIDE and CIG competitions. Its overall strategy implements four simple one-base rush strategies: four-pool, Speedlings, Hydralisks, and Mutalisks. If the initial rush does not end the game, the bot switches to either Mutalisks or Guardians for the late game, while researching upgrades for all its units. The bot records win/loss information for each opponent, and uses this information to pick the best combination of strategy parameters for future games in a rule-based manner. The majority of the bots rules for unit control and micromanagement are simple rule-based behaviors based on expert knowledge prioritization.

We can observe over the past few years that StarCraft AI bots are indeed getting stronger overall. In the AIIDE and CIG competitions, several bots from previous years are intentionally left in the next year to serve as a benchmark for progress, and we see each time that these benchmark bots do worse over time. Also, expert players and enthusiasts observe replays and note how they feel bots have gotten better or worse over time. Most notably, many of these expert players feel that the bots have been gradually adapting a more “standard” playing style than earlier bots, who traditionally did one strategy such as a rush, but not much else. More modern bots have developed mid and even late game tactics that were not seen in earlier rushing bots. Overall, bots seem to be getting better at army composition, build-order selection, building placement, and overall game strategy.

While the strongest bots currently play at an amateur human level, expert players have noted that they still appear to be weak in a few key areas. Most importantly, bots still seem quite weak at adapting their strategies dynamically during a match in response to information gained about their opponent. The majority of bots employ a playbook of several strategies that they choose from at the start of a match and follow through to the end of the game, with only a few bots attempting to dramatically change things if the opponent does something unexpected. This means that bots are still quite vulnerable to human players who are more easily able to change strategies and tactics as a game goes on. Current bots also seem quite vulnerable to the human ability to quickly identify bot patterns and behavior, and exploit this quickly during a match. For example, one human player during a human versus machine match noted that one bot unit would chase his Zergling when it got close to the bots’ units, and proceeded to run the Zergling next to the bot army, and then lead the bot on a wild goose chase throughout the entire map. The entire time, the bot may have been reasoning that its army could win the fight against the single Zergling unit while not realizing that the human was just buying time until its army

¹⁸<https://github.com/davechurchill/commandcenter/>

¹⁹<https://github.com/chriscoxe/ZZZKBot>



Fig. 5. Tournament Manager software running in a computer laboratory for the CIG competition, which distributes bot games to multiple machines.

was ready for the final attack. This also illustrates one of the biggest challenges in all of artificial intelligence: understanding the long-term effects of actions that have delayed rewards. An expert human is able to quickly understand that they are being exploited in such a way, and that it will have negative effects down the road, and is able to stop the behavior. This long-term vision that is so intuitive to humans remains a problem for current RTS AI.

VI. TOURNAMENT MANAGER SOFTWARE

All three StarCraft AI competitions covered in this paper use the same open-source tool (with different parameters) to automate the bot games. The tool is called StarCraft AI Tournament Manager (TM)²⁰ and was created/maintained by David Churchill and Richard Kelly for the AIIDE competition. It allows tournaments of thousands of bot versus bot games to be played automatically on any number of physical or virtual machines, as shown in Fig. 5. The original version of the software was created in 2011 for the AIIDE StarCraft AI Competition. The CIG StarCraft AI Competition has used the TM software since 2012, and SCCAIT has used a modified version of the Tournament Manager since 2014.

The TM software supports both round robin and one versus all tournaments for testing one bot against others. The server stores all bot and map files, as well as results and replay files generated by the *BroodWar* clients. Files are sent over Java sockets between the server and client machines. The Tournament Manager supports bots using different versions of BWAPI, and support for new versions can easily be added, allowing bots written in any version of BWAPI to play in the same tournament. Each client machine currently requires an installation of *StarCraft: BroodWar* version 1.16.1.

The TM software uses a server-client architecture distributed over multiple physical or virtual machines connected via LAN, with one machine acting as a server (coordinating the matchups and processing results) and any number of other machines acting as clients (running the bots and *StarCraft*). The tournament manager is written entirely in Java. The clients should run on Windows machines due to system requirements of *StarCraft*.

²⁰<http://github.com/davechurchill/StarcraftAITournamentManager>

File Actions		Tournament progress:		Server uptime:		Filter Log:		
Client	Status	Game / Round #	Self	Enemy	Map	Duration	Win	Properties
192.168.1.102	SENDING	223/1	Unbeatbot	Tit	C'Destinatio...	3:11	Victory	
192.168.1.103	READY	223/1	Unbeatbot	Unbeatbot	C'Destinatio...	0:00		
192.168.1.104	SENDING	223/1	Tit	Unbeatbot	C'Destinatio...	0:00		
192.168.1.112	STARTING	224/1				11s		
192.168.1.113	RUNNING	225/1	Ortakia	Unbeatbot	C'Destinatio...	2:45		
192.168.1.105	RUNNING	226/1	Unbeatbot	Ortakia	C'Destinatio...	2:45		
192.168.1.103	RUNNING	227/1	Ortakia	Unbeatbot	C'Destinatio...	2:15		
192.168.1.114	STARTING	224/1				11s		

```

Aug 17, 13:59:20 Receiving Game: (224/1) Ortakia vs. Unbeatbot
Aug 17, 13:59:21 Sending Message to Client 192.168.1.112: C'mex Unbeatbot true (224/1)
Aug 17, 13:59:21 Sending Message to Client 192.168.1.112: REQUIRED_DIR Required_BWAPI_374.zip 2628 kb
Aug 17, 13:59:21 Sending Message to Client 192.168.1.112: BOT_DIR C'mex 590 kb
Aug 17, 13:59:21 Sending Message to Client 192.168.1.114: C'mex Unbeatbot false (224/1)
Aug 17, 13:59:21 Sending Message to Client 192.168.1.114: REQUIRED_DIR Required_BWAPI_420.zip 2058 kb
Aug 17, 13:59:21 Sending Message to Client 192.168.1.114: BOT_DIR C'mex 590 kb
Aug 17, 13:59:21 Sending Message to Client 192.168.1.112: Start The Game Already!
Aug 17, 13:59:21 Sending Message to Client 192.168.1.114: Start The Game Already!
Aug 17, 13:59:23 Not enough clients to start next game. Waiting for more free clients.
Aug 17, 13:59:29 Receiving Replay: (223/1)
Aug 17, 13:59:30 Receiving Replay: (223/1)
Aug 17, 13:59:34 Message from Client 192.168.1.104: REPLAY 59 kb
Aug 17, 13:59:35 Message from Client 192.168.1.102: REPLAY 59 kb

```

Fig. 6. Tournament Manager server GUI.

whereas the server is fully platform independent. All the data sent and received are compressed and passed through Java sockets over TCP/IP.

A. Server

When running the software, one machine acts as a server for the tournament. The server machine holds central repository of all the bot files, their custom I/O data files, cumulative results, and replay files. The server monitors and controls each client remotely and displays the tournament's progress in real time via the server GUI (see Fig. 6), also writing the current results to an HTML file every few seconds.

The server program has a threaded component that monitors client connections and detects disconnections, maintaining a current list of clients. Each client is in one of the following states at all times (column "Status" in Fig. 6).

- 1) READY: Client is ready to start a game of *StarCraft*.
- 2) STARTING: Client has started the *StarCraft* LAN lobby but the match has not yet begun.
- 3) RUNNING: A game of *StarCraft* is currently in progress on the client machine.
- 4) SENDING: Client has finished the game and is sending results and data back to the server.

The server's main scheduling loop tries to schedule the next game from the games list every 2 s. A new game is started whenever both of these conditions are true:

- 1) two or more Clients are in READY state;
- 2) no clients are in STARTING state.

Once these two conditions are met, the server sends the required bot files, BWAPI version used by the bots, map file, and DLL injector to the client machines. The state of those clients is then set to STARTING.

Each client is handled by a separate thread in the server, and if the client is STARTING, RUNNING, or SENDING, it sends periodic status updates back to the server once per second for remote monitoring. Data updates include current game time, time-out information, map name, game ID, etc. When a client finishes a game, it also sends the results, I/O data files created by the bots and replay files, which are all stored on the server. This process is repeated until the tournament has finished.

Shutting down the server via the GUI will cause all the client games to stop and all clients to shut down and properly clean

up remote machines. The tournament can be resumed upon relaunching the server as long as the results file, games list, and settings files do not change. If the server is shut down with games in progress (results not yet received by the server), those games will be rescheduled and played again. The server GUI can send commands to the client machines, take screenshots of the client machine desktops, and remove clients from the tournament. Individual client machines can be added and removed without stopping the current tournament.

B. Client

The client software can be run on as many machines as needed. After an initial setup of the client machine (installing *StarCraft*, required libraries, etc.) the client software connects to the server machine via TCP/IP and awaits instructions.

The client machine will stay idle until it receives instructions from the server that a game should be run. Once the client receives the instructions and required files from the server, it ensures that no current *StarCraft* processes are running, ensures a clean working *StarCraft* directory, records a current list of the running processes on the client machine, writes the BWAPI settings file, and starts the game. When the game starts, a custom BWAPI Tournament Module DLL is injected into the *StarCraft* process. It updates a special GameState file on the hard drive every few frames—this is used to monitor the current state of the running *StarCraft* game. The client software watches this file to check for various conditions, such as bot time outs, crashes, game frame progression, and game termination. While the game is running, the client also sends the contents of the GameState file to the server once per second for centralized monitoring.

Once the game has ended or was terminated for any reason, the results of the game, replay files, and bot's I/O data files are sent to the server. After this, the client shuts down any processes on the machine, which were not running when the game began, to prevent crashed proxy bots or stray threads from hogging system resources from future games. *StarCraft* is shut down, the machine is cleaned of any files written during the previous game, and the client state is reverted to READY.

Since 2017, client machines can be labeled with custom properties such as extra ram or GPU, and bots can be labeled with matching custom requirements. Only clients that have all the requirements of a bot will be used for hosting that bot, and clients with special properties will be reserved to be used last to increase their availability for bots requiring them. The TM software also supports both DLL-based bots and bots with their own executable file that interface with BWAPI. This server-client architecture has allowed tens of thousands of games to be played in AI competitions each year.

VII. CONCLUSION

In this paper, we have given an overview of the three major annual *StarCraft* AI competitions, introduced the open-source software powering them, and described some of the top performing bot participants. As shown in Fig. 3, each year, participation in these competitions has continued to rise, as well as the number of games played between bots in the competitions. In the past

two to three years, the bots in these competitions have become more strategically complex and functionally robust, employing a range of state-of-the-art AI techniques from the fields of heuristic search, machine learning, neural networks, and reinforcement learning. For many researchers, *StarCraft* AI competitions continue to be the environment of choice to showcase state-of-the art techniques for real-time strategic AI systems. With the increase in participation in *StarCraft* AI competitions, combined with the surge in activity in the field of RTS AI from hobbyist programmers, academic institutions, and industry researchers, we are hopeful that we may see a *StarCraft* AI capable of challenging human experts within the next few years.

REFERENCES

- [1] M. Buro, “Call for AI research in RTS games,” in *Proc. 4th Workshop Challenges Game AI*, 2004, pp. 139–142.
- [2] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, “A survey of real-time strategy game AI research and competition in *StarCraft*,” *IEEE Trans. Comput. Intell. AI Games*, vol. 5, no. 4, pp. 293–311, Dec. 2013.
- [3] M. Buro and D. Churchill, “Real-time strategy game competitions,” *AI Mag.*, vol. 33, no. 3, 2012, Art. no. 106.
- [4] E. Gibney, “What Google’s winning go algorithm will do next,” *Nature*, vol. 531, no. 7594, pp. 284–285, 2016.
- [5] A. Heinermann, “Broodwar API,” 2013. [Online]. Available: <https://github.com/bwapi/bwapi>
- [6] D. Churchill *et al.*, “StarCraft bots and competitions,” *Encyclopedia of Computer Graphics and Games*. New York, NY, USA: Springer, 2016.
- [7] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, “RTS AI: Problems and techniques,” in *Encyclopedia of Computer Graphics and Games*. New York, NY, USA: Springer, 2015.
- [8] J. Malý, M. Šustr, and M. Čertický, “Multi-platform version of StarCraft: Brood war in a Docker container: Technical report,” 2018, arXiv:1801.02193.
- [9] B. P. Mattsson, T. Vajda, and M. Čertický, “Automatic observer script for StarCraft: Brood War bot games (technical report),” 2015, arXiv:1505.00278.
- [10] A. E. Elo, *The Rating of Chessplayers, Past and Present*. New York, NY, USA: Arco, 1978.
- [11] D. Loiacono *et al.*, “The 2009 simulated car racing championship,” *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 2, pp. 131–147, Jun. 2010.
- [12] G. Synnaeve *et al.*, “Torchcraft: A library for machine learning research on real-time strategy games,” 2016, arXiv:1611.00625.
- [13] A. Uriarte and S. Ontanón, “High-level representations for game-tree search in RTS games,” in *Proc. 10th Artif. Intell. Interactive Digit. Entertainment Conf.*, 2014.
- [14] A. Tavares, H. Azpúrua, A. Santos, and L. Chaimowicz, “Rock, paper, starCraft: Strategy selection in real-time strategy games,” in *Proc. 12th Artif. Intell. Interactive Digit. Entertainment Conf.*, 2016, pp. 93–99.
- [15] P. García-Sánchez, A. Tonda, A. M. Mora, G. Squillero, and J. Merelo, “Towards automatic starCraft strategy generation using genetic programming,” in *Proc. IEEE Conf. Comput. Intell. Games*, 2015, pp. 284–291.



Michal Čertický received the M.Sc. degree in cognitive science and Ph.D. degree in computer science.

He is currently Senior Researcher with the Artificial Intelligence Center, Czech Technical University, Prague, Czech Republic.

His research interests include machine learning and agent-based modeling. Since 2011, he has also been involved in computer game AI research. He is the Founder of the Student *StarCraft* AI tournament.



David Churchill received the M.Sc. degree in computer science and the Ph.D. degree in computing science.

He is currently an Assistant Professor in computer science with the Memorial University of Newfoundland, St. John's, NF, Canada.

Since 2011, he has been organizing and running the AIIDE StarCraft AI Competition. He is also the author of UAlbertaBot and CommandCenter StarCraft AI agents.



Martin Čertický is currently working toward the Ph.D. degree in the field of AI at the Department of Cybernetics and Artificial Intelligence, Technical University of Košice, Košice, Slovakia.

His research is focused on optimizing user experience in electronic entertainment and the use of artificial intelligence in video games.



Kyung-Joong Kim received the B.S., M.S., and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 2000, 2002, and 2007, respectively.

He is currently an Associate Professor with the Department of Computer Science and Engineering, Sejong University, Seoul, South Korea. His research interests include artificial intelligence, game, and robotics.



Richard Kelly is currently working toward the Graduate degree at the Memorial University of Newfoundland, St. John's, NF, Canada.

Since 2016, he has been a co-organizer of the AIIDE StarCraft AI Competition.

Team-Partitioned, Opaque-Transition Reinforcement Learning

Peter Stone and Manuela Veloso *

Computer Science Department

Carnegie Mellon University

Pittsburgh, PA 15213 {pstone,veloso}@cs.cmu.edu

<http://www.cs.cmu.edu/~pstone/~mmv>

In *RoboCup-98: Robot Soccer World Cup II*,
M. Asada and H. Kitano (eds.), 1999. Springer Verlag, Berlin.

Abstract

In this paper, we present a novel multi-agent learning paradigm called team-partitioned, opaque-transition reinforcement learning (TPOT-RL). TPOT-RL introduces the concept of using action-dependent features to generalize the state space. In our work, we use a learned action-dependent feature space. TPOT-RL is an effective technique to allow a team of agents to learn to cooperate towards the achievement of a specific goal. It is an adaptation of traditional RL methods that is applicable in complex, non-Markovian, multi-agent domains with large state spaces and limited training opportunities. Multi-agent scenarios are opaque-transition, as team members are not always in full communication with one another and adversaries may affect the environment. Hence, each learner cannot rely on having knowledge of future state transitions after acting in the world. TPOT-RL enables teams of agents to learn effective policies with very few training examples even in the face of a large state space with large amounts of hidden state. The main responsible features are: dividing the learning task among team members, using a very coarse, action-dependent feature space, and allowing agents to gather reinforcement directly from observation of the environment. TPOT-RL is fully implemented and has been tested in the robotic soccer domain, a complex, multi-agent framework. This paper presents the algorithmic details of TPOT-RL as well as empirical results demonstrating the effectiveness of the developed multi-agent learning approach with learned features.

1 Introduction

Reinforcement learning (RL) is an effective paradigm for training an artificial agent to act in its environment in pursuit of a goal. RL techniques rely on the premise that an agent's action policy affects its overall reward over time. As surveyed in [Kaelbling, Littman, & Moore, 1996], several popular RL techniques use dynamic programming to enable a single agent to learn an effective control policy as it traverses a stationary (Markovian) environment.

Dynamic programming requires that agents have or learn at least an approximate model of the state transitions resulting from its actions. Q-values encode future rewards attainable from neighboring states. A single agent can keep track of state transitions as its actions move it from state to state.

This paper focusses on teams of agents learning to collaborate towards a common goal in adversarial environments. While agents can still affect their reward through their actions, they can no longer necessarily track the team's state transitions: teammates and opponents affect and experience state transitions that are completely opaque to the agent. For example, an information agent may broadcast a message without any knowledge of who receives and reacts to it. In such *opaque-transition* settings, Q-values cannot relate to neighboring states (which are unknown), but must still reflect real-world long-term reward resulting from chosen actions.

While opaque-transition settings eliminate the possibility of using dynamic programming, they permit parallel learning among teammates. Since agents do not track state transitions, they can each explore a separate partition of the state space without any knowledge of state values in other partitions. This *team-partitioned* characteristic speeds

*This research is sponsored in part by the DARPA/RL Knowledge Based Planning and Scheduling Initiative under grant number F30602-95-1-0018. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of the U. S. Government.

up learning by reducing the learning task of each agent. Nevertheless, the challenge of learning in a non-stationary (non-Markovian) environment remains.

This general setup builds upon the robotic soccer framework, which has been the substrate of our work. In more detail in this setup, agents' actions are *chained*, i.e., a single agent's set of actions allows the agent to select which other agent will be chained after in the pursuit to achieve a goal. A single agent cannot control directly the full achievement of a goal, but a chain of agents will. In robotic soccer the chaining of actions corresponds to passing a ball between the different agents. There are a variety of other such examples, such as information agents that may communicate through message passing. (These domains are for example in contrast with grid world domains in which a single agent moves from some initial location to some final goal location, domains where agents take actions in parallel though also possibly in coordination - two robots executing tasks in parallel, and game domains where the rules of the game enforce an agent and its opponent to alternate actions.) Because of our chaining of agents and the corresponding lack of control of single agents to fully achieve goals, we call these domains team-partitioned.

In addition, we assume that agents do not know the state that the world will be in after an action is selected, as another agent will *continue* the path to the goal. Adversarial agents can also intercept the chain and take control of the game. The domain becomes therefore opaque-transition. In short, We identify a way to do RL when the learning cannot even observe what state the team enters next, but the agent can use a reward function that captures a medium-to long-term result of the whole ensemble's learning.

In this paper we present team-partitioned, opaque-transition reinforcement learning (TPOT-RL). TPOT-RL can learn a set of effective policies with very few training examples. It relies on action-dependent dynamic features which coarsely generalize the state space. While feature selection is often a crucial issue in learning systems, our work uses a previously *learned* action-dependent feature. We empirically demonstrate the effectiveness of TPOT-RL in a multi-agent, adversarial environment, and show that the previously learned action-dependent feature can improve the performance of TPOT-RL.

The remainder of the paper is organized as follows. Section 2 formally presents the TPOT-RL algorithm. Section 3 details an implementation of TPOT-RL in the simulated robotic soccer domain with extensive empirical results presented in Section 4. Section 5 relates TPOT-RL to previous work and Section 6 concludes.

2 Team-Partitioned, Opaque-Transition RL

Formally, a policy is a mapping from a state space S to an action space A such that the agent using that policy executes action a whenever in state s . At the coarsest level, when in state s , an agent compares the expected, long-term rewards for taking each action $a \in A$, choosing an action based on these expected rewards. These expected rewards are learned through experience.

Designed to work in real-world domains with far too many states to handle individually, TPOT-RL constructs a smaller feature space V using action-dependent feature functions. The expected reward $Q(v, a)$ is then computed based on the state's corresponding entry in feature space.

In short, the policy's mapping from S to A in TPOT-RL can be thought of as a 3-step process:

State generalization: the state s is generalized to a feature vector v using the state generalization function $f : S \mapsto V$.

Value function learning: the feature vector v is used to estimate the expected reward for taking each possible action using the changing (learned) value function $Q : (V, A) \mapsto \mathbb{R}$.

Action selection: an action a is chosen for execution and its real-world reward is used to further update Q .

While these steps are common in other RL paradigms, each step has unique characteristics in TPOT-RL.

2.1 State Generalization

TPOT-RL's state generalization function $f : S \mapsto V$ relies on a unique approach to constructing V . Rather than discretizing the various dimensions of S , it uses *action-dependent* features. In particular, each possible action a_i is evaluated locally based on the current state of the world using a fixed function $e : (S, A) \mapsto U$. Unlike Q , e does not produce the expected long-term reward of taking an action; rather, it classifies the likely short-term effects of the action. For example, if actions sometimes succeed and sometimes fail to achieve their intended effects, e could indicate something of the following form: if selected, action a_7 is (or is not) likely to produce its intended effects.

In the multi-agent scenario, other than one output of e for each action, the feature space V also involves one coarse component that partitions the state space S among the agents. If the size of the team is m , then the partition function is $P : S \mapsto M$ with $|M| = m$. In particular, if the set of possible actions $A = \{a_0, a_1, \dots, a_{n-1}\}$, then

$$\begin{aligned} f(s) &= \langle e(s, a_0), e(s, a_1), \dots, e(s, a_{n-1}), P(s) \rangle, \text{ and so} \\ V &= U^{|A|} \times M. \end{aligned}$$

Thus, $|V| = |U|^{|A|} * m$. Since TPOT-RL has no control over $|A|$ or m , and since the goal of constructing V is to have a small feature space over which to learn, TPOT-RL will be more effective for small sets U .

This state generalization process reduces the complexity of the learning task by constructing a small feature space V which partitions S into m regions. Each agent need learn how to act only within its own partition. Nevertheless, for large sets A , the feature space can still be too large for learning, especially with limited training examples. Our particular action-dependent formulation allows us to reduce the effective size of the feature space in the value-function-learning step. Choosing features for state generalization is generally a hard problem. While TPOT-RL does not specify the function e , our work uses a previously-learned dynamic feature function.

2.2 Value Function Learning

As we have seen, TPOT-RL uses action-dependent features. Therefore, we can assume that the expected long-term reward for taking action a_i depends only on the feature value related to action a_i . That is,

$$Q(\langle e(s, a_1), \dots, e(s, a_{n-1}), P(s) \rangle, a_i) = Q(\langle e(s', a_1), \dots, e(s', a_{n-1}), P(s') \rangle, a_i)$$

whenever $e(s, a_i) = e(s', a_i)$ and $P(s) = P(s')$. In other words, if $f(s) = v$, $Q(v, a_i)$ depends entirely upon $e(s, a_i)$ and is independent of $e(s, a_j)$ for all $j \neq i$.

Without this assumption, since there are $|A|$ actions possible for each feature vector, the value function Q has $|V| * |A| = |U|^{|A|} * |A| * m$ independent values. Under this assumption, however, the Q-table has at most $|A| * |U| * m$ entries: for each action possible from each position, there is only one relevant feature value. Therefore, even with only a small number of training examples available, we can treat the value function Q as a lookup-table without the need for any complex function approximation. To be precise, Q stores one value for every possible combination of action a , $e(s, a)$, and $P(s)$.

For example, Table 1 shows the entire feature space for one agent's partition of the state space when $|U| = 3$ and $|A| = 2$. There are $|U|^{|A|} = 3^2$ different entries in feature space with 2 Q-values for each entry: one for each possible action. $|U|^{|A|} * m$ is much smaller than the original state space for any realistic problem, but it can grow large quickly, particularly as $|A|$ increases. However, notice in Table 1 that, under the assumption described above, there are only $3 * 2$ independent Q-values to learn, reducing the number of free variables in the learning problem by 67% in this case.

$e(s, a_0)$	$e(s, a_1)$	$Q(v, a_0)$	$Q(v, a_1)$
u_0	u_0	$q_{0,0}$	$q_{1,0}$
u_0	u_1	$q_{0,0}$	$q_{1,1}$
u_0	u_2	$q_{0,0}$	$q_{1,2}$
u_1	u_0	$q_{0,1}$	$q_{1,0}$
u_1	u_1	$q_{0,1}$	$q_{1,1}$
u_1	u_2	$q_{0,1}$	$q_{1,2}$
u_2	u_0	$q_{0,2}$	$q_{1,0}$
u_2	u_1	$q_{0,2}$	$q_{1,1}$
u_2	u_2	$q_{0,2}$	$q_{1,2}$

⇒

$e(s, a_i)$	$Q(v, a_0)$	$Q(v, a_1)$
u_0	$q_{0,0}$	$q_{1,0}$
u_1	$q_{0,1}$	$q_{1,1}$
u_2	$q_{0,2}$	$q_{1,2}$

Table 1: A sample Q-table for a single agent when $|U| = 3$ and $|A| = 2$: $U = \{u_0, u_1, u_2\}$, $A = \{a_0, a_1\}$. $q_{i,j}$ is the estimated value of taking action a_i when $e(s, a_i) = u_j$. Since this table is for a single agent, $P(s)$ remains constant.

The Q-values learned depend on the agent's past experiences in the domain. In particular, after taking an action a while in state s with $f(s) = v$, an agent receives reward r and uses it to update $Q(v, a)$ as follows:

$$Q(v, a) = Q(v, a) + \alpha(r - Q(v, a)) \quad (1)$$

Since the agent is not able to access its teammates' internal states, future team transitions are completely opaque from the agent's perspective. Thus it cannot use dynamic programming to update its Q-table. Instead, the reward r comes directly from the observable environmental characteristics—those that are captured in S —over a maximum

number of time steps t_{lim} after the action is taken. The reward function $R : S^{t_{lim}} \mapsto \mathbb{R}$ returns a value at some time no further than t_{lim} in the future. During that time, other teammates or opponents can act in the environment and affect the action's outcome, but the agent may not be able to observe these actions. For practical purposes, it is crucial that the reward function is only a function of the observable world *from the acting agent's perspective*. In practice, the range of R is $[-Q_{max}, Q_{max}]$ where Q_{max} is the reward for immediate goal achievement.

The reward function, including t_{lim} and Q_{max} , is domain-dependent. One possible type of reward function is based entirely upon reaching the ultimate goal. In this case, an agent charts the actual (long-term) results of its policy in the environment. However, it is often the case that goal achievement is very infrequent. In order to increase the feedback from actions taken, it is useful to use an internal reinforcement function, which provides feedback based on intermediate states towards the goal. We use this internal reinforcement approach in our work.

2.3 Action Selection

Informative action-dependent features can be used to reduce the free variables in the learning task still further at the action-selection stage if the features themselves discriminate situations in which actions should not be used. For example, if whenever $e(s, a_i) = u_1$, a_i is not likely to achieve its expected reward, then the agent can decide to ignore actions with $e(s, a_i) = u_1$.

Formally, consider $W \subseteq U$ and $B(s) \subseteq A$ with $B(s) = \{a \in A | e(s, a) \in W\}$. When in state s , the agent then chooses an action from $B(s)$, either randomly when exploring or according to maximum Q-value when exploiting. Any exploration strategy, such as Boltzman exploration, can be used over the possible actions in $B(s)$. In effect, W acts in TPOT-RL as an action filter which reduces the number of options under consideration at any given time. Of course, exploration at the filter level can be achieved by dynamically adjusting W .

$e(s, a_0)$	$e(s, a_1)$	$Q(v, a_0)$	$Q(v, a_1)$
u_0	u_0	$q_{0,0}$	$q_{1,0}$
u_0	u_1	$q_{0,0}$	—
u_0	u_2	$q_{0,0}$	$q_{1,2}$
u_1	u_0	—	$q_{1,0}$
u_1	u_1	—	—
u_1	u_2	—	$q_{1,2}$
u_2	u_0	$q_{0,2}$	$q_{1,0}$
u_2	u_1	$q_{0,2}$	—
u_2	u_2	$q_{0,2}$	$q_{1,2}$

$e(s, a_0)$	$e(s, a_1)$	$Q(v, a_0)$	$Q(v, a_1)$
u_0	u_0	—	—
u_0	u_1	—	—
u_0	u_2	—	$q_{1,2}$
u_1	u_0	—	—
u_1	u_1	—	—
u_1	u_2	—	$q_{1,2}$
u_2	u_0	$q_{0,2}$	—
u_2	u_1	$q_{0,2}$	—
u_2	u_2	$q_{0,2}$	$q_{1,2}$

Table 2: The resulting Q-tables when (a) $W = \{u_0, u_2\}$, and (b) $W = \{u_2\}$.

For example, Table 2, illustrates the effect of varying $|W|$. In the rare event that $B(s) = \emptyset$, i.e. $\forall a_i \in A, e(s, a_i) \notin W$, either a random action can be chosen, or rough Q-value estimates can be stored using sparse training data. This condition becomes rarer as $|A|$ increases. For example, with $|U| = 3, |W| = 1, |A| = 2$ as in Table 2(b), $4/9 = 44.4\%$ of feature vectors have no action that passes the W filter. However, with $|A| = 8$ only $256/6561 = 3.9\%$ of feature vectors have no action that passes the W filter. If $|W| = 2$ and $|A| = 8$, only 1 of 6561 feature vectors fails to pass the filter. Thus using W to filter action selection can reduce the number of free variables in the learning problem without significantly reducing the coverage of the learned Q-table.

By using action-dependent features to create a coarse feature space, and with the help of a reward function based entirely on individual observation of the environment, TPOT-RL enables team learning in a multi-agent, adversarial environment even when agents cannot track state transitions.

3 TPOT-RL Applied to a Complex Multi-Agent Learning Task

Our research has been focussed on multi-agent learning in complex, collaborative and adversarial environments. Our general approach, called *layered learning*, is based on the premise that realistic domains are too complex for learning mappings directly from sensor inputs to actuator outputs. Instead, intermediate domain-dependent skills should be learned in a bottom-up hierarchical fashion [Stone & Veloso, 1998a]. We implemented TPOT-RL as the current highest layer of a layered learning system in the RoboCup soccer server [Noda, Matsubara, & Hiraki, 1996].

The soccer server used at RoboCup-97 [Kitano *et al.*, 1997] is a much more complex domain than has previously been used for studying multi-agent policy learning. With 11 players on each team controlled by separate processes; noisy, low-level, real-time sensors and actions; limited communication; and a fine-grained world state model including hidden state, the RoboCup soccer server provides a framework in which machine learning can improve performance. Newly developed multi-agent learning techniques could well apply in real-world domains.

A key feature of the layered learning approach is that learned skills at lower levels are used to train higher-level skills. For example, we used a neural network to help players learn how to intercept a moving ball. Then, with all players using the learned interception behavior, a decision tree (DT) enabled players to estimate the likelihood that a pass to a given field location would succeed. Based on almost 200 continuous-valued attributes describing teammate and opponent positions on the field, players learned to classify the pass as a likely success (ball reaches its destination or a teammate gets it) or likely failure (opponent intercepts the ball). Using the C4.5 DT algorithm [Quinlan, 1993], the classifications were learned with associated confidence factors. The learned behaviors proved effective both in controlled testing scenarios [Stone & Veloso, 1998a, Stone & Veloso, 1998c] and against other previously-unseen opponents in an international tournament setting [Kitano *et al.*, 1997].

These two previously-learned behaviors were both trained off-line in limited, controlled training situations. They could be trained in such a manner due to the fact that they only involved a few players: ball interception only depends on the ball's and the agent's motions; passing only involves the passer, the receiver, and the agents in the immediate vicinity. On the other hand, deciding where to pass the ball during the course of a game requires training in game-situations since the value of a particular action can only be judged in terms of how well it works when playing with particular teammates against particular opponents. For example, passing backwards to a defender could be the right thing to do if the defender has a good action policy, but the wrong thing to do if the defender is likely to lose the ball to an opponent.

Although the DT accurately predicts whether a player can execute a pass, it gives no indication of the strategic value of doing so. But the DT reduces a detailed state description to a single continuous output. It can then be used to drastically reduce the complex state and provide great generalization. In this work we use the DT as the crucial action-dependent feature function e in TPOT-RL.

3.1 State Generalization Using a Learned Feature

In the soccer example, we applied TPOT-RL to enable each teammate to simultaneously learn a high-level action policy. The policy is a function that determines what an agent should do *when it has possession of the ball*.¹ The input of the policy is the agent's perception of the current world state; the output is a target destination for the ball in terms of a location on the field, e.g. the opponent's goal. In our experiment, each agent has 8 possible actions as illustrated in Figure 1(a). Since a player may not be able to tell the results of other players' actions, or even when they can act, the domain is opaque-transition.

A team formation is divided into 11 positions ($m = 11$), as also shown in Figure 1(a) [Stone & Veloso, 1998c]. Thus, the partition function $P(s)$ returns the player's position. Using our layered learning approach, we use the previously trained DT as e . Each possible pass is classified as either a likely success or a likely failure with a confidence factor. Outputs of the DT could be clustered based on the confidence factors. In our experiments, we cluster into only two sets indicating success and failure. Therefore $|U| = 2$ and $V = U^8 \times \{PlayerPositions\}$ so $|V| = |U|^{|A|} * m = 2^8 * 11$. Even though each agent only gets about 10 training examples per 10-minute game and the reward function shifts as teammate policies improve, the learning task becomes feasible. This feature space is immensely smaller than the original state space, which has more than 22^{10^9} states.² Since e indicates the likely success or failure of each possible action, at action-selection time, we only consider the actions that are likely to succeed ($|W|=1$). Therefore, each player learns 8 Q-values, with a total of 88 learned by the team as a whole. Even with sparse training and shifting concepts, such a learning task is tractable.

¹In the soccer server there is no actual perception of having “possession” of the ball. Therefore we consider the agent to have possession when it is within kicking distance.

²Each of the 22 players can be in any of $680*1050*3600$ (x, y, θ) locations, not to mention the player velocities and the ball position and velocity.

3.2 Internal Reinforcement through Observation

As in any RL approach, the reward function plays a large role in determining what policy is learned. One possible reward function is based entirely upon reaching the ultimate goal. Although goals scored are the true rewards in this domain, such events are very sparse. In order to increase the feedback from actions taken, it is useful to use an internal reinforcement function, which provides feedback based on intermediate states towards the goal. Without exploring the space of possible such functions, we created one reward function R .

R gives rewards for goals scored. However, players also receive rewards if the ball goes out of bounds, or else after a fixed period of time t_{lim} based on the ball's average lateral position on the field. In particular, when a player takes action a_i in state s such that $e(s, a_i) = u$, the player records the time t at which the action was taken as well as the x coordinate of the ball's position at time t , x_t . The reward function R takes as input the observed ball position over time t_{lim} (a subset of $S^{t_{lim}}$) and outputs a reward r . Since the ball position over time depends also on other agents' actions, the reward is stochastic and non-stationary. Under the following conditions, the player fixes the reward r :

1. if the ball goes out of bounds (including a goal) at time $t + t_o$ ($t_o < t_{lim}$);
2. if the ball returns to the player at time $t + t_r$ ($t_r < t_{lim}$);
3. if the ball is still in bounds at time $t + t_{lim}$.

In case 1, the reward r is based on the value r_o as indicated in Figure 1(b): $r = \frac{r_o}{1 + (\phi - 1) * t_o / t_{lim}}$. Thus, the farther in the future the ball goes out of bounds (i.e. the larger t_o), the smaller the absolute value of r . This scaling by time is akin to the discount factor used in Q-learning. We use $t_{lim} = 30sec.$ and $\phi = 10$.

In cases 2 and 3, the reward r is based on the average x-position of the ball over the time t to the time $t + t_r$ or $t + t_{lim}$. Over that entire time span, the player samples the x-coordinate of the ball at fixed, periodic intervals and computes the average x_{avg} over the times at which the ball position is known. Then if $x_{avg} > x_t$, $r = \phi * \frac{x_{avg} - x_t}{x_{og} - x_t}$ where x_{og} is the x-coordinate of the opponent goal (the right goal in Figure 1(b)). Otherwise, if $x_{avg} \leq x_t$, $r = -\phi * \frac{x_t - x_{avg}}{x_t - x_{lg}}$ where x_{lg} is the x-coordinate of the learner's goal.³ Thus, the reward is the fraction of the available field by which the ball was advanced, on average, over the time-period in question. Note that a backwards pass can lead to positive reward if the ball then moves forward in the near future.

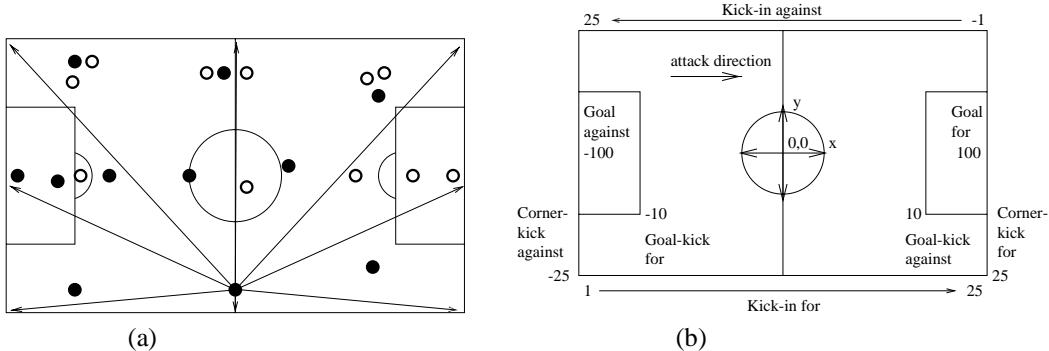


Figure 1: (a) The black and white dots represent the players attacking the right and left goals respectively. Arrows indicate a single player's action options when in possession of the ball. The player kicks the ball towards a fixed set of markers around the field, including the corner flags and the goals. (b) The component r_o of the reward function R based on the circumstances under which the ball went out of bounds. For kick-ins, the reward varies linearly with the x position of the ball.

The reward r is based on direct environmental feedback. It is a domain-dependent internal reinforcement function based upon heuristic knowledge of progress towards the goal. Notice that it relies solely upon the player's own impression of the environment. If it fails to notice the ball's position for a period of time, the internal reward is affected. However, players can track the ball much more easily than they can deduce the internal states of other players as they would have to do were they to determine future team state transitions.

As teammates learn concurrently, the concept to be learned by each individual agent changes over time. We address this problem by gradually increasing exploitation as opposed to exploration in all teammates and by using a learning rate $\alpha = .02$ (see Equation 1). Thus, even though we are averaging several reward values for taking an action in a

³The parameter ϕ insures that intermediate rewards cannot override rewards for attaining the ultimate goal.

given state, each new example accounts for 2% of the updated Q-value: rewards gained while teammates were acting more randomly are weighted less heavily.

4 Results

Empirical testing has demonstrated that TPOT-RL can effectively learn multi-agent control policies with very few training instances in a complex, dynamic domain. Figure 2(a) plots cumulative goals scored by a learning soccer team playing against an otherwise equally-skilled team that passes to random destinations over the course of a single long run equivalent in time to 160 10-minute games. In this experiment, and in all the remaining ones, the learning agents start out acting randomly and with empty Q-tables. Over the course of the games, the probability of acting randomly as opposed to taking the action with the maximum Q-value decreases linearly over periods of 40 games from 1 to .5 in game 40, to .1 in game 80, to point .01 in game 120 and thereafter. As apparent from the graph, the team using TPOT-RL learns to vastly outperform the randomly passing team. During this experiment, $|U| = 1$, thus rendering the function e irrelevant: the only relevant state feature is the player’s position on the field.

A key characteristic of TPOT-RL is the ability to learn with minimal training examples. During the run graphed in Figure 2(a), the 11 players got an average of 1490 action-reinforcement pairs over 160 games. Thus, players only get reinforcement an average of 9.3 times each game, or less than once every minute. Since each player has 8 actions from which to choose, each is only tried an average of 186.3 times over 160 games, or just over once every game. Under these training circumstances, very efficient learning is clearly needed.

TPOT-RL is effective not only against random teams, but also against goal-directed, hand-coded teams. For testing purposes, we constructed an opponent team which plays with all of its players on the same side of the field, leaving the other side open as illustrated by the white team in Figure 1. The agents use a hand-coded policy which directs them to pass the ball up the side of the field to the forwards who then shoot on goal. The team periodically switches from one side of the field to the other. We call this team the “switching team.”

Were the opponent team to always stay on the same side of the field, the learning team could always advance the ball up the other side of the field without any regard for current player positions. Thus, TPOT-RL could be run with $|U| = 1$, which renders e inconsequential. Indeed, we verified empirically that TPOT-RL is able to learn an effective policy against such an opponent using $|U| = 1$.

Against the switching team, a player’s best action depends on the current state. Thus a feature that discriminates among possible actions dynamically can help TPOT-RL. Figure 2(b) compares TPOT-RL with different functions e and different sets W when learning against the switching team.

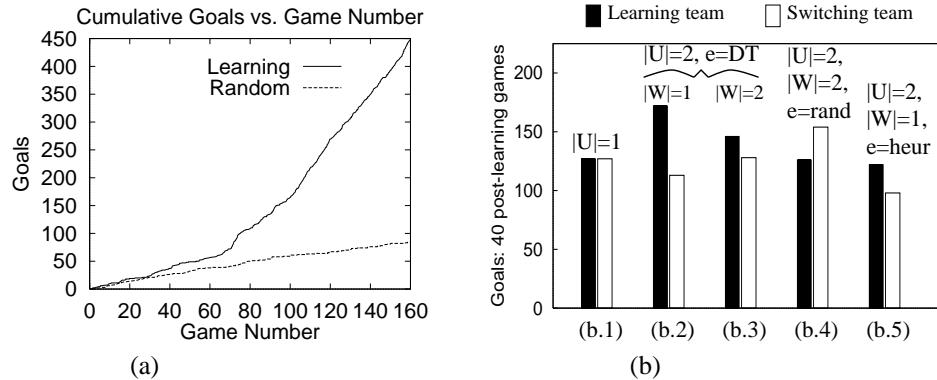


Figure 2: (a) Cumulative goals scored by a learning team playing against a randomly passing team. The independent variable is the number of 10-minute game intervals that have elapsed. (b) The results after training of 5 different TPOT-RL runs against the switching team.

With $|U| = 1$ (Figure 2(b.1)), the learning team is unable to capture different opponent states since each player has only one Q-value associated with each possible action, losing 139-127 (cumulative score over 40 games after 160 games of training). Recall that if $|U| = 1$ the function e cannot discriminate between different classes of states: we end up with a poor state generalization.

In contrast, with the previously trained DT classifying passes as likely successes or failures ($e = \text{DT}$) and TPOT-RL filtering out the failures, the learning team wins 172-113 (Figure 2(b.2)). Therefore the learned pass-evaluation feature is able to usefully distinguish among possible actions and help TPOT-RL to learn a successful action policy. The DT also helps learning when $W = U$ (Figure 2(b.3)), but when $|W| = 1$ performance is better.

Figure 2(b.4) demonstrates the value of using an informative action-dependent feature function e . When a random function $e = \text{rand}$ is used, TPOT-RL performs noticeably worse than when using the DT. For the random e we show $|W| = 2$ because it only makes sense to filter out actions when e contains useful information. Indeed, when $e = \text{rand}$ and $|W| = 1$, the learning team performs even worse than when $|W| = 2$ (it loses 167-60). The DT even helps TPOT-RL more than a hand-coded heuristic pass-evaluation function ($e = \text{heur}$) based on one that we successfully used on our real robot team [Veloso *et al.*, 1998] (Figure 2(b.5)).

Final score is the ultimate performance measure. However, we examined learning more closely in the best case experiment ($e = \text{DT}$, $|W| = 1$ — Figure 2(b.2)). Recall that the learned feature provides no information about which actions are *strategically* good. TPOT-RL must learn that on its own. To test that it is indeed learning to advance the ball towards the opponent’s goal (other than by final score), we calculated the number of times each action was predicted to succeed by e and the number of times it was actually selected by TPOT-RL after training. Throughout the entire team, the 3 of 8 actions towards the opponent’s goal were selected $6437/9967 = 64.6\%$ of the times that they were available after filtering. Thus TPOT-RL learns that it is, in general, better to advance the ball towards the opponent’s goal.

To test that the filter was eliminating action choices based on likelihood of failure we found that 39.6% of action options were filtered out when $e = \text{DT}$ and $|W| = 1$. Out of 10,400 actions, it was never the case that all 8 actions were filtered out.

5 Discussion and Related Work

TPOT-RL brings together several techniques that have been proposed theoretically or tested in simple domains. This section highlights the components of TPOT-RL and relates them to previous work.

Typical RL paradigms update the value of a state-action pair based upon the value of the subsequent state (or state distribution). As presented in [Kaelbling, Littman, & Moore, 1996], the typical update function in Q-learning is $Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_a Q(s', a) - Q(s, a))$ where s' is the state next reached after executing action a in state s and γ is the discount factor. While characterized as “model-free” in the sense that the agent need not know the transition function $T : (S, A) \mapsto S$, these paradigms assume that the agent can observe the subsequent state that it enters.

However, the vast amount of hidden state coupled with the multi-agent nature of this domain make such a paradigm impossible for the following reasons. Having only local world-views, agents cannot reliably discern when a teammate is able to take an action. Furthermore, even when able to notice that a teammate is within kicking distance of the ball, the agent certainly cannot tell the feature values for the teammate’s possible actions. Worse than being model-free, multi-agent RL must deal with the inability to even track the team’s state trajectory. Thus we use Equation 1, which doesn’t rely on knowing s' .

Notice that the opaque-transition characteristic also does not fit into the partially observable Markov decision process (POMDP) framework [Kaelbling, Cassandra, & Littman, 1994]. While POMDPs deal with hidden state, they do assume that the agent at least knows when it has transitioned to a new state and may act again.

The construction of feature space V can have a huge effect on the nature of Q . For example, in [Salustowicz, Wiering, & Schmidhuber, 1998], a grid-like discretization is used for V . Since too many states result for a lookup-table, a neural network is used as the value function approximator. This approach is shown not to work very well, and the authors conclude that a more complex function approximator might work better. In contrast, we take the approach of using a smaller feature space and the simplest possible evaluation function: a lookup-table. We can do so by relying on our layered learning approach to learn the state generalization function.

The use of machine learning in multi-agent systems has recently been receiving a good deal of attention. For an extensive survey, see [Stone & Veloso, 1997]. This section highlights the work most related to TPOT-RL.

The internal reinforcement in the reward function R is similar to Mataric’s progress estimators [Mataric, 1994]. There, the short-term real-world effects of actions are used as an intermediate reward to help robots reach the ultimate goal location. Mataric’s *conditions* also play a similar role to the features used here, reducing the size of the evaluation function domain. This work was in a completely collaborative, as opposed to our adversarial, setting.

Previous multi-agent reinforcement learning systems have typically dealt with much simpler tasks than the one presented here. Littman uses Markov games to learn stochastic policies in a very abstract version of 1-on-1 robotic soccer [Littman, 1994]. There have also been a number of studies of multi-agent reinforcement learning in the pursuit domain [Arai, Miyazaki, & Kobayashi, 1997, Tan, 1993]. In this domain, four predators chase a single prey in a small grid-like world.

Also in a predator-like task, [Zhao & Schmidhuber, 1996] uses a single run to deal with the opponents' shifting policies and ignore the opponents' policies just as we do. The effects of opponent actions are captured in the reward function.

Another team-partitioned, opaque transition domain is network routing as considered in [Boyan & Littman, 1994]. Each network node is considered as a separate agent which cannot see a packet's route beyond its own action. A major difference between that work and our own is that neighboring nodes send back their own value estimates whereas we assume that agents do not even know their neighboring states. Thus unlike TPOT-RL agents, the nodes are able to use dynamic programming.

In other soccer systems, there have been a number of learning techniques that have been explored. However, most have learned low-level, individual skills as opposed to team-based policies [Asada *et al.*, 1996, Stone & Veloso, 1998b]. Interestingly, [Luke *et al.*, 1998] uses genetic programming to evolve team behaviors from scratch as opposed to our layered learning approach.

6 Conclusion

TPOT-RL is an adaptation of RL to non-Markovian multi-agent domains with opaque transitions, large state spaces, hidden state and limited training opportunities. The fully implemented algorithm has been successfully tested in simulated robotic soccer, such a complex multi-agent domain with opaque transitions. TPOT-RL facilitates learning by partitioning the learning task among teammates, using coarse, action-dependent features, and gathering rewards directly from environmental observations. Our work uses a learned feature within TPOT-RL.

TPOT-RL represents the third and currently highest layer within our ongoing research effort to construct a complete learning team using the layered learning paradigm. As advocated by layered learning, it uses the previous learned layer—an action-dependent feature—to improve learning. TPOT-RL can learn against any opponent since the learned values capture opponent characteristics. The next learned layer could learn to choose among learned team policies based on characteristics of the current opponent. TPOT-RL represents a crucial step towards completely learned collaborative and adversarial strategic reasoning within a team of agents.

Acknowledgements

We thank Justin Boyan, Andrew Moore, and Astro Teller for helpful comments and suggestions.

References

- [Arai, Miyazaki, & Kobayashi, 1997] Arai, S.; Miyazaki, K.; and Kobayashi, S. 1997. Generating cooperative behavior by multi-agent reinforcement learning. In *Sixth European Workshop on Learning Robots*.
- [Asada *et al.*, 1996] Asada, M.; Noda, S.; Tawaratumida, S.; and Hosoda, K. 1996. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning* 23:279–303.
- [Boyan & Littman, 1994] Boyan, J. A., and Littman, M. L. 1994. Packet routing in dynamically changing networks: A reinforcement learning approach. In Cowan, J. D.; Tesauro, G.; and Alspector, J., eds., *Advances In Neural Information Processing Systems 6*. Morgan Kaufmann Publishers.
- [Kaelbling, Cassandra, & Littman, 1994] Kaelbling, L. P.; Cassandra, A. R.; and Littman, M. L. 1994. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*.

- [Kaelbling, Littman, & Moore, 1996] Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.
- [Kitano *et al.*, 1997] Kitano, H.; Kuniyoshi, Y.; Noda, I.; Asada, M.; Matsubara, H.; and Osawa, E. 1997. RoboCup: A challenge problem for AI. *AI Magazine* 18(1):73–85.
- [Littman, 1994] Littman, M. L. 1994. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, 157–163. San Mateo, CA: Morgan Kaufman.
- [Luke *et al.*, 1998] Luke, S.; Hohn, C.; Farris, J.; Jackson, G.; and Hendler, J. 1998. Co-evolving soccer softbot team coordination with genetic programming. In Kitano, H., ed., *RoboCup-97: Robot Soccer World Cup I*, 398–411. Berlin: Springer Verlag.
- [Mataric, 1994] Mataric, M. J. 1994. Interaction and intelligent behavior. MIT EECS PhD Thesis AITR-1495, MIT AI Lab.
- [Noda, Matsubara, & Hiraki, 1996] Noda, I.; Matsubara, H.; and Hiraki, K. 1996. Learning cooperative behavior in multi-agent environment: a case study of choice of play-plans in soccer. In *PRICAI'96: Topics in Artificial Intelligence (Proc. of 4th Pacific Rim International Conference on Artificial Intelligence, Cairns, Australia)*, 570–579.
- [Quinlan, 1993] Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.
- [Salustowicz, Wiering, & Schmidhuber, 1998] Salustowicz, R. P.; Wiering, M. A.; and Schmidhuber, J. 1998. Learning team strategies: Soccer case studies. *Machine Learning*. To appear.
- [Stone & Veloso, 1997] Stone, P., and Veloso, M. 1997. Multiagent systems: A survey from a machine learning perspective. Technical Report CMU-CS-97-193, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.
- [Stone & Veloso, 1998a] Stone, P., and Veloso, M. 1998a. A layered approach to learning client behaviors in the RoboCup soccer server. *Applied Artificial Intelligence* 12:165–188.
- [Stone & Veloso, 1998b] Stone, P., and Veloso, M. 1998b. Towards collaborative and adversarial learning: A case study in robotic soccer. *International Journal of Human-Computer Studies* 48(1):83–104.
- [Stone & Veloso, 1998c] Stone, P., and Veloso, M. 1998c. Using decision tree confidence factors for multiagent control. In Kitano, H., ed., *RoboCup-97: Robot Soccer World Cup I*. Berlin: Springer Verlag. 99–111.
- [Tan, 1993] Tan, M. 1993. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, 330–337.
- [Veloso *et al.*, 1998] Veloso, M.; Stone, P.; Han, K.; and Achim, S. 1998. The CMUnited-97 small-robot team. In Kitano, H., ed., *RoboCup-97: Robot Soccer World Cup I*. Berlin: Springer Verlag. 242–256.
- [Zhao & Schmidhuber, 1996] Zhao, J., and Schmidhuber, J. 1996. Incremental self-improvement for life-time multi-agent reinforcement learning. In *Proceedings of the 4th International Conference of Simulation of Adaptive Behaviors (SAB 4)*, 363–372. MIT Press.

Reinforcement Learning: A Survey

Leslie Pack Kaelbling

Michael L. Littman

Computer Science Department, Box 1910, Brown University

Providence, RI 02912-1910 USA

LPK@CS.BROWN.EDU

MLITTMAN@CS.BROWN.EDU

Andrew W. Moore

Smith Hall 221, Carnegie Mellon University, 5000 Forbes Avenue

Pittsburgh, PA 15213 USA

AWM@CS.CMU.EDU

Abstract

This paper surveys the field of reinforcement learning from a computer-science perspective. It is written to be accessible to researchers familiar with machine learning. Both the historical basis of the field and a broad selection of current work are summarized. Reinforcement learning is the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment. The work described here has a resemblance to work in psychology, but differs considerably in the details and in the use of the word "reinforcement." The paper discusses central issues of reinforcement learning, including trading off exploration and exploitation, establishing the foundations of the field via Markov decision theory, learning from delayed reinforcement, constructing empirical models to accelerate learning, making use of generalization and hierarchy, and coping with hidden state. It concludes with a survey of some implemented systems and an assessment of the practical utility of current methods for reinforcement learning.

1. Introduction

Reinforcement learning dates back to the early days of cybernetics and work in statistics, psychology, neuroscience, and computer science. In the last five to ten years, it has attracted rapidly increasing interest in the machine learning and artificial intelligence communities. Its promise is beguiling—a way of programming agents by reward and punishment without needing to specify *how* the task is to be achieved. But there are formidable computational obstacles to fulfilling the promise.

This paper surveys the historical basis of reinforcement learning and some of the current work from a computer science perspective. We give a high-level overview of the field and a taste of some specific approaches. It is, of course, impossible to mention all of the important work in the field; this should not be taken to be an exhaustive account.

Reinforcement learning is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment. The work described here has a strong family resemblance to eponymous work in psychology, but differs considerably in the details and in the use of the word "reinforcement." It is appropriately thought of as a class of problems, rather than as a set of techniques.

There are two main strategies for solving reinforcement-learning problems. The first is to search in the space of behaviors in order to find one that performs well in the environment. This approach has been taken by work in genetic algorithms and genetic programming,

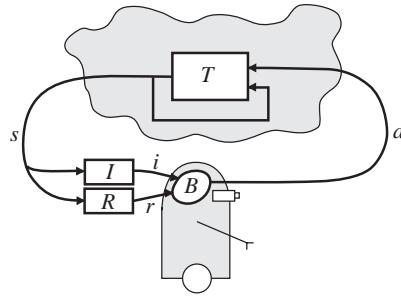


Figure 1: The standard reinforcement-learning model.

as well as some more novel search techniques (Schmidhuber, 1996). The second is to use statistical techniques and dynamic programming methods to estimate the utility of taking actions in states of the world. This paper is devoted almost entirely to the second set of techniques because they take advantage of the special structure of reinforcement-learning problems that is not available in optimization problems in general. It is not yet clear which set of approaches is best in which circumstances.

The rest of this section is devoted to establishing notation and describing the basic reinforcement-learning model. Section 2 explains the trade-off between exploration and exploitation and presents some solutions to the most basic case of reinforcement-learning problems, in which we want to maximize the immediate reward. Section 3 considers the more general problem in which rewards can be delayed in time from the actions that were crucial to gaining them. Section 4 considers some classic model-free algorithms for reinforcement learning from delayed reward: adaptive heuristic critic, $TD(\lambda)$ and Q-learning. Section 5 demonstrates a continuum of algorithms that are sensitive to the amount of computation an agent can perform between actual steps of action in the environment. Generalization—the cornerstone of mainstream machine learning research—has the potential of considerably aiding reinforcement learning, as described in Section 6. Section 7 considers the problems that arise when the agent does not have complete perceptual access to the state of the environment. Section 8 catalogs some of reinforcement learning’s successful applications. Finally, Section 9 concludes with some speculations about important open problems and the future of reinforcement learning.

1.1 Reinforcement-Learning Model

In the standard reinforcement-learning model, an agent is connected to its environment via perception and action, as depicted in Figure 1. On each step of interaction the agent receives as input, i , some indication of the current state, s , of the environment; the agent then chooses an action, a , to generate as output. The action changes the state of the environment, and the value of this state transition is communicated to the agent through a scalar *reinforcement signal*, r . The agent’s behavior, B , should choose actions that tend to increase the long-run sum of values of the reinforcement signal. It can learn to do this over time by systematic trial and error, guided by a wide variety of algorithms that are the subject of later sections of this paper.

Formally, the model consists of

- a discrete set of environment states, \mathcal{S} ;
- a discrete set of agent actions, \mathcal{A} ; and
- a set of scalar reinforcement signals; typically $\{0, 1\}$, or the real numbers.

The figure also includes an input function I , which determines how the agent views the environment state; we will assume that it is the identity function (that is, the agent perceives the exact state of the environment) until we consider partial observability in Section 7.

An intuitive way to understand the relation between the agent and its environment is with the following example dialogue.

Environment:	You are in state 65. You have 4 possible actions.
Agent:	I'll take action 2.
Environment:	You received a reinforcement of 7 units. You are now in state 15. You have 2 possible actions.
Agent:	I'll take action 1.
Environment:	You received a reinforcement of -4 units. You are now in state 65. You have 4 possible actions.
Agent:	I'll take action 2.
Environment:	You received a reinforcement of 5 units. You are now in state 44. You have 5 possible actions.
:	:

The agent's job is to find a policy π , mapping states to actions, that maximizes some long-run measure of reinforcement. We expect, in general, that the environment will be non-deterministic; that is, that taking the same action in the same state on two different occasions may result in different next states and/or different reinforcement values. This happens in our example above: from state 65, applying action 2 produces differing reinforcements and differing states on two occasions. However, we assume the environment is stationary; that is, that the *probabilities* of making state transitions or receiving specific reinforcement signals do not change over time.¹

Reinforcement learning differs from the more widely studied problem of supervised learning in several ways. The most important difference is that there is no presentation of input/output pairs. Instead, after choosing an action the agent is told the immediate reward and the subsequent state, but is *not* told which action would have been in its best long-term interests. It is necessary for the agent to gather useful experience about the possible system states, actions, transitions and rewards actively to act optimally. Another difference from supervised learning is that on-line performance is important: the evaluation of the system is often concurrent with learning.

1. This assumption may be disappointing; after all, operation in non-stationary environments is one of the motivations for building learning systems. In fact, many of the algorithms described in later sections are effective in slowly-varying non-stationary environments, but there is very little theoretical analysis in this area.

Some aspects of reinforcement learning are closely related to search and planning issues in artificial intelligence. AI search algorithms generate a satisfactory trajectory through a graph of states. Planning operates in a similar manner, but typically within a construct with more complexity than a graph, in which states are represented by compositions of logical expressions instead of atomic symbols. These AI algorithms are less general than the reinforcement-learning methods, in that they require a predefined model of state transitions, and with a few exceptions assume determinism. On the other hand, reinforcement learning, at least in the kind of discrete cases for which theory has been developed, assumes that the entire state space can be enumerated and stored in memory—an assumption to which conventional search algorithms are not tied.

1.2 Models of Optimal Behavior

Before we can start thinking about algorithms for learning to behave optimally, we have to decide what our model of optimality will be. In particular, we have to specify how the agent should take the future into account in the decisions it makes about how to behave now. There are three models that have been the subject of the majority of work in this area.

The *finite-horizon* model is the easiest to think about; at a given moment in time, the agent should optimize its expected reward for the next h steps:

$$E\left(\sum_{t=0}^h r_t\right) ;$$

it need not worry about what will happen after that. In this and subsequent expressions, r_t represents the scalar reward received t steps into the future. This model can be used in two ways. In the first, the agent will have a non-stationary policy; that is, one that changes over time. On its first step it will take what is termed a *h -step optimal action*. This is defined to be the best action available given that it has h steps remaining in which to act and gain reinforcement. On the next step it will take a $(h - 1)$ -step optimal action, and so on, until it finally takes a 1-step optimal action and terminates. In the second, the agent does *receding-horizon control*, in which it always takes the h -step optimal action. The agent always acts according to the same policy, but the value of h limits how far ahead it looks in choosing its actions. The finite-horizon model is not always appropriate. In many cases we may not know the precise length of the agent's life in advance.

The infinite-horizon discounted model takes the long-run reward of the agent into account, but rewards that are received in the future are geometrically discounted according to discount factor γ , (where $0 \leq \gamma < 1$):

$$E\left(\sum_{t=0}^{\infty} \gamma^t r_t\right) .$$

We can interpret γ in several ways. It can be seen as an interest rate, a probability of living another step, or as a mathematical trick to bound the infinite sum. The model is conceptually similar to receding-horizon control, but the discounted model is more mathematically tractable than the finite-horizon model. This is a dominant reason for the wide attention this model has received.

Another optimality criterion is the *average-reward model*, in which the agent is supposed to take actions that optimize its long-run average reward:

$$\lim_{h \rightarrow \infty} E\left(\frac{1}{h} \sum_{t=0}^h r_t\right) .$$

Such a policy is referred to as a *gain optimal* policy; it can be seen as the limiting case of the infinite-horizon discounted model as the discount factor approaches 1 (Bertsekas, 1995). One problem with this criterion is that there is no way to distinguish between two policies, one of which gains a large amount of reward in the initial phases and the other of which does not. Reward gained on any initial prefix of the agent's life is overshadowed by the long-run average performance. It is possible to generalize this model so that it takes into account both the long run average and the amount of initial reward than can be gained. In the generalized, *bias optimal* model, a policy is preferred if it maximizes the long-run average and ties are broken by the initial extra reward.

Figure 2 contrasts these models of optimality by providing an environment in which changing the model of optimality changes the optimal policy. In this example, circles represent the states of the environment and arrows are state transitions. There is only a single action choice from every state except the start state, which is in the upper left and marked with an incoming arrow. All rewards are zero except where marked. Under a finite-horizon model with $h = 5$, the three actions yield rewards of +6.0, +0.0, and +0.0, so the first action should be chosen; under an infinite-horizon discounted model with $\gamma = 0.9$, the three choices yield +16.2, +59.0, and +58.5 so the second action should be chosen; and under the average reward model, the third action should be chosen since it leads to an average reward of +11. If we change h to 1000 and γ to 0.2, then the second action is optimal for the finite-horizon model and the first for the infinite-horizon discounted model; however, the average reward model will always prefer the best long-term average. Since the choice of optimality model and parameters matters so much, it is important to choose it carefully in any application.

The finite-horizon model is appropriate when the agent's lifetime is known; one important aspect of this model is that as the length of the remaining lifetime decreases, the agent's policy may change. A system with a hard deadline would be appropriately modeled this way. The relative usefulness of infinite-horizon discounted and bias-optimal models is still under debate. Bias-optimality has the advantage of not requiring a discount parameter; however, algorithms for finding bias-optimal policies are not yet as well-understood as those for finding optimal infinite-horizon discounted policies.

1.3 Measuring Learning Performance

The criteria given in the previous section can be used to assess the policies learned by a given algorithm. We would also like to be able to evaluate the quality of learning itself. There are several incompatible measures in use.

- **Eventual convergence to optimal.** Many algorithms come with a provable guarantee of asymptotic convergence to optimal behavior (Watkins & Dayan, 1992). This is reassuring, but useless in practical terms. An agent that quickly reaches a plateau

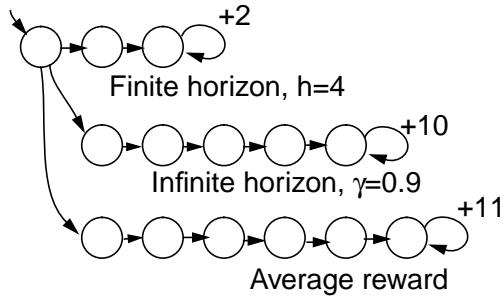


Figure 2: Comparing models of optimality. All unlabeled arrows produce a reward of zero.

at 99% of optimality may, in many applications, be preferable to an agent that has a guarantee of eventual optimality but a sluggish early learning rate.

- **Speed of convergence to optimality.** Optimality is usually an asymptotic result, and so convergence speed is an ill-defined measure. More practical is the *speed of convergence to near-optimality*. This measure begs the definition of how near to optimality is sufficient. A related measure is *level of performance after a given time*, which similarly requires that someone define the given time.

It should be noted that here we have another difference between reinforcement learning and conventional supervised learning. In the latter, expected future predictive accuracy or statistical efficiency are the prime concerns. For example, in the well-known PAC framework (Valiant, 1984), there is a learning period during which mistakes do not count, then a performance period during which they do. The framework provides bounds on the necessary length of the learning period in order to have a probabilistic guarantee on the subsequent performance. That is usually an inappropriate view for an agent with a long existence in a complex environment.

In spite of the mismatch between embedded reinforcement learning and the train/test perspective, Fiechter (1994) provides a PAC analysis for Q-learning (described in Section 4.2) that sheds some light on the connection between the two views.

Measures related to speed of learning have an additional weakness. An algorithm that merely tries to achieve optimality as fast as possible may incur unnecessarily large penalties during the learning period. A less aggressive strategy taking longer to achieve optimality, but gaining greater total reinforcement during its learning might be preferable.

- **Regret.** A more appropriate measure, then, is the expected decrease in reward gained due to executing the learning algorithm instead of behaving optimally from the very beginning. This measure is known as *regret* (Berry & Fristedt, 1985). It penalizes mistakes wherever they occur during the run. Unfortunately, results concerning the regret of algorithms are quite hard to obtain.

1.4 Reinforcement Learning and Adaptive Control

Adaptive control (Burghes & Graham, 1980; Stengel, 1986) is also concerned with algorithms for improving a sequence of decisions from experience. Adaptive control is a much more mature discipline that concerns itself with dynamic systems in which states and actions are vectors and system dynamics are smooth: linear or locally linearizable around a desired trajectory. A very common formulation of cost functions in adaptive control are quadratic penalties on deviation from desired state and action vectors. Most importantly, although the dynamic model of the system is not known in advance, and must be estimated from data, the *structure* of the dynamic model is fixed, leaving model estimation as a parameter estimation problem. These assumptions permit deep, elegant and powerful mathematical analysis, which in turn lead to robust, practical, and widely deployed adaptive control algorithms.

2. Exploitation versus Exploration: The Single-State Case

One major difference between reinforcement learning and supervised learning is that a reinforcement-learner must explicitly explore its environment. In order to highlight the problems of exploration, we treat a very simple case in this section. The fundamental issues and approaches described here will, in many cases, transfer to the more complex instances of reinforcement learning discussed later in the paper.

The simplest possible reinforcement-learning problem is known as the k -armed bandit problem, which has been the subject of a great deal of study in the statistics and applied mathematics literature (Berry & Fristedt, 1985). The agent is in a room with a collection of k gambling machines (each called a “one-armed bandit” in colloquial English). The agent is permitted a fixed number of pulls, h . Any arm may be pulled on each turn. The machines do not require a deposit to play; the only cost is in wasting a pull playing a suboptimal machine. When arm i is pulled, machine i pays off 1 or 0, according to some underlying probability parameter p_i , where payoffs are independent events and the p_i s are unknown. What should the agent’s strategy be?

This problem illustrates the fundamental tradeoff between exploitation and exploration. The agent might believe that a particular arm has a fairly high payoff probability; should it choose that arm all the time, or should it choose another one that it has less information about, but seems to be worse? Answers to these questions depend on how long the agent is expected to play the game; the longer the game lasts, the worse the consequences of prematurely converging on a sub-optimal arm, and the more the agent should explore.

There is a wide variety of solutions to this problem. We will consider a representative selection of them, but for a deeper discussion and a number of important theoretical results, see the book by Berry and Fristedt (1985). We use the term “action” to indicate the agent’s choice of arm to pull. This eases the transition into delayed reinforcement models in Section 3. It is very important to note that bandit problems fit our definition of a reinforcement-learning environment with a single state with only self transitions.

Section 2.1 discusses three solutions to the basic one-state bandit problem that have formal correctness results. Although they can be extended to problems with real-valued rewards, they do not apply directly to the general multi-state delayed-reinforcement case.

Section 2.2 presents three techniques that are not formally justified, but that have had wide use in practice, and can be applied (with similar lack of guarantee) to the general case.

2.1 Formally Justified Techniques

There is a fairly well-developed formal theory of exploration for very simple problems. Although it is instructive, the methods it provides do not scale well to more complex problems.

2.1.1 DYNAMIC-PROGRAMMING APPROACH

If the agent is going to be acting for a total of h steps, it can use basic Bayesian reasoning to solve for an optimal strategy (Berry & Fristedt, 1985). This requires an assumed prior joint distribution for the parameters $\{p_i\}$, the most natural of which is that each p_i is independently uniformly distributed between 0 and 1. We compute a mapping from *belief states* (summaries of the agent's experiences during this run) to actions. Here, a belief state can be represented as a tabulation of action choices and payoffs: $\{n_1, w_1, n_2, w_2, \dots, n_k, w_k\}$ denotes a state of play in which each arm i has been pulled n_i times with w_i payoffs. We write $V^*(n_1, w_1, \dots, n_k, w_k)$ as the expected payoff remaining, given that a total of h pulls are available, and we use the remaining pulls optimally.

If $\sum_i n_i = h$, then there are no remaining pulls, and $V^*(n_1, w_1, \dots, n_k, w_k) = 0$. This is the basis of a recursive definition. If we know the V^* value for all belief states with t pulls remaining, we can compute the V^* value of any belief state with $t + 1$ pulls remaining:

$$\begin{aligned} V^*(n_1, w_1, \dots, n_k, w_k) &= \max_i E \left[\begin{array}{l} \text{Future payoff if agent takes action } i, \\ \text{then acts optimally for remaining pulls} \end{array} \right] \\ &= \max_i \left(\begin{array}{l} \rho_i V^*(n_1, w_i, \dots, n_i + 1, w_i + 1, \dots, n_k, w_k) + \\ (1 - \rho_i) V^*(n_1, w_i, \dots, n_i + 1, w_i, \dots, n_k, w_k) \end{array} \right) \end{aligned}$$

where ρ_i is the posterior subjective probability of action i paying off given n_i , w_i and our prior probability. For the uniform priors, which result in a beta distribution, $\rho_i = (w_i + 1)/(n_i + 2)$.

The expense of filling in the table of V^* values in this way for all attainable belief states is linear in the number of belief states times actions, and thus exponential in the horizon.

2.1.2 GITTINS ALLOCATION INDICES

Gittins gives an “allocation index” method for finding the optimal choice of action at each step in k -armed bandit problems (Gittins, 1989). The technique only applies under the discounted expected reward criterion. For each action, consider the number of times it has been chosen, n , versus the number of times it has paid off, w . For certain discount factors, there are published tables of “index values,” $I(n, w)$ for each pair of n and w . Look up the index value for each action i , $I(n_i, w_i)$. It represents a comparative measure of the combined value of the expected payoff of action i (given its history of payoffs) and the value of the information that we would get by choosing it. Gittins has shown that choosing the action with the largest index value guarantees the optimal balance between exploration and exploitation.

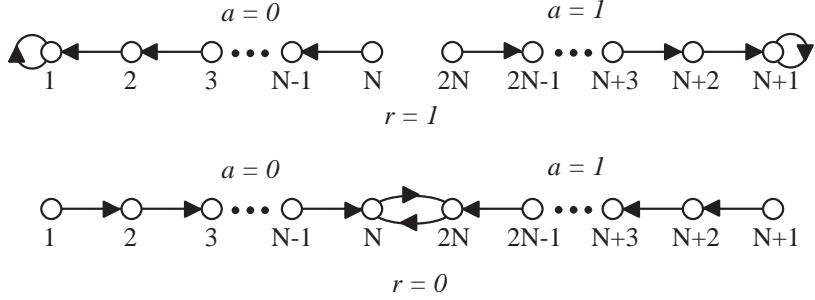


Figure 3: A Tsetlin automaton with $2N$ states. The top row shows the state transitions that are made when the previous action resulted in a reward of 1; the bottom row shows transitions after a reward of 0. In states in the left half of the figure, action 0 is taken; in those on the right, action 1 is taken.

Because of the guarantee of optimal exploration and the simplicity of the technique (given the table of index values), this approach holds a great deal of promise for use in more complex applications. This method proved useful in an application to robotic manipulation with immediate reward (Salganicoff & Ungar, 1995). Unfortunately, no one has yet been able to find an analog of index values for delayed reinforcement problems.

2.1.3 LEARNING AUTOMATA

A branch of the theory of adaptive control is devoted to *learning automata*, surveyed by Narendra and Thathachar (1989), which were originally described explicitly as finite state automata. The *Tsetlin automaton* shown in Figure 3 provides an example that solves a 2-armed bandit arbitrarily near optimally as N approaches infinity.

It is inconvenient to describe algorithms as finite-state automata, so a move was made to describe the internal state of the agent as a probability distribution according to which actions would be chosen. The probabilities of taking different actions would be adjusted according to their previous successes and failures.

An example, which stands among a set of algorithms independently developed in the mathematical psychology literature (Hilgard & Bower, 1975), is the *linear reward-inaction* algorithm. Let p_i be the agent's probability of taking action i .

- When action a_i succeeds,

$$\begin{aligned} p_i &:= p_i + \alpha(1 - p_i) \\ p_j &:= p_j - \alpha p_j \text{ for } j \neq i \end{aligned}$$

- When action a_i fails, p_j remains unchanged (for all j).

This algorithm converges with probability 1 to a vector containing a single 1 and the rest 0's (choosing a particular action with probability 1). Unfortunately, it does not always converge to the correct action; but the probability that it converges to the wrong one can be made arbitrarily small by making α small (Narendra & Thathachar, 1974). There is no literature on the regret of this algorithm.

2.2 Ad-Hoc Techniques

In reinforcement-learning practice, some simple, *ad hoc* strategies have been popular. They are rarely, if ever, the best choice for the models of optimality we have used, but they may be viewed as reasonable, computationally tractable, heuristics. Thrun (1992) has surveyed a variety of these techniques.

2.2.1 GREEDY STRATEGIES

The first strategy that comes to mind is to always choose the action with the highest estimated payoff. The flaw is that early unlucky sampling might indicate that the best action's reward is less than the reward obtained from a suboptimal action. The suboptimal action will always be picked, leaving the true optimal action starved of data and its superiority never discovered. An agent must explore to ameliorate this outcome.

A useful heuristic is *optimism in the face of uncertainty* in which actions are selected greedily, but strongly optimistic prior beliefs are put on their payoffs so that strong negative evidence is needed to eliminate an action from consideration. This still has a measurable danger of starving an optimal but unlucky action, but the risk of this can be made arbitrarily small. Techniques like this have been used in several reinforcement learning algorithms including the interval exploration method (Kaelbling, 1993b) (described shortly), the *exploration bonus* in Dyna (Sutton, 1990), *curiosity-driven exploration* (Schmidhuber, 1991a), and the exploration mechanism in prioritized sweeping (Moore & Atkeson, 1993).

2.2.2 RANDOMIZED STRATEGIES

Another simple exploration strategy is to take the action with the best estimated expected reward by default, but with probability p , choose an action at random. Some versions of this strategy start with a large value of p to encourage initial exploration, which is slowly decreased.

An objection to the simple strategy is that when it experiments with a non-greedy action it is no more likely to try a promising alternative than a clearly hopeless alternative. A slightly more sophisticated strategy is *Boltzmann exploration*. In this case, the expected reward for taking action a , $ER(a)$ is used to choose an action probabilistically according to the distribution

$$P(a) = \frac{e^{ER(a)/T}}{\sum_{a' \in A} e^{ER(a')/T}} .$$

The *temperature* parameter T can be decreased over time to decrease exploration. This method works well if the best action is well separated from the others, but suffers somewhat when the values of the actions are close. It may also converge unnecessarily slowly unless the temperature schedule is manually tuned with great care.

2.2.3 INTERVAL-BASED TECHNIQUES

Exploration is often more efficient when it is based on second-order information about the certainty or variance of the estimated values of actions. Kaelbling's *interval estimation* algorithm (1993b) stores statistics for each action a_i : w_i is the number of successes and n_i the number of trials. An action is chosen by computing the upper bound of a $100 \cdot (1 - \alpha)\%$

confidence interval on the success probability of each action and choosing the action with the highest upper bound. Smaller values of the α parameter encourage greater exploration. When payoffs are boolean, the normal approximation to the binomial distribution can be used to construct the confidence interval (though the binomial should be used for small n). Other payoff distributions can be handled using their associated statistics or with nonparametric methods. The method works very well in empirical trials. It is also related to a certain class of statistical techniques known as *experiment design* methods (Box & Draper, 1987), which are used for comparing multiple treatments (for example, fertilizers or drugs) to determine which treatment (if any) is best in as small a set of experiments as possible.

2.3 More General Problems

When there are multiple states, but reinforcement is still immediate, then any of the above solutions can be replicated, once for each state. However, when generalization is required, these solutions must be integrated with generalization methods (see section 6); this is straightforward for the simple ad-hoc methods, but it is not understood how to maintain theoretical guarantees.

Many of these techniques focus on converging to some regime in which exploratory actions are taken rarely or never; this is appropriate when the environment is stationary. However, when the environment is non-stationary, exploration must continue to take place, in order to notice changes in the world. Again, the more ad-hoc techniques can be modified to deal with this in a plausible manner (keep temperature parameters from going to 0; decay the statistics in interval estimation), but none of the theoretically guaranteed methods can be applied.

3. Delayed Reward

In the general case of the reinforcement learning problem, the agent's actions determine not only its immediate reward, but also (at least probabilistically) the next state of the environment. Such environments can be thought of as networks of bandit problems, but the agent must take into account the next state as well as the immediate reward when it decides which action to take. The model of long-run optimality the agent is using determines exactly how it should take the value of the future into account. The agent will have to be able to learn from delayed reinforcement: it may take a long sequence of actions, receiving insignificant reinforcement, then finally arrive at a state with high reinforcement. The agent must be able to learn which of its actions are desirable based on reward that can take place arbitrarily far in the future.

3.1 Markov Decision Processes

Problems with delayed reinforcement are well modeled as *Markov decision processes* (MDPs). An MDP consists of

- a set of states \mathcal{S} ,
- a set of actions \mathcal{A} ,

- a reward function $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, and
- a state transition function $T : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\mathcal{S})$, where a member of $\Pi(\mathcal{S})$ is a probability distribution over the set \mathcal{S} (i.e. it maps states to probabilities). We write $T(s, a, s')$ for the probability of making a transition from state s to state s' using action a .

The state transition function probabilistically specifies the next state of the environment as a function of its current state and the agent's action. The reward function specifies expected instantaneous reward as a function of the current state and action. The model is *Markov* if the state transitions are independent of any previous environment states or agent actions. There are many good references to MDP models (Bellman, 1957; Bertsekas, 1987; Howard, 1960; Puterman, 1994).

Although general MDPs may have infinite (even uncountable) state and action spaces, we will only discuss methods for solving finite-state and finite-action problems. In section 6, we discuss methods for solving problems with continuous input and output spaces.

3.2 Finding a Policy Given a Model

Before we consider algorithms for learning to behave in MDP environments, we will explore techniques for determining the optimal policy given a correct model. These dynamic programming techniques will serve as the foundation and inspiration for the learning algorithms to follow. We restrict our attention mainly to finding optimal policies for the infinite-horizon discounted model, but most of these algorithms have analogs for the finite-horizon and average-case models as well. We rely on the result that, for the infinite-horizon discounted model, there exists an optimal deterministic stationary policy (Bellman, 1957).

We will speak of the optimal *value* of a state—it is the expected infinite discounted sum of reward that the agent will gain if it starts in that state and executes the optimal policy. Using π as a complete decision policy, it is written

$$V^*(s) = \max_{\pi} E \left(\sum_{t=0}^{\infty} \gamma^t r_t \right) .$$

This optimal value function is unique and can be defined as the solution to the simultaneous equations

$$V^*(s) = \max_a \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') V^*(s') \right), \forall s \in \mathcal{S} , \quad (1)$$

which assert that the value of a state s is the expected instantaneous reward plus the expected discounted value of the next state, using the best available action. Given the optimal value function, we can specify the optimal policy as

$$\pi^*(s) = \arg \max_a \left(R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') V^*(s') \right) .$$

3.2.1 VALUE ITERATION

One way, then, to find an optimal policy is to find the optimal value function. It can be determined by a simple iterative algorithm called *value iteration* that can be shown to converge to the correct V^* values (Bellman, 1957; Bertsekas, 1987).

```

initialize  $V(s)$  arbitrarily
loop until policy good enough
  loop for  $s \in \mathcal{S}$ 
    loop for  $a \in \mathcal{A}$ 
       $Q(s, a) := R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s')V(s')$ 
       $V(s) := \max_a Q(s, a)$ 
    end loop
  end loop

```

It is not obvious when to stop the value iteration algorithm. One important result bounds the performance of the current greedy policy as a function of the *Bellman residual* of the current value function (Williams & Baird, 1993b). It says that if the maximum difference between two successive value functions is less than ϵ , then the value of the greedy policy, (the policy obtained by choosing, in every state, the action that maximizes the estimated discounted reward, using the current estimate of the value function) differs from the value function of the optimal policy by no more than $2\epsilon\gamma/(1 - \gamma)$ at any state. This provides an effective stopping criterion for the algorithm. Puterman (1994) discusses another stopping criterion, based on the *span semi-norm*, which may result in earlier termination. Another important result is that the greedy policy is guaranteed to be optimal in some finite number of steps even though the value function may not have converged (Bertsekas, 1987). And in practice, the greedy policy is often optimal long before the value function has converged.

Value iteration is very flexible. The assignments to V need not be done in strict order as shown above, but instead can occur asynchronously in parallel provided that the value of every state gets updated infinitely often on an infinite run. These issues are treated extensively by Bertsekas (1989), who also proves convergence results.

Updates based on Equation 1 are known as *full backups* since they make use of information from all possible successor states. It can be shown that updates of the form

$$Q(s, a) := Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

can also be used as long as each pairing of a and s is updated infinitely often, s' is sampled from the distribution $T(s, a, s')$, r is sampled with mean $R(s, a)$ and bounded variance, and the learning rate α is decreased slowly. This type of *sample backup* (Singh, 1993) is critical to the operation of the model-free methods discussed in the next section.

The computational complexity of the value-iteration algorithm with full backups, per iteration, is quadratic in the number of states and linear in the number of actions. Commonly, the transition probabilities $T(s, a, s')$ are sparse. If there are on average a constant number of next states with non-zero probability then the cost per iteration is linear in the number of states and linear in the number of actions. The number of iterations required to reach the optimal value function is polynomial in the number of states and the magnitude of the largest reward if the discount factor is held constant. However, in the worst case the number of iterations grows polynomially in $1/(1 - \gamma)$, so the convergence rate slows considerably as the discount factor approaches 1 (Littman, Dean, & Kaelbling, 1995b).

3.2.2 POLICY ITERATION

The *policy iteration* algorithm manipulates the policy directly, rather than finding it indirectly via the optimal value function. It operates as follows:

```

choose an arbitrary policy  $\pi'$ 
loop
   $\pi := \pi'$ 
  compute the value function of policy  $\pi$ :
    solve the linear equations
     $V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} T(s, \pi(s), s') V_\pi(s')$ 
  improve the policy at each state:
     $\pi'(s) := \arg \max_a (R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') V_\pi(s'))$ 
until  $\pi = \pi'$ 
```

The value function of a policy is just the expected infinite discounted reward that will be gained, at each state, by executing that policy. It can be determined by solving a set of linear equations. Once we know the value of each state under the current policy, we consider whether the value could be improved by changing the first action taken. If it can, we change the policy to take the new action whenever it is in that situation. This step is guaranteed to strictly improve the performance of the policy. When no improvements are possible, then the policy is guaranteed to be optimal.

Since there are at most $|\mathcal{A}|^{|\mathcal{S}|}$ distinct policies, and the sequence of policies improves at each step, this algorithm terminates in at most an exponential number of iterations (Puterman, 1994). However, it is an important open question how many iterations policy iteration takes in the worst case. It is known that the running time is pseudopolynomial and that for any fixed discount factor, there is a polynomial bound in the total size of the MDP (Littman et al., 1995b).

3.2.3 ENHANCEMENT TO VALUE ITERATION AND POLICY ITERATION

In practice, value iteration is much faster per iteration, but policy iteration takes fewer iterations. Arguments have been put forth to the effect that each approach is better for large problems. Puterman's *modified policy iteration* algorithm (Puterman & Shin, 1978) provides a method for trading iteration time for iteration improvement in a smoother way. The basic idea is that the expensive part of policy iteration is solving for the exact value of V_π . Instead of finding an exact value for V_π , we can perform a few steps of a modified value-iteration step where the policy is held fixed over successive iterations. This can be shown to produce an approximation to V_π that converges linearly in γ . In practice, this can result in substantial speedups.

Several standard numerical-analysis techniques that speed the convergence of dynamic programming can be used to accelerate value and policy iteration. *Multigrid methods* can be used to quickly seed a good initial approximation to a high resolution value function by initially performing value iteration at a coarser resolution (Rüde, 1993). *State aggregation* works by collapsing groups of states to a single meta-state solving the abstracted problem (Bertsekas & Castañon, 1989).

3.2.4 COMPUTATIONAL COMPLEXITY

Value iteration works by producing successive approximations of the optimal value function. Each iteration can be performed in $O(|A||S|^2)$ steps, or faster if there is sparsity in the transition function. However, the number of iterations required can grow exponentially in the discount factor (Condon, 1992); as the discount factor approaches 1, the decisions must be based on results that happen farther and farther into the future. In practice, policy iteration converges in fewer iterations than value iteration, although the per-iteration costs of $O(|A||S|^2 + |S|^3)$ can be prohibitive. There is no known tight worst-case bound available for policy iteration (Littman et al., 1995b). Modified policy iteration (Puterman & Shin, 1978) seeks a trade-off between cheap and effective iterations and is preferred by some practitioners (Rust, 1996).

Linear programming (Schrijver, 1986) is an extremely general problem, and MDPs can be solved by general-purpose linear-programming packages (Derman, 1970; D'Epenoux, 1963; Hoffman & Karp, 1966). An advantage of this approach is that commercial-quality linear-programming packages are available, although the time and space requirements can still be quite high. From a theoretic perspective, linear programming is the only known algorithm that can solve MDPs in polynomial time, although the theoretically efficient algorithms have not been shown to be efficient in practice.

4. Learning an Optimal Policy: Model-free Methods

In the previous section we reviewed methods for obtaining an optimal policy for an MDP assuming that we already had a model. The model consists of knowledge of the state transition probability function $T(s, a, s')$ and the reinforcement function $R(s, a)$. Reinforcement learning is primarily concerned with how to obtain the optimal policy when such a model is not known in advance. The agent must interact with its environment directly to obtain information which, by means of an appropriate algorithm, can be processed to produce an optimal policy.

At this point, there are two ways to proceed.

- **Model-free:** Learn a controller without learning a model.
- **Model-based:** Learn a model, and use it to derive a controller.

Which approach is better? This is a matter of some debate in the reinforcement-learning community. A number of algorithms have been proposed on both sides. This question also appears in other fields, such as adaptive control, where the dichotomy is between *direct* and *indirect* adaptive control.

This section examines model-free learning, and Section 5 examines model-based methods.

The biggest problem facing a reinforcement-learning agent is *temporal credit assignment*. How do we know whether the action just taken is a good one, when it might have far-reaching effects? One strategy is to wait until the “end” and reward the actions taken if the result was good and punish them if the result was bad. In ongoing tasks, it is difficult to know what the “end” is, and this might require a great deal of memory. Instead, we will use insights from value iteration to adjust the estimated value of a state based on

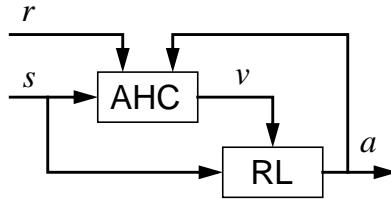


Figure 4: Architecture for the adaptive heuristic critic.

the immediate reward and the estimated value of the next state. This class of algorithms is known as *temporal difference methods* (Sutton, 1988). We will consider two different temporal-difference learning strategies for the discounted infinite-horizon model.

4.1 Adaptive Heuristic Critic and $TD(\lambda)$

The *adaptive heuristic critic* algorithm is an adaptive version of policy iteration (Barto, Sutton, & Anderson, 1983) in which the value-function computation is no longer implemented by solving a set of linear equations, but is instead computed by an algorithm called $TD(0)$. A block diagram for this approach is given in Figure 4. It consists of two components: a critic (labeled AHC), and a reinforcement-learning component (labeled RL). The reinforcement-learning component can be an instance of any of the k -armed bandit algorithms, modified to deal with multiple states and non-stationary rewards. But instead of acting to maximize instantaneous reward, it will be acting to maximize the heuristic value, v , that is computed by the critic. The critic uses the real external reinforcement signal to learn to map states to their expected discounted values given that the policy being executed is the one currently instantiated in the RL component.

We can see the analogy with modified policy iteration if we imagine these components working in alternation. The policy π implemented by RL is fixed and the critic learns the value function V_π for that policy. Now we fix the critic and let the RL component learn a new policy π' that maximizes the new value function, and so on. In most implementations, however, both components operate simultaneously. Only the alternating implementation can be guaranteed to converge to the optimal policy, under appropriate conditions. Williams and Baird explored the convergence properties of a class of AHC-related algorithms they call “incremental variants of policy iteration” (Williams & Baird, 1993a).

It remains to explain how the critic can learn the value of a policy. We define $\langle s, a, r, s' \rangle$ to be an *experience tuple* summarizing a single transition in the environment. Here s is the agent’s state before the transition, a is its choice of action, r the instantaneous reward it receives, and s' its resulting state. The value of a policy is learned using Sutton’s $TD(0)$ algorithm (Sutton, 1988) which uses the update rule

$$V(s) := V(s) + \alpha(r + \gamma V(s') - V(s)) .$$

Whenever a state s is visited, its estimated value is updated to be closer to $r + \gamma V(s')$, since r is the instantaneous reward received and $V(s')$ is the estimated value of the actually occurring next state. This is analogous to the sample-backup rule from value iteration—the only difference is that the sample is drawn from the real world rather than by simulating a known model. The key idea is that $r + \gamma V(s')$ is a sample of the value of $V(s)$, and it is

more likely to be correct because it incorporates the real r . If the learning rate α is adjusted properly (it must be slowly decreased) and the policy is held fixed, $TD(0)$ is guaranteed to converge to the optimal value function.

The $TD(0)$ rule as presented above is really an instance of a more general class of algorithms called $TD(\lambda)$, with $\lambda = 0$. $TD(0)$ looks only one step ahead when adjusting value estimates; although it will eventually arrive at the correct answer, it can take quite a while to do so. The general $TD(\lambda)$ rule is similar to the $TD(0)$ rule given above,

$$V(u) := V(u) + \alpha(r + \gamma V(s') - V(s))e(u) ,$$

but it is applied to *every state* according to its eligibility $e(u)$, rather than just to the immediately previous state, s . One version of the eligibility trace is defined to be

$$e(s) = \sum_{k=1}^t (\lambda\gamma)^{t-k} \delta_{s,s_k} , \text{ where } \delta_{s,s_k} = \begin{cases} 1 & \text{if } s = s_k \\ 0 & \text{otherwise} \end{cases} .$$

The eligibility of a state s is the degree to which it has been visited in the recent past; when a reinforcement is received, it is used to update all the states that have been recently visited, according to their eligibility. When $\lambda = 0$ this is equivalent to $TD(0)$. When $\lambda = 1$, it is roughly equivalent to updating all the states according to the number of times they were visited by the end of a run. Note that we can update the eligibility online as follows:

$$e(s) := \begin{cases} \gamma\lambda e(s) + 1 & \text{if } s = \text{current state} \\ \gamma\lambda e(s) & \text{otherwise} \end{cases} .$$

It is computationally more expensive to execute the general $TD(\lambda)$, though it often converges considerably faster for large λ (Dayan, 1992; Dayan & Sejnowski, 1994). There has been some recent work on making the updates more efficient (Cichosz & Mulawka, 1995) and on changing the definition to make $TD(\lambda)$ more consistent with the certainty-equivalent method (Singh & Sutton, 1996), which is discussed in Section 5.1.

4.2 Q-learning

The work of the two components of AHC can be accomplished in a unified manner by Watkins' Q-learning algorithm (Watkins, 1989; Watkins & Dayan, 1992). Q-learning is typically easier to implement. In order to understand Q-learning, we have to develop some additional notation. Let $Q^*(s, a)$ be the expected discounted reinforcement of taking action a in state s , then continuing by choosing actions optimally. Note that $V^*(s)$ is the value of s assuming the best action is taken initially, and so $V^*(s) = \max_a Q^*(s, a)$. $Q^*(s, a)$ can hence be written recursively as

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} T(s, a, s') \max_{a'} Q^*(s', a') .$$

Note also that, since $V^*(s) = \max_a Q^*(s, a)$, we have $\pi^*(s) = \arg \max_a Q^*(s, a)$ as an optimal policy.

Because the Q function makes the action explicit, we can estimate the Q values online using a method essentially the same as $TD(0)$, but also use them to define the policy,

because an action can be chosen just by taking the one with the maximum Q value for the current state.

The Q-learning rule is

$$Q(s, a) := Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)) ,$$

where $\langle s, a, r, s' \rangle$ is an experience tuple as described earlier. If each action is executed in each state an infinite number of times on an infinite run and α is decayed appropriately, the Q values will converge with probability 1 to Q^* (Watkins, 1989; Tsitsiklis, 1994; Jaakkola, Jordan, & Singh, 1994). Q-learning can also be extended to update states that occurred more than one step previously, as in $TD(\lambda)$ (Peng & Williams, 1994).

When the Q values are nearly converged to their optimal values, it is appropriate for the agent to act greedily, taking, in each situation, the action with the highest Q value. During learning, however, there is a difficult exploitation versus exploration trade-off to be made. There are no good, formally justified approaches to this problem in the general case; standard practice is to adopt one of the *ad hoc* methods discussed in section 2.2.

AHC architectures seem to be more difficult to work with than Q-learning on a practical level. It can be hard to get the relative learning rates right in AHC so that the two components converge together. In addition, Q-learning is *exploration insensitive*: that is, that the Q values will converge to the optimal values, independent of how the agent behaves while the data is being collected (as long as all state-action pairs are tried often enough). This means that, although the exploration-exploitation issue must be addressed in Q-learning, the details of the exploration strategy will not affect the convergence of the learning algorithm. For these reasons, Q-learning is the most popular and seems to be the most effective model-free algorithm for learning from delayed reinforcement. It does not, however, address any of the issues involved in generalizing over large state and/or action spaces. In addition, it may converge quite slowly to a good policy.

4.3 Model-free Learning With Average Reward

As described, Q-learning can be applied to discounted infinite-horizon MDPs. It can also be applied to undiscounted problems as long as the optimal policy is guaranteed to reach a reward-free absorbing state and the state is periodically reset.

Schwartz (1993) examined the problem of adapting Q-learning to an average-reward framework. Although his R-learning algorithm seems to exhibit convergence problems for some MDPs, several researchers have found the average-reward criterion closer to the true problem they wish to solve than a discounted criterion and therefore prefer R-learning to Q-learning (Mahadevan, 1994).

With that in mind, researchers have studied the problem of learning optimal average-reward policies. Mahadevan (1996) surveyed model-based average-reward algorithms from a reinforcement-learning perspective and found several difficulties with existing algorithms. In particular, he showed that existing reinforcement-learning algorithms for average reward (and some dynamic programming algorithms) do not always produce bias-optimal policies. Jaakkola, Jordan and Singh (1995) described an average-reward learning algorithm with guaranteed convergence properties. It uses a Monte-Carlo component to estimate the expected future reward for each state as the agent moves through the environment. In

addition, Bertsekas presents a Q-learning-like algorithm for average-case reward in his new textbook (1995). Although this recent work provides a much needed theoretical foundation to this area of reinforcement learning, many important problems remain unsolved.

5. Computing Optimal Policies by Learning Models

The previous section showed how it is possible to learn an optimal policy without knowing the models $T(s, a, s')$ or $R(s, a)$ and without even learning those models en route. Although many of these methods are guaranteed to find optimal policies eventually and use very little computation time per experience, they make extremely inefficient use of the data they gather and therefore often require a great deal of experience to achieve good performance. In this section we still begin by assuming that we don't know the models in advance, but we examine algorithms that do operate by learning these models. These algorithms are especially important in applications in which computation is considered to be cheap and real-world experience costly.

5.1 Certainty Equivalent Methods

We begin with the most conceptually straightforward method: first, learn the T and R functions by exploring the environment and keeping statistics about the results of each action; next, compute an optimal policy using one of the methods of Section 3. This method is known as *certainty equivlance* (Kumar & Varaiya, 1986).

There are some serious objections to this method:

- It makes an arbitrary division between the learning phase and the acting phase.
- How should it gather data about the environment initially? Random exploration might be dangerous, and in some environments is an immensely inefficient method of gathering data, requiring exponentially more data (Whitehead, 1991) than a system that interleaves experience gathering with policy-building more tightly (Koenig & Simmons, 1993). See Figure 5 for an example.
- The possibility of changes in the environment is also problematic. Breaking up an agent's life into a pure learning and a pure acting phase has a considerable risk that the optimal controller based on early life becomes, without detection, a suboptimal controller if the environment changes.

A variation on this idea is *certainty equivalence*, in which the model is learned continually through the agent's lifetime and, at each step, the current model is used to compute an optimal policy and value function. This method makes very effective use of available data, but still ignores the question of exploration and is extremely computationally demanding, even for fairly small state spaces. Fortunately, there are a number of other model-based algorithms that are more practical.

5.2 Dyna

Sutton's Dyna architecture (1990, 1991) exploits a middle ground, yielding strategies that are both more effective than model-free learning and more computationally efficient than

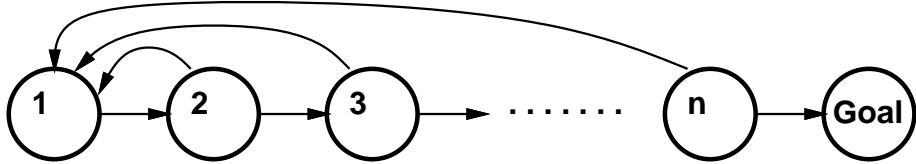


Figure 5: In this environment, due to Whitehead (1991), random exploration would take $O(2^n)$ steps to reach the goal even once, whereas a more intelligent exploration strategy (e.g. “assume any untried action leads directly to goal”) would require only $O(n^2)$ steps.

the certainty-equivalence approach. It simultaneously uses experience to build a model (\hat{T} and \hat{R}), uses experience to adjust the policy, and uses the model to adjust the policy.

Dyna operates in a loop of interaction with the environment. Given an experience tuple $\langle s, a, s', r \rangle$, it behaves as follows:

- Update the model, incrementing statistics for the transition from s to s' on action a and for receiving reward r for taking action a in state s . The updated models are \hat{T} and \hat{R} .
- Update the policy at state s based on the newly updated model using the rule

$$Q(s, a) := \hat{R}(s, a) + \gamma \sum_{s'} \hat{T}(s, a, s') \max_{a'} Q(s', a') ,$$

which is a version of the value-iteration update for Q values.

- Perform k additional updates: choose k state-action pairs at random and update them according to the same rule as before:

$$Q(s_k, a_k) := \hat{R}(s_k, a_k) + \gamma \sum_{s'} \hat{T}(s_k, a_k, s') \max_{a'} Q(s', a') .$$

- Choose an action a' to perform in state s' , based on the Q values but perhaps modified by an exploration strategy.

The Dyna algorithm requires about k times the computation of Q-learning per instance, but this is typically vastly less than for the naive model-based method. A reasonable value of k can be determined based on the relative speeds of computation and of taking action.

Figure 6 shows a grid world in which in each cell the agent has four actions (N, S, E, W) and transitions are made deterministically to an adjacent cell, unless there is a block, in which case no movement occurs. As we will see in Table 1, Dyna requires an order of magnitude fewer steps of experience than does Q-learning to arrive at an optimal policy. Dyna requires about six times more computational effort, however.

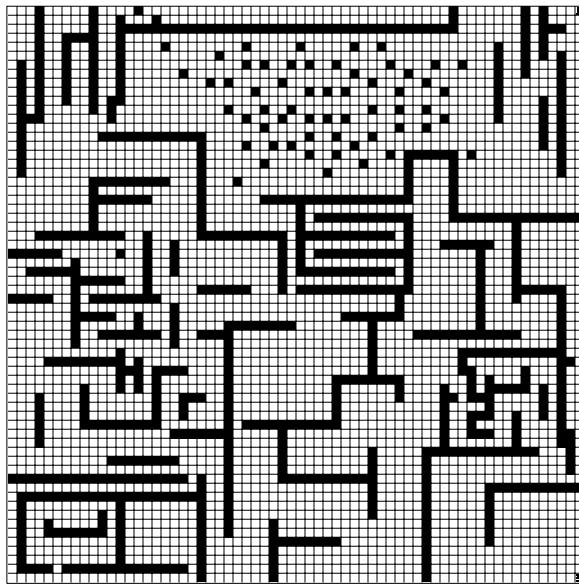


Figure 6: A 3277-state grid world. This was formulated as a shortest-path reinforcement-learning problem, which yields the same result as if a reward of 1 is given at the goal, a reward of zero elsewhere and a discount factor is used.

	Steps before convergence	Backups before convergence
Q-learning	531,000	531,000
Dyna	62,000	3,055,000
prioritized sweeping	28,000	1,010,000

Table 1: The performance of three algorithms described in the text. All methods used the exploration heuristic of “optimism in the face of uncertainty”: any state not previously visited was assumed by default to be a goal state. Q-learning used its optimal learning rate parameter for a deterministic maze: $\alpha = 1$. Dyna and prioritized sweeping were permitted to take $k = 200$ backups per transition. For prioritized sweeping, the priority queue often emptied before all backups were used.

5.3 Prioritized Sweeping / Queue-Dyna

Although Dyna is a great improvement on previous methods, it suffers from being relatively undirected. It is particularly unhelpful when the goal has just been reached or when the agent is stuck in a dead end; it continues to update random state-action pairs, rather than concentrating on the “interesting” parts of the state space. These problems are addressed by prioritized sweeping (Moore & Atkeson, 1993) and Queue-Dyna (Peng & Williams, 1993), which are two independently-developed but very similar techniques. We will describe prioritized sweeping in some detail.

The algorithm is similar to Dyna, except that updates are no longer chosen at random and values are now associated with states (as in value iteration) instead of state-action pairs (as in Q-learning). To make appropriate choices, we must store additional information in the model. Each state remembers its *predecessors*: the states that have a non-zero transition probability to it under some action. In addition, each state has a *priority*, initially set to zero.

Instead of updating k random state-action pairs, prioritized sweeping updates k states with the highest priority. For each high-priority state s , it works as follows:

- Remember the current value of the state: $V_{old} = V(s)$.
- Update the state’s value

$$V(s) := \max_a \left(\hat{R}(s, a) + \gamma \sum_{s'} \hat{T}(s, a, s') V(s') \right) .$$

- Set the state’s priority back to 0.
- Compute the value change $\Delta = |V_{old} - V(s)|$.
- Use Δ to modify the priorities of the predecessors of s .

If we have updated the V value for state s' and it has changed by amount Δ , then the immediate predecessors of s' are informed of this event. Any state s for which there exists an action a such that $\hat{T}(s, a, s') \neq 0$ has its priority promoted to $\Delta \cdot \hat{T}(s, a, s')$, unless its priority already exceeded that value.

The global behavior of this algorithm is that when a real-world transition is “surprising” (the agent happens upon a goal state, for instance), then lots of computation is directed to propagate this new information back to relevant predecessor states. When the real-world transition is “boring” (the actual result is very similar to the predicted result), then computation continues in the most deserving part of the space.

Running prioritized sweeping on the problem in Figure 6, we see a large improvement over Dyna. The optimal policy is reached in about half the number of steps of experience and one-third the computation as Dyna required (and therefore about 20 times fewer steps and twice the computational effort of Q-learning).

5.4 Other Model-Based Methods

Methods proposed for solving MDPs given a model can be used in the context of model-based methods as well.

RTDP (real-time dynamic programming) (Barto, Bradtke, & Singh, 1995) is another model-based method that uses Q-learning to concentrate computational effort on the areas of the state-space that the agent is most likely to occupy. It is specific to problems in which the agent is trying to achieve a particular goal state and the reward everywhere else is 0. By taking into account the start state, it can find a short path from the start to the goal, without necessarily visiting the rest of the state space.

The Plexus planning system (Dean, Kaelbling, Kirman, & Nicholson, 1993; Kirman, 1994) exploits a similar intuition. It starts by making an approximate version of the MDP which is much smaller than the original one. The approximate MDP contains a set of states, called the *envelope*, that includes the agent's current state and the goal state, if there is one. States that are not in the envelope are summarized by a single "out" state. The planning process is an alternation between finding an optimal policy on the approximate MDP and adding useful states to the envelope. Action may take place in parallel with planning, in which case irrelevant states are also pruned out of the envelope.

6. Generalization

All of the previous discussion has tacitly assumed that it is possible to enumerate the state and action spaces and store tables of values over them. Except in very small environments, this means impractical memory requirements. It also makes inefficient use of experience. In a large, smooth state space we generally expect similar states to have similar values and similar optimal actions. Surely, therefore, there should be some more compact representation than a table. Most problems will have continuous or large discrete state spaces; some will have large or continuous action spaces. The problem of learning in large spaces is addressed through *generalization techniques*, which allow compact storage of learned information and transfer of knowledge between "similar" states and actions.

The large literature of generalization techniques from inductive concept learning can be applied to reinforcement learning. However, techniques often need to be tailored to specific details of the problem. In the following sections, we explore the application of standard function-approximation techniques, adaptive resolution models, and hierarchical methods to the problem of reinforcement learning.

The reinforcement-learning architectures and algorithms discussed above have included the storage of a variety of mappings, including $\mathcal{S} \rightarrow \mathcal{A}$ (policies), $\mathcal{S} \rightarrow \mathbb{R}$ (value functions), $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ (Q functions and rewards), $\mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ (deterministic transitions), and $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ (transition probabilities). Some of these mappings, such as transitions and immediate rewards, can be learned using straightforward supervised learning, and can be handled using any of the wide variety of function-approximation techniques for supervised learning that support noisy training examples. Popular techniques include various neural-network methods (Rumelhart & McClelland, 1986), fuzzy logic (Berenji, 1991; Lee, 1991), CMAC (Albus, 1981), and local memory-based methods (Moore, Atkeson, & Schaal, 1995), such as generalizations of nearest neighbor methods. Other mappings, especially the policy

mapping, typically need specialized algorithms because training sets of input-output pairs are not available.

6.1 Generalization over Input

A reinforcement-learning agent's current state plays a central role in its selection of reward-maximizing actions. Viewing the agent as a state-free black box, a description of the current state is its input. Depending on the agent architecture, its output is either an action selection, or an evaluation of the current state that can be used to select an action. The problem of deciding how the different aspects of an input affect the value of the output is sometimes called the "structural credit-assignment" problem. This section examines approaches to generating actions or evaluations as a function of a description of the agent's current state.

The first group of techniques covered here is specialized to the case when reward is not delayed; the second group is more generally applicable.

6.1.1 IMMEDIATE REWARD

When the agent's actions do not influence state transitions, the resulting problem becomes one of choosing actions to maximize immediate reward as a function of the agent's current state. These problems bear a resemblance to the bandit problems discussed in Section 2 except that the agent should condition its action selection on the current state. For this reason, this class of problems has been described as *associative* reinforcement learning.

The algorithms in this section address the problem of learning from immediate boolean reinforcement where the state is vector valued and the action is a boolean vector. Such algorithms can and have been used in the context of a delayed reinforcement, for instance, as the RL component in the AHC architecture described in Section 4.1. They can also be generalized to real-valued reward through *reward comparison* methods (Sutton, 1984).

CRBP The complementary reinforcement backpropagation algorithm (Ackley & Littman, 1990) (CRBP) consists of a feed-forward network mapping an encoding of the state to an encoding of the action. The action is determined probabilistically from the activation of the output units: if output unit i has activation y_i , then bit i of the action vector has value 1 with probability y_i , and 0 otherwise. Any neural-network supervised training procedure can be used to adapt the network as follows. If the result of generating action a is $r = 1$, then the network is trained with input-output pair $\langle s, a \rangle$. If the result is $r = 0$, then the network is trained with input-output pair $\langle s, \bar{a} \rangle$, where $\bar{a} = (1 - a_1, \dots, 1 - a_n)$.

The idea behind this training rule is that whenever an action fails to generate reward, CRBP will try to generate an action that is different from the current choice. Although it seems like the algorithm might oscillate between an action and its complement, that does not happen. One step of training a network will only change the action slightly and since the output probabilities will tend to move toward 0.5, this makes action selection more random and increases search. The hope is that the random distribution will generate an action that works better, and then that action will be reinforced.

ARC The associative reinforcement comparison (ARC) algorithm (Sutton, 1984) is an instance of the AHC architecture for the case of boolean actions, consisting of two feed-

forward networks. One learns the value of situations, the other learns a policy. These can be simple linear networks or can have hidden units.

In the simplest case, the entire system learns only to optimize immediate reward. First, let us consider the behavior of the network that learns the policy, a mapping from a vector describing s to a 0 or 1. If the output unit has activation y_i , then a , the action generated, will be 1 if $y + \nu > 0$, where ν is normal noise, and 0 otherwise.

The adjustment for the output unit is, in the simplest case,

$$e = r(a - 1/2) ,$$

where the first factor is the reward received for taking the most recent action and the second encodes which action was taken. The actions are encoded as 0 and 1, so $a - 1/2$ always has the same magnitude; if the reward and the action have the same sign, then action 1 will be made more likely, otherwise action 0 will be.

As described, the network will tend to seek actions that given positive reward. To extend this approach to maximize reward, we can compare the reward to some baseline, b . This changes the adjustment to

$$e = (r - b)(a - 1/2) ,$$

where b is the output of the second network. The second network is trained in a standard supervised mode to estimate r as a function of the input state s .

Variations of this approach have been used in a variety of applications (Anderson, 1986; Barto et al., 1983; Lin, 1993b; Sutton, 1984).

REINFORCE Algorithms Williams (1987, 1992) studied the problem of choosing actions to maximize immediate reward. He identified a broad class of update rules that perform gradient descent on the expected reward and showed how to integrate these rules with backpropagation. This class, called REINFORCE algorithms, includes linear reward-inaction (Section 2.1.3) as a special case.

The generic REINFORCE update for a parameter w_{ij} can be written

$$\Delta w_{ij} = \alpha_{ij}(r - b_{ij}) \frac{\partial}{\partial w_{ij}} \ln(g_j)$$

where α_{ij} is a non-negative factor, r the current reinforcement, b_{ij} a reinforcement baseline, and g_j is the probability density function used to randomly generate actions based on unit activations. Both α_{ij} and b_{ij} can take on different values for each w_{ij} , however, when α_{ij} is constant throughout the system, the expected update is exactly in the direction of the expected reward gradient. Otherwise, the update is in the same half space as the gradient but not necessarily in the direction of steepest increase.

Williams points out that the choice of baseline, b_{ij} , can have a profound effect on the convergence speed of the algorithm.

Logic-Based Methods Another strategy for generalization in reinforcement learning is to reduce the learning problem to an associative problem of learning boolean functions. A boolean function has a vector of boolean inputs and a single boolean output. Taking inspiration from mainstream machine learning work, Kaelbling developed two algorithms for learning boolean functions from reinforcement: one uses the bias of k -DNF to drive

the generalization process (Kaelbling, 1994b); the other searches the space of syntactic descriptions of functions using a simple generate-and-test method (Kaelbling, 1994a).

The restriction to a single boolean output makes these techniques difficult to apply. In very benign learning situations, it is possible to extend this approach to use a collection of learners to independently learn the individual bits that make up a complex output. In general, however, that approach suffers from the problem of very unreliable reinforcement: if a single learner generates an inappropriate output bit, all of the learners receive a low reinforcement value. The CASCADE method (Kaelbling, 1993b) allows a collection of learners to be trained collectively to generate appropriate joint outputs; it is considerably more reliable, but can require additional computational effort.

6.1.2 DELAYED REWARD

Another method to allow reinforcement-learning techniques to be applied in large state spaces is modeled on value iteration and Q-learning. Here, a function approximator is used to represent the value function by mapping a state description to a value.

Many researchers have experimented with this approach: Boyan and Moore (1995) used local memory-based methods in conjunction with value iteration; Lin (1991) used backpropagation networks for Q-learning; Watkins (1989) used CMAC for Q-learning; Tesauro (1992, 1995) used backpropagation for learning the value function in backgammon (described in Section 8.1); Zhang and Dietterich (1995) used backpropagation and $TD(\lambda)$ to learn good strategies for job-shop scheduling.

Although there have been some positive examples, in general there are unfortunate interactions between function approximation and the learning rules. In discrete environments there is a guarantee that any operation that updates the value function (according to the Bellman equations) can only reduce the error between the current value function and the optimal value function. This guarantee no longer holds when generalization is used. These issues are discussed by Boyan and Moore (1995), who give some simple examples of value function errors growing arbitrarily large when generalization is used with value iteration. Their solution to this, applicable only to certain classes of problems, discourages such divergence by only permitting updates whose estimated values can be shown to be near-optimal via a battery of Monte-Carlo experiments.

Thrun and Schwartz (1993) theorize that function approximation of value functions is also dangerous because the errors in value functions due to generalization can become compounded by the “max” operator in the definition of the value function.

Several recent results (Gordon, 1995; Tsitsiklis & Van Roy, 1996) show how the appropriate choice of function approximator can guarantee convergence, though not necessarily to the optimal values. Baird’s *residual gradient* technique (Baird, 1995) provides guaranteed convergence to locally optimal solutions.

Perhaps the gloominess of these counter-examples is misplaced. Boyan and Moore (1995) report that their counter-examples *can* be made to work with problem-specific hand-tuning despite the unreliability of untuned algorithms that provably converge in discrete domains. Sutton (1996) shows how modified versions of Boyan and Moore’s examples can converge successfully. An open question is whether general principles, ideally supported by theory, can help us understand when value function approximation will succeed. In Sutton’s com-

parative experiments with Boyan and Moore's counter-examples, he changes four aspects of the experiments:

1. Small changes to the task specifications.
2. A very different kind of function approximator (CMAC (Albus, 1975)) that has weak generalization.
3. A different learning algorithm: SARSA (Rummery & Niranjan, 1994) instead of value iteration.
4. A different training regime. Boyan and Moore sampled states uniformly in state space, whereas Sutton's method sampled along empirical trajectories.

There are intuitive reasons to believe that the fourth factor is particularly important, but more careful research is needed.

Adaptive Resolution Models In many cases, what we would like to do is partition the environment into regions of states that can be considered the same for the purposes of learning and generating actions. Without detailed prior knowledge of the environment, it is very difficult to know what granularity or placement of partitions is appropriate. This problem is overcome in methods that use adaptive resolution; during the course of learning, a partition is constructed that is appropriate to the environment.

Decision Trees In environments that are characterized by a set of boolean or discrete-valued variables, it is possible to learn compact decision trees for representing Q values. The *G-learning* algorithm (Chapman & Kaelbling, 1991), works as follows. It starts by assuming that no partitioning is necessary and tries to learn Q values for the entire environment as if it were one state. In parallel with this process, it gathers statistics based on individual input bits; it asks the question whether there is some bit b in the state description such that the Q values for states in which $b = 1$ are significantly different from Q values for states in which $b = 0$. If such a bit is found, it is used to split the decision tree. Then, the process is repeated in each of the leaves. This method was able to learn very small representations of the Q function in the presence of an overwhelming number of irrelevant, noisy state attributes. It outperformed Q-learning with backpropagation in a simple video-game environment and was used by McCallum (1995) (in conjunction with other techniques for dealing with partial observability) to learn behaviors in a complex driving-simulator. It cannot, however, acquire partitions in which attributes are only significant in combination (such as those needed to solve parity problems).

Variable Resolution Dynamic Programming The VRDP algorithm (Moore, 1991) enables conventional dynamic programming to be performed in real-valued multivariate state-spaces where straightforward discretization would fall prey to the curse of dimensionality. A kd -tree (similar to a decision tree) is used to partition state space into coarse regions. The coarse regions are refined into detailed regions, but only in parts of the state space which are predicted to be important. This notion of importance is obtained by running “mental trajectories” through state space. This algorithm proved effective on a number of problems for which full high-resolution arrays would have been impractical. It has the disadvantage of requiring a guess at an initially valid trajectory through state-space.

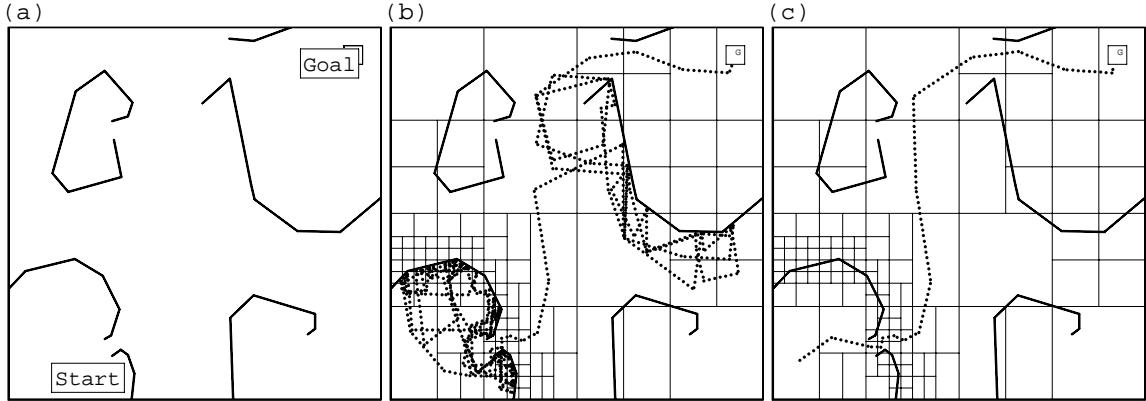


Figure 7: (a) A two-dimensional maze problem. The point robot must find a path from start to goal without crossing any of the barrier lines. (b) The path taken by PartiGame during the entire first trial. It begins with intense exploration to find a route out of the almost entirely enclosed start region. Having eventually reached a sufficiently high resolution, it discovers the gap and proceeds greedily towards the goal, only to be temporarily blocked by the goal’s barrier region. (c) The second trial.

PartiGame Algorithm Moore’s PartiGame algorithm (Moore, 1994) is another solution to the problem of learning to achieve goal configurations in deterministic high-dimensional continuous spaces by learning an adaptive-resolution model. It also divides the environment into cells; but in each cell, the actions available consist of aiming at the neighboring cells (this aiming is accomplished by a local controller, which must be provided as part of the problem statement). The graph of cell transitions is solved for shortest paths in an online incremental manner, but a minimax criterion is used to detect when a group of cells is too coarse to prevent movement between obstacles or to avoid limit cycles. The offending cells are split to higher resolution. Eventually, the environment is divided up just enough to choose appropriate actions for achieving the goal, but no unnecessary distinctions are made. An important feature is that, as well as reducing memory and computational requirements, it also structures exploration of state space in a multi-resolution manner. Given a failure, the agent will initially try something very different to rectify the failure, and only resort to small local changes when all the qualitatively different strategies have been exhausted.

Figure 7a shows a two-dimensional continuous maze. Figure 7b shows the performance of a robot using the PartiGame algorithm during the very first trial. Figure 7c shows the second trial, started from a slightly different position.

This is a very fast algorithm, learning policies in spaces of up to nine dimensions in less than a minute. The restriction of the current implementation to deterministic environments limits its applicability, however. McCallum (1995) suggests some related tree-structured methods.

6.2 Generalization over Actions

The networks described in Section 6.1.1 generalize over state descriptions presented as inputs. They also produce outputs in a discrete, factored representation and thus could be seen as generalizing over actions as well.

In cases such as this when actions are described combinatorially, it is important to generalize over actions to avoid keeping separate statistics for the huge number of actions that can be chosen. In continuous action spaces, the need for generalization is even more pronounced.

When estimating Q values using a neural network, it is possible to use either a distinct network for each action, or a network with a distinct output for each action. When the action space is continuous, neither approach is possible. An alternative strategy is to use a single network with both the state and action as input and Q value as the output. Training such a network is not conceptually difficult, but using the network to find the optimal action can be a challenge. One method is to do a local gradient-ascent search on the action in order to find one with high value (Baird & Klopff, 1993).

Gullapalli (1990, 1992) has developed a “neural” reinforcement-learning unit for use in continuous action spaces. The unit generates actions with a normal distribution; it adjusts the mean and variance based on previous experience. When the chosen actions are not performing well, the variance is high, resulting in exploration of the range of choices. When an action performs well, the mean is moved in that direction and the variance decreased, resulting in a tendency to generate more action values near the successful one. This method was successfully employed to learn to control a robot arm with many continuous degrees of freedom.

6.3 Hierarchical Methods

Another strategy for dealing with large state spaces is to treat them as a hierarchy of learning problems. In many cases, hierarchical solutions introduce slight sub-optimality in performance, but potentially gain a good deal of efficiency in execution time, learning time, and space.

Hierarchical learners are commonly structured as *gated behaviors*, as shown in Figure 8. There is a collection of *behaviors* that map environment states into low-level actions and a *gating function* that decides, based on the state of the environment, which behavior’s actions should be switched through and actually executed. Maes and Brooks (1990) used a version of this architecture in which the individual behaviors were fixed *a priori* and the gating function was learned from reinforcement. Mahadevan and Connell (1991b) used the dual approach: they fixed the gating function, and supplied reinforcement functions for the individual behaviors, which were learned. Lin (1993a) and Dorigo and Colombetti (1995, 1994) both used this approach, first training the behaviors and then training the gating function. Many of the other hierarchical learning methods can be cast in this framework.

6.3.1 FEUDAL Q-LEARNING

Feudal Q-learning (Dayan & Hinton, 1993; Watkins, 1989) involves a hierarchy of learning modules. In the simplest case, there is a high-level master and a low-level slave. The master receives reinforcement from the external environment. Its actions consist of commands that

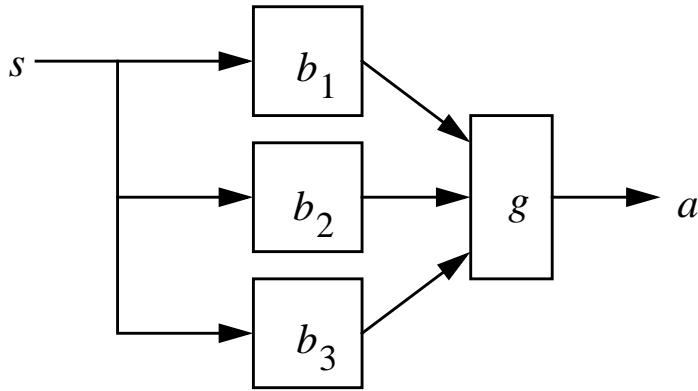


Figure 8: A structure of gated behaviors.

it can give to the low-level learner. When the master generates a particular command to the slave, it must reward the slave for taking actions that satisfy the command, even if they do not result in external reinforcement. The master, then, learns a mapping from states to commands. The slave learns a mapping from commands and states to external actions. The set of “commands” and their associated reinforcement functions are established in advance of the learning.

This is really an instance of the general “gated behaviors” approach, in which the slave can execute any of the behaviors depending on its command. The reinforcement functions for the individual behaviors (commands) are given, but learning takes place simultaneously at both the high and low levels.

6.3.2 COMPOSITIONAL Q-LEARNING

Singh’s compositional Q-learning (1992b, 1992a) (C-QL) consists of a hierarchy based on the temporal sequencing of subgoals. The *elemental tasks* are behaviors that achieve some recognizable condition. The high-level goal of the system is to achieve some set of conditions in sequential order. The achievement of the conditions provides reinforcement for the elemental tasks, which are trained first to achieve individual subgoals. Then, the gating function learns to switch the elemental tasks in order to achieve the appropriate high-level sequential goal. This method was used by Tham and Prager (1994) to learn to control a simulated multi-link robot arm.

6.3.3 HIERARCHICAL DISTANCE TO GOAL

Especially if we consider reinforcement learning modules to be part of larger agent architectures, it is important to consider problems in which goals are dynamically input to the learner. Kaelbling’s HDG algorithm (1993a) uses a hierarchical approach to solving problems when goals of achievement (the agent should get to a particular state as quickly as possible) are given to an agent dynamically.

The HDG algorithm works by analogy with navigation in a harbor. The environment is partitioned (*a priori*, but more recent work (Ashar, 1994) addresses the case of learning the partition) into a set of regions whose centers are known as “landmarks.” If the agent is

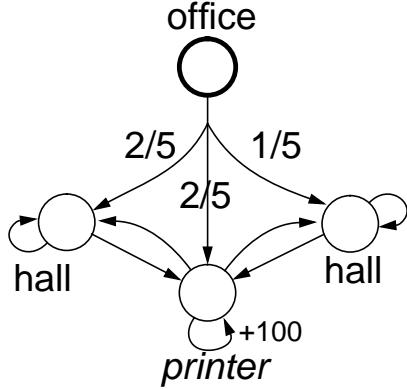


Figure 9: An example of a partially observable environment.

currently in the same region as the goal, then it uses low-level actions to move to the goal. If not, then high-level information is used to determine the next landmark on the shortest path from the agent’s closest landmark to the goal’s closest landmark. Then, the agent uses low-level information to aim toward that next landmark. If errors in action cause deviations in the path, there is no problem; the best aiming point is recomputed on every step.

7. Partially Observable Environments

In many real-world environments, it will not be possible for the agent to have perfect and complete perception of the state of the environment. Unfortunately, complete observability is necessary for learning methods based on MDPs. In this section, we consider the case in which the agent makes *observations* of the state of the environment, but these observations may be noisy and provide incomplete information. In the case of a robot, for instance, it might observe whether it is in a corridor, an open room, a T-junction, etc., and those observations might be error-prone. This problem is also referred to as the problem of “incomplete perception,” “perceptual aliasing,” or “hidden state.”

In this section, we will consider extensions to the basic MDP framework for solving partially observable problems. The resulting formal model is called a *partially observable Markov decision process* or POMDP.

7.1 State-Free Deterministic Policies

The most naive strategy for dealing with partial observability is to ignore it. That is, to treat the observations as if they were the states of the environment and try to learn to behave. Figure 9 shows a simple environment in which the agent is attempting to get to the printer from an office. If it moves from the office, there is a good chance that the agent will end up in one of two places that look like “hall”, but that require different actions for getting to the printer. If we consider these states to be the same, then the agent cannot possibly behave optimally. But how well can it do?

The resulting problem is not Markovian, and Q-learning cannot be guaranteed to converge. Small breaches of the Markov requirement are well handled by Q-learning, but it is possible to construct simple environments that cause Q-learning to oscillate (Chrisman &

Littman, 1993). It is possible to use a model-based approach, however; act according to some policy and gather statistics about the transitions between observations, then solve for the optimal policy based on those observations. Unfortunately, when the environment is not Markovian, the transition probabilities depend on the policy being executed, so this new policy will induce a new set of transition probabilities. This approach may yield plausible results in some cases, but again, there are no guarantees.

It is reasonable, though, to ask what the optimal policy (mapping from observations to actions, in this case) is. It is NP-hard (Littman, 1994b) to find this mapping, and even the best mapping can have very poor performance. In the case of our agent trying to get to the printer, for instance, any deterministic state-free policy takes an infinite number of steps to reach the goal on average.

7.2 State-Free Stochastic Policies

Some improvement can be gained by considering stochastic policies; these are mappings from observations to probability distributions over actions. If there is randomness in the agent's actions, it will not get stuck in the hall forever. Jaakkola, Singh, and Jordan (1995) have developed an algorithm for finding locally-optimal stochastic policies, but finding a globally optimal policy is still NP hard.

In our example, it turns out that the optimal stochastic policy is for the agent, when in a state that looks like a hall, to go east with probability $2 - \sqrt{2} \approx 0.6$ and west with probability $\sqrt{2} - 1 \approx 0.4$. This policy can be found by solving a simple (in this case) quadratic program. The fact that such a simple example can produce irrational numbers gives some indication that it is a difficult problem to solve exactly.

7.3 Policies with Internal State

The only way to behave truly effectively in a wide-range of environments is to use memory of previous actions and observations to disambiguate the current state. There are a variety of approaches to learning policies with internal state.

Recurrent Q-learning One intuitively simple approach is to use a recurrent neural network to learn Q values. The network can be trained using backpropagation through time (or some other suitable technique) and learns to retain “history features” to predict value. This approach has been used by a number of researchers (Meeden, McGraw, & Blank, 1993; Lin & Mitchell, 1992; Schmidhuber, 1991b). It seems to work effectively on simple problems, but can suffer from convergence to local optima on more complex problems.

Classifier Systems Classifier systems (Holland, 1975; Goldberg, 1989) were explicitly developed to solve problems with delayed reward, including those requiring short-term memory. The internal mechanism typically used to pass reward back through chains of decisions, called the *bucket brigade algorithm*, bears a close resemblance to Q-learning. In spite of some early successes, the original design does not appear to handle partially observed environments robustly.

Recently, this approach has been reexamined using insights from the reinforcement-learning literature, with some success. Dorigo did a comparative study of Q-learning and classifier systems (Dorigo & Bersini, 1994). Cliff and Ross (1994) start with Wilson's zeroth-

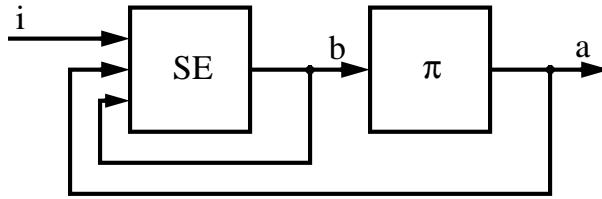


Figure 10: Structure of a POMDP agent.

level classifier system (Wilson, 1995) and add one and two-bit memory registers. They find that, although their system can learn to use short-term memory registers effectively, the approach is unlikely to scale to more complex environments.

Dorigo and Colombetti applied classifier systems to a moderately complex problem of learning robot behavior from immediate reinforcement (Dorigo, 1995; Dorigo & Colombetti, 1994).

Finite-history-window Approach One way to restore the Markov property is to allow decisions to be based on the history of recent observations and perhaps actions. Lin and Mitchell (1992) used a fixed-width finite history window to learn a pole balancing task. McCallum (1995) describes the “utile suffix memory” which learns a variable-width window that serves simultaneously as a model of the environment and a finite-memory policy. This system has had excellent results in a very complex driving-simulation domain (McCallum, 1995). Ring (1994) has a neural-network approach that uses a variable history window, adding history when necessary to disambiguate situations.

POMDP Approach Another strategy consists of using hidden Markov model (HMM) techniques to learn a model of the environment, including the hidden state, then to use that model to construct a *perfect memory* controller (Cassandra, Kaelbling, & Littman, 1994; Lovejoy, 1991; Monahan, 1982).

Chrisman (1992) showed how the forward-backward algorithm for learning HMMs could be adapted to learning POMDPs. He, and later McCallum (1993), also gave heuristic *state-splitting rules* to attempt to learn the smallest possible model for a given environment. The resulting model can then be used to integrate information from the agent’s observations in order to make decisions.

Figure 10 illustrates the basic structure for a perfect-memory controller. The component on the left is the *state estimator*, which computes the agent’s *belief state*, b as a function of the old belief state, the last action a , and the current observation i . In this context, a belief state is a probability distribution over states of the environment, indicating the likelihood, given the agent’s past experience, that the environment is actually in each of those states. The state estimator can be constructed straightforwardly using the estimated world model and Bayes’ rule.

Now we are left with the problem of finding a policy mapping belief states into action. This problem can be formulated as an MDP, but it is difficult to solve using the techniques described earlier, because the input space is continuous. Chrisman’s approach (1992) does not take into account future uncertainty, but yields a policy after a small amount of computation. A standard approach from the operations-research literature is to solve for the

optimal policy (or a close approximation thereof) based on its representation as a piecewise-linear and convex function over the belief space. This method is computationally intractable, but may serve as inspiration for methods that make further approximations (Cassandra et al., 1994; Littman, Cassandra, & Kaelbling, 1995a).

8. Reinforcement Learning Applications

One reason that reinforcement learning is popular is that it serves as a theoretical tool for studying the principles of agents learning to act. But it is unsurprising that it has also been used by a number of researchers as a practical computational tool for constructing autonomous systems that improve themselves with experience. These applications have ranged from robotics, to industrial manufacturing, to combinatorial search problems such as computer game playing.

Practical applications provide a test of the efficacy and usefulness of learning algorithms. They are also an inspiration for deciding which components of the reinforcement learning framework are of practical importance. For example, a researcher with a real robotic task can provide a data point to questions such as:

- How important is optimal exploration? Can we break the learning period into exploration phases and exploitation phases?
- What is the most useful model of long-term reward: Finite horizon? Discounted? Infinite horizon?
- How much computation is available between agent decisions and how should it be used?
- What prior knowledge can we build into the system, and which algorithms are capable of using that knowledge?

Let us examine a set of practical applications of reinforcement learning, while bearing these questions in mind.

8.1 Game Playing

Game playing has dominated the Artificial Intelligence world as a problem domain ever since the field was born. Two-player games do not fit into the established reinforcement-learning framework since the optimality criterion for games is not one of maximizing reward in the face of a fixed environment, but one of maximizing reward against an optimal adversary (minimax). Nonetheless, reinforcement-learning algorithms can be adapted to work for a very general class of games (Littman, 1994a) and many researchers have used reinforcement learning in these environments. One application, spectacularly far ahead of its time, was Samuel's checkers playing system (Samuel, 1959). This learned a value function represented by a linear function approximator, and employed a training scheme similar to the updates used in value iteration, temporal differences and Q-learning.

More recently, Tesauro (1992, 1994, 1995) applied the temporal difference algorithm to backgammon. Backgammon has approximately 10^{20} states, making table-based reinforcement learning impossible. Instead, Tesauro used a backpropagation-based three-layer

	Training Games	Hidden Units	Results
Basic			Poor
TD 1.0	300,000	80	Lost by 13 points in 51 games
TD 2.0	800,000	40	Lost by 7 points in 38 games
TD 2.1	1,500,000	80	Lost by 1 point in 40 games

Table 2: TD-Gammon’s performance in games against the top human professional players. A backgammon tournament involves playing a series of games for points until one player reaches a set target. TD-Gammon won none of these tournaments but came sufficiently close that it is now considered one of the best few players in the world.

neural network as a function approximator for the value function

$$\text{Board Position} \rightarrow \text{Probability of victory for current player.}$$

Two versions of the learning algorithm were used. The first, which we will call Basic TD-Gammon, used very little predefined knowledge of the game, and the representation of a board position was virtually a raw encoding, sufficiently powerful only to permit the neural network to distinguish between conceptually different positions. The second, TD-Gammon, was provided with the same raw state information supplemented by a number of hand-crafted features of backgammon board positions. Providing hand-crafted features in this manner is a good example of how inductive biases from human knowledge of the task can be supplied to a learning algorithm.

The training of both learning algorithms required several months of computer time, and was achieved by constant self-play. No exploration strategy was used—the system always greedily chose the move with the largest expected probability of victory. This naive exploration strategy proved entirely adequate for this environment, which is perhaps surprising given the considerable work in the reinforcement-learning literature which has produced numerous counter-examples to show that greedy exploration can lead to poor learning performance. Backgammon, however, has two important properties. Firstly, whatever policy is followed, every game is guaranteed to end in finite time, meaning that useful reward information is obtained fairly frequently. Secondly, the state transitions are sufficiently stochastic that independent of the policy, all states will occasionally be visited—a wrong initial value function has little danger of starving us from visiting a critical part of state space from which important information could be obtained.

The results (Table 2) of TD-Gammon are impressive. It has competed at the very top level of international human play. Basic TD-Gammon played respectably, but not at a professional standard.

Figure 11: Schaal and Atkeson’s devil-sticking robot. The tapered stick is hit alternately by each of the two hand sticks. The task is to keep the devil stick from falling for as many hits as possible. The robot has three motors indicated by torque vectors τ_1, τ_2, τ_3 .

Although experiments with other games have in some cases produced interesting learning behavior, no success close to that of TD-Gammon has been repeated. Other games that have been studied include Go (Schraudolph, Dayan, & Sejnowski, 1994) and Chess (Thrun, 1995). It is still an open question as to if and how the success of TD-Gammon can be repeated in other domains.

8.2 Robotics and Control

In recent years there have been many robotics and control applications that have used reinforcement learning. Here we will concentrate on the following four examples, although many other interesting ongoing robotics investigations are underway.

1. Schaal and Atkeson (1994) constructed a two-armed robot, shown in Figure 11, that learns to juggle a device known as a devil-stick. This is a complex non-linear control task involving a six-dimensional state space and less than 200 msecs per control decision. After about 40 initial attempts the robot learns to keep juggling for hundreds of hits. A typical human learning the task requires an order of magnitude more practice to achieve proficiency at mere tens of hits.

The juggling robot learned a world model from experience, which was generalized to unvisited states by a function approximation scheme known as locally weighted regression (Cleveland & Delvin, 1988; Moore & Atkeson, 1992). Between each trial, a form of dynamic programming specific to linear control policies and locally linear transitions was used to improve the policy. The form of dynamic programming is known as linear-quadratic-regulator design (Sage & White, 1977).

2. Mahadevan and Connell (1991a) discuss a task in which a mobile robot pushes large boxes for extended periods of time. Box-pushing is a well-known difficult robotics problem, characterized by immense uncertainty in the results of actions. Q-learning was used in conjunction with some novel clustering techniques designed to enable a higher-dimensional input than a tabular approach would have permitted. The robot learned to perform competitively with the performance of a human-programmed solution. Another aspect of this work, mentioned in Section 6.3, was a pre-programmed breakdown of the monolithic task description into a set of lower level tasks to be learned.
3. Mataric (1994) describes a robotics experiment with, from the viewpoint of theoretical reinforcement learning, an unthinkably high dimensional state space, containing many dozens of degrees of freedom. Four mobile robots traveled within an enclosure collecting small disks and transporting them to a destination region. There were three enhancements to the basic Q-learning algorithm. Firstly, pre-programmed signals called *progress estimators* were used to break the monolithic task into subtasks. This was achieved in a robust manner in which the robots were not forced to use the estimators, but had the freedom to profit from the inductive bias they provided. Secondly, control was decentralized. Each robot learned its own policy independently without explicit communication with the others. Thirdly, state space was brutally quantized into a small number of discrete states according to values of a small number of pre-programmed boolean features of the underlying sensors. The performance of the Q-learned policies were almost as good as a simple hand-crafted controller for the job.
4. Q-learning has been used in an elevator dispatching task (Crites & Barto, 1996). The problem, which has been implemented in simulation only at this stage, involved four elevators servicing ten floors. The objective was to minimize the average squared wait time for passengers, discounted into future time. The problem can be posed as a discrete Markov system, but there are 10^{22} states even in the most simplified version of the problem. Crites and Barto used neural networks for function approximation and provided an excellent comparison study of their Q-learning approach against the most popular and the most sophisticated elevator dispatching algorithms. The squared wait time of their controller was approximately 7% less than the best alternative algorithm (“Empty the System” heuristic with a receding horizon controller) and less than half the squared wait time of the controller most frequently used in real elevator systems.
5. The final example concerns an application of reinforcement learning by one of the authors of this survey to a packaging task from a food processing industry. The problem involves filling containers with variable numbers of non-identical products. The product characteristics also vary with time, but can be sensed. Depending on the task, various constraints are placed on the container-filling procedure. Here are three examples:
 - The mean weight of all containers produced by a shift must not be below the manufacturer’s declared weight W .

- The number of containers below the declared weight must be less than $P\%$.
- No containers may be produced below weight W' .

Such tasks are controlled by machinery which operates according to various *setpoints*. Conventional practice is that setpoints are chosen by human operators, but this choice is not easy as it is dependent on the current product characteristics and the current task constraints. The dependency is often difficult to model and highly non-linear. The task was posed as a finite-horizon Markov decision task in which the state of the system is a function of the product characteristics, the amount of time remaining in the production shift and the mean wastage and percent below declared in the shift so far. The system was discretized into 200,000 discrete states and local weighted regression was used to learn and generalize a transition model. Prioritized sweeping was used to maintain an optimal value function as each new piece of transition information was obtained. In simulated experiments the savings were considerable, typically with wastage reduced by a factor of ten. Since then the system has been deployed successfully in several factories within the United States.

Some interesting aspects of practical reinforcement learning come to light from these examples. The most striking is that in all cases, to make a real system work it proved necessary to supplement the fundamental algorithm with extra pre-programmed knowledge. Supplying extra knowledge comes at a price: more human effort and insight is required and the system is subsequently less autonomous. But it is also clear that for tasks such as these, a knowledge-free approach would not have achieved worthwhile performance within the finite lifetime of the robots.

What forms did this pre-programmed knowledge take? It included an assumption of linearity for the juggling robot's policy, a manual breaking up of the task into subtasks for the two mobile-robot examples, while the box-pusher also used a clustering technique for the Q values which assumed locally consistent Q values. The four disk-collecting robots additionally used a manually discretized state space. The packaging example had far fewer dimensions and so required correspondingly weaker assumptions, but there, too, the assumption of local piecewise continuity in the transition model enabled massive reductions in the amount of learning data required.

The exploration strategies are interesting too. The juggler used careful statistical analysis to judge where to profitably experiment. However, both mobile robot applications were able to learn well with greedy exploration—always exploiting without deliberate exploration. The packaging task used optimism in the face of uncertainty. None of these strategies mirrors theoretically optimal (but computationally intractable) exploration, and yet all proved adequate.

Finally, it is also worth considering the computational regimes of these experiments. They were all very different, which indicates that the differing computational demands of various reinforcement learning algorithms do indeed have an array of differing applications. The juggler needed to make very fast decisions with low latency between each hit, but had long periods (30 seconds and more) between each trial to consolidate the experiences collected on the previous trial and to perform the more aggressive computation necessary to produce a new reactive controller on the next trial. The box-pushing robot was meant to

operate autonomously for hours and so had to make decisions with a uniform length control cycle. The cycle was sufficiently long for quite substantial computations beyond simple Q-learning backups. The four disk-collecting robots were particularly interesting. Each robot had a short life of less than 20 minutes (due to battery constraints) meaning that substantial number crunching was impractical, and any significant combinatorial search would have used a significant fraction of the robot's learning lifetime. The packaging task had easy constraints. One decision was needed every few minutes. This provided opportunities for fully computing the optimal value function for the 200,000-state system between every control cycle, in addition to performing massive cross-validation-based optimization of the transition model being learned.

A great deal of further work is currently in progress on practical implementations of reinforcement learning. The insights and task constraints that they produce will have an important effect on shaping the kind of algorithms that are developed in future.

9. Conclusions

There are a variety of reinforcement-learning techniques that work effectively on a variety of small problems. But very few of these techniques scale well to larger problems. This is not because researchers have done a bad job of inventing learning techniques, but because it is very difficult to solve arbitrary problems in the general case. In order to solve highly complex problems, we must give up *tabula rasa* learning techniques and begin to incorporate bias that will give leverage to the learning process.

The necessary bias can come in a variety of forms, including the following:

shaping: The technique of shaping is used in training animals (Hilgard & Bower, 1975); a teacher presents very simple problems to solve first, then gradually exposes the learner to more complex problems. Shaping has been used in supervised-learning systems, and can be used to train hierarchical reinforcement-learning systems from the bottom up (Lin, 1991), and to alleviate problems of delayed reinforcement by decreasing the delay until the problem is well understood (Dorigo & Colombetti, 1994; Dorigo, 1995).

local reinforcement signals: Whenever possible, agents should be given reinforcement signals that are local. In applications in which it is possible to compute a gradient, rewarding the agent for taking steps up the gradient, rather than just for achieving the final goal, can speed learning significantly (Mataric, 1994).

imitation: An agent can learn by “watching” another agent perform the task (Lin, 1991). For real robots, this requires perceptual abilities that are not yet available. But another strategy is to have a human supply appropriate motor commands to a robot through a joystick or steering wheel (Pomerleau, 1993).

problem decomposition: Decomposing a huge learning problem into a collection of smaller ones, and providing useful reinforcement signals for the subproblems is a very powerful technique for biasing learning. Most interesting examples of robotic reinforcement learning employ this technique to some extent (Connell & Mahadevan, 1993).

reflexes: One thing that keeps agents that know nothing from learning anything is that they have a hard time even finding the interesting parts of the space; they wander

around at random never getting near the goal, or they are always “killed” immediately. These problems can be ameliorated by programming a set of “reflexes” that cause the agent to act initially in some way that is reasonable (Mataric, 1994; Singh, Barto, Grupen, & Connolly, 1994). These reflexes can eventually be overridden by more detailed and accurate learned knowledge, but they at least keep the agent alive and pointed in the right direction while it is trying to learn. Recent work by Millan (1996) explores the use of reflexes to make robot learning safer and more efficient.

With appropriate biases, supplied by human programmers or teachers, complex reinforcement-learning problems will eventually be solvable. There is still much work to be done and many interesting questions remaining for learning techniques and especially regarding methods for approximating, decomposing, and incorporating bias into problems.

Acknowledgements

Thanks to Marco Dorigo and three anonymous reviewers for comments that have helped to improve this paper. Also thanks to our many colleagues in the reinforcement-learning community who have done this work and explained it to us.

Leslie Pack Kaelbling was supported in part by NSF grants IRI-9453383 and IRI-9312395. Michael Littman was supported in part by Bellcore. Andrew Moore was supported in part by an NSF Research Initiation Award and by 3M Corporation.

References

- Ackley, D. H., & Littman, M. L. (1990). Generalization and scaling in reinforcement learning. In Touretzky, D. S. (Ed.), *Advances in Neural Information Processing Systems 2*, pp. 550–557 San Mateo, CA. Morgan Kaufmann.
- Albus, J. S. (1975). A new approach to manipulator control: Cerebellar model articulation controller (cmac). *Journal of Dynamic Systems, Measurement and Control*, 97, 220–227.
- Albus, J. S. (1981). *Brains, Behavior, and Robotics*. BYTE Books, Subsidiary of McGraw-Hill, Peterborough, New Hampshire.
- Anderson, C. W. (1986). *Learning and Problem Solving with Multilayer Connectionist Systems*. Ph.D. thesis, University of Massachusetts, Amherst, MA.
- Ashar, R. R. (1994). Hierarchical learning in stochastic domains. Master’s thesis, Brown University, Providence, Rhode Island.
- Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In Prieditis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 30–37 San Francisco, CA. Morgan Kaufmann.
- Baird, L. C., & Klopff, A. H. (1993). Reinforcement learning with high-dimensional, continuous actions. Tech. rep. WL-TR-93-1147, Wright-Patterson Air Force Base Ohio: Wright Laboratory.

- Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1), 81–138.
- Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5), 834–846.
- Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.
- Berenji, H. R. (1991). Artificial neural networks and approximate reasoning for intelligent control in space. In *American Control Conference*, pp. 1075–1080.
- Berry, D. A., & Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, London, UK.
- Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ.
- Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, Massachusetts. Volumes 1 and 2.
- Bertsekas, D. P., & Castañon, D. A. (1989). Adaptive aggregation for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, 34(6), 589–598.
- Bertsekas, D. P., & Tsitsiklis, J. N. (1989). *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ.
- Box, G. E. P., & Draper, N. R. (1987). *Empirical Model-Building and Response Surfaces*. Wiley.
- Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D. S., & Leen, T. K. (Eds.), *Advances in Neural Information Processing Systems 7* Cambridge, MA. The MIT Press.
- Burghes, D., & Graham, A. (1980). *Introduction to Control Theory including Optimal Control*. Ellis Horwood.
- Cassandra, A. R., Kaelbling, L. P., & Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* Seattle, WA.
- Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the International Joint Conference on Artificial Intelligence* Sydney, Australia.
- Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 183–188 San Jose, CA. AAAI Press.

- Chrisman, L., & Littman, M. (1993). Hidden state and short-term memory.. Presentation at Reinforcement Learning Workshop, Machine Learning Conference.
- Cichosz, P., & Mulawka, J. J. (1995). Fast and efficient reinforcement learning with truncated temporal differences. In Prieditis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 99–107 San Francisco, CA. Morgan Kaufmann.
- Cleveland, W. S., & Delvin, S. J. (1988). Locally weighted regression: An approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403), 596–610.
- Cliff, D., & Ross, S. (1994). Adding temporary memory to ZCS. *Adaptive Behavior*, 3(2), 101–150.
- Condon, A. (1992). The complexity of stochastic games. *Information and Computation*, 96(2), 203–224.
- Connell, J., & Mahadevan, S. (1993). Rapid task learning for real robots. In *Robot Learning*. Kluwer Academic Publishers.
- Crites, R. H., & Barto, A. G. (1996). Improving elevator performance using reinforcement learning. In Touretzky, D., Mozer, M., & Hasselmo, M. (Eds.), *Neural Information Processing Systems 8*.
- Dayan, P. (1992). The convergence of $\text{TD}(\lambda)$ for general λ . *Machine Learning*, 8(3), 341–362.
- Dayan, P., & Hinton, G. E. (1993). Feudal reinforcement learning. In Hanson, S. J., Cowan, J. D., & Giles, C. L. (Eds.), *Advances in Neural Information Processing Systems 5* San Mateo, CA. Morgan Kaufmann.
- Dayan, P., & Sejnowski, T. J. (1994). $\text{TD}(\lambda)$ converges with probability 1. *Machine Learning*, 14(3).
- Dean, T., Kaelbling, L. P., Kirman, J., & Nicholson, A. (1993). Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence* Washington, DC.
- D'Epenoux, F. (1963). A probabilistic production and inventory problem. *Management Science*, 10, 98–108.
- Derman, C. (1970). *Finite State Markovian Decision Processes*. Academic Press, New York.
- Dorigo, M., & Bersini, H. (1994). A comparison of q-learning and classifier systems. In *From Animals to Animats: Proceedings of the Third International Conference on the Simulation of Adaptive Behavior* Brighton, UK.
- Dorigo, M., & Colombetti, M. (1994). Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2), 321–370.

- Dorigo, M. (1995). Alecsys and the AutonoMouse: Learning to control a real robot by distributed classifier systems. *Machine Learning*, 19.
- Fiechter, C.-N. (1994). Efficient reinforcement learning. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory*, pp. 88–97. Association of Computing Machinery.
- Gittins, J. C. (1989). *Multi-armed Bandit Allocation Indices*. Wiley-Interscience series in systems and optimization. Wiley, Chichester, NY.
- Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, MA.
- Gordon, G. J. (1995). Stable function approximation in dynamic programming. In Priditidis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 261–268 San Francisco, CA. Morgan Kaufmann.
- Gullapalli, V. (1990). A stochastic reinforcement learning algorithm for learning real-valued functions. *Neural Networks*, 3, 671–692.
- Gullapalli, V. (1992). *Reinforcement learning and its application to control*. Ph.D. thesis, University of Massachusetts, Amherst, MA.
- Hilgard, E. R., & Bower, G. H. (1975). *Theories of Learning* (fourth edition). Prentice-Hall, Englewood Cliffs, NJ.
- Hoffman, A. J., & Karp, R. M. (1966). On nonterminating stochastic games. *Management Science*, 12, 359–370.
- Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.
- Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. The MIT Press, Cambridge, MA.
- Jaakkola, T., Jordan, M. I., & Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6).
- Jaakkola, T., Singh, S. P., & Jordan, M. I. (1995). Monte-carlo reinforcement learning in non-Markovian decision problems. In Tesauro, G., Touretzky, D. S., & Leen, T. K. (Eds.), *Advances in Neural Information Processing Systems 7* Cambridge, MA. The MIT Press.
- Kaelbling, L. P. (1993a). Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning* Amherst, MA. Morgan Kaufmann.
- Kaelbling, L. P. (1993b). *Learning in Embedded Systems*. The MIT Press, Cambridge, MA.
- Kaelbling, L. P. (1994a). Associative reinforcement learning: A generate and test algorithm. *Machine Learning*, 15(3).

- Kaelbling, L. P. (1994b). Associative reinforcement learning: Functions in k -DNF. *Machine Learning*, 15(3).
- Kirman, J. (1994). *Predicting Real-Time Planner Performance by Domain Characterization*. Ph.D. thesis, Department of Computer Science, Brown University.
- Koenig, S., & Simmons, R. G. (1993). Complexity analysis of real-time reinforcement learning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 99–105 Menlo Park, California. AAAI Press/MIT Press.
- Kumar, P. R., & Varaiya, P. P. (1986). *Stochastic Systems: Estimation, Identification, and Adaptive Control*. Prentice Hall, Englewood Cliffs, New Jersey.
- Lee, C. C. (1991). A self learning rule-based controller employing approximate reasoning and neural net concepts. *International Journal of Intelligent Systems*, 6(1), 71–93.
- Lin, L.-J. (1991). Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence*.
- Lin, L.-J. (1993a). Hierarchical learning of robot skills by reinforcement. In *Proceedings of the International Conference on Neural Networks*.
- Lin, L.-J. (1993b). *Reinforcement Learning for Robots Using Neural Networks*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.
- Lin, L.-J., & Mitchell, T. M. (1992). Memory approaches to reinforcement learning in non-Markovian domains. Tech. rep. CMU-CS-92-138, Carnegie Mellon University, School of Computer Science.
- Littman, M. L. (1994a). Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 157–163 San Francisco, CA. Morgan Kaufmann.
- Littman, M. L. (1994b). Memoryless policies: Theoretical limitations and practical results. In Cliff, D., Husbands, P., Meyer, J.-A., & Wilson, S. W. (Eds.), *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior* Cambridge, MA. The MIT Press.
- Littman, M. L., Cassandra, A., & Kaelbling, L. P. (1995a). Learning policies for partially observable environments: Scaling up. In Priedtis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 362–370 San Francisco, CA. Morgan Kaufmann.
- Littman, M. L., Dean, T. L., & Kaelbling, L. P. (1995b). On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)* Montreal, Québec, Canada.
- Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research*, 28, 47–66.

- Maes, P., & Brooks, R. A. (1990). Learning to coordinate behaviors. In *Proceedings Eighth National Conference on Artificial Intelligence*, pp. 796–802. Morgan Kaufmann.
- Mahadevan, S. (1994). To discount or not to discount in reinforcement learning: A case study comparing R learning and Q learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 164–172 San Francisco, CA. Morgan Kaufmann.
- Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1).
- Mahadevan, S., & Connell, J. (1991a). Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence* Anaheim, CA.
- Mahadevan, S., & Connell, J. (1991b). Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 328–332.
- Mataric, M. J. (1994). Reward functions for accelerated learning. In Cohen, W. W., & Hirsh, H. (Eds.), *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann.
- McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. thesis, Department of Computer Science, University of Rochester.
- McCallum, R. A. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 190–196 Amherst, Massachusetts. Morgan Kaufmann.
- McCallum, R. A. (1995). Instance-based utile distinctions for reinforcement learning with hidden state. In *Proceedings of the Twelfth International Conference Machine Learning*, pp. 387–395 San Francisco, CA. Morgan Kaufmann.
- Meeden, L., McGraw, G., & Blank, D. (1993). Emergent control and planning in an autonomous vehicle. In Touretsky, D. (Ed.), *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pp. 735–740. Lawerence Erlbaum Associates, Hillsdale, NJ.
- Millan, J. d. R. (1996). Rapid, safe, and incremental learning of navigation strategies. *IEEE Transactions on Systems, Man, and Cybernetics*, 26(3).
- Monahan, G. E. (1982). A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28, 1–16.
- Moore, A. W. (1991). Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued spaces. In *Proc. Eighth International Machine Learning Workshop*.

- Moore, A. W. (1994). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. In Cowan, J. D., Tesauro, G., & Alspector, J. (Eds.), *Advances in Neural Information Processing Systems 6*, pp. 711–718 San Mateo, CA. Morgan Kaufmann.
- Moore, A. W., & Atkeson, C. G. (1992). An investigation of memory-based function approximators for learning control. Tech. rep., MIT Artificial Intelligence Laboratory, Cambridge, MA.
- Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13.
- Moore, A. W., Atkeson, C. G., & Schaal, S. (1995). Memory-based learning for control. Tech. rep. CMU-RI-TR-95-18, CMU Robotics Institute.
- Narendra, K., & Thathachar, M. A. L. (1989). *Learning Automata: An Introduction*. Prentice-Hall, Englewood Cliffs, NJ.
- Narendra, K. S., & Thathachar, M. A. L. (1974). Learning automata—a survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4(4), 323–334.
- Peng, J., & Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4), 437–454.
- Peng, J., & Williams, R. J. (1994). Incremental multi-step Q-learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 226–232 San Francisco, CA. Morgan Kaufmann.
- Pomerleau, D. A. (1993). *Neural network perception for mobile robot guidance*. Kluwer Academic Publishing.
- Puterman, M. L. (1994). *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY.
- Puterman, M. L., & Shin, M. C. (1978). Modified policy iteration algorithms for discounted Markov decision processes. *Management Science*, 24, 1127–1137.
- Ring, M. B. (1994). *Continual Learning in Reinforcement Environments*. Ph.D. thesis, University of Texas at Austin, Austin, Texas.
- Rüde, U. (1993). *Mathematical and computational techniques for multilevel adaptive methods*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.
- Rumelhart, D. E., & McClelland, J. L. (Eds.). (1986). *Parallel Distributed Processing: Explorations in the microstructures of cognition. Volume 1: Foundations*. The MIT Press, Cambridge, MA.
- Rummery, G. A., & Niranjan, M. (1994). On-line Q-learning using connectionist systems. Tech. rep. CUED/F-INFENG/TR166, Cambridge University.

- Rust, J. (1996). Numerical dynamic programming in economics. In *Handbook of Computational Economics*. Elsevier, North Holland.
- Sage, A. P., & White, C. C. (1977). *Optimum Systems Control*. Prentice Hall.
- Salganicoff, M., & Ungar, L. H. (1995). Active exploration and learning in real-valued spaces using multi-armed bandit allocation indices. In Prieditis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 480–487 San Francisco, CA. Morgan Kaufmann.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3, 211–229. Reprinted in E. A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, McGraw-Hill, New York 1963.
- Schaal, S., & Atkeson, C. (1994). Robot juggling: An implementation of memory-based learning. *Control Systems Magazine*, 14.
- Schmidhuber, J. (1996). A general method for multi-agent learning and incremental self-improvement in unrestricted environments. In Yao, X. (Ed.), *Evolutionary Computation: Theory and Applications*. Scientific Publ. Co., Singapore.
- Schmidhuber, J. H. (1991a). Curious model-building control systems. In *Proc. International Joint Conference on Neural Networks, Singapore*, Vol. 2, pp. 1458–1463. IEEE.
- Schmidhuber, J. H. (1991b). Reinforcement learning in Markovian and non-Markovian environments. In Lippman, D. S., Moody, J. E., & Touretzky, D. S. (Eds.), *Advances in Neural Information Processing Systems 3*, pp. 500–506 San Mateo, CA. Morgan Kaufmann.
- Schraudolph, N. N., Dayan, P., & Sejnowski, T. J. (1994). Temporal difference learning of position evaluation in the game of Go. In Cowan, J. D., Tesauro, G., & Alspector, J. (Eds.), *Advances in Neural Information Processing Systems 6*, pp. 817–824 San Mateo, CA. Morgan Kaufmann.
- Schrijver, A. (1986). *Theory of Linear and Integer Programming*. Wiley-Interscience, New York, NY.
- Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 298–305 Amherst, Massachusetts. Morgan Kaufmann.
- Singh, S. P., Barto, A. G., Grupen, R., & Connolly, C. (1994). Robust reinforcement learning in motion planning. In Cowan, J. D., Tesauro, G., & Alspector, J. (Eds.), *Advances in Neural Information Processing Systems 6*, pp. 655–662 San Mateo, CA. Morgan Kaufmann.
- Singh, S. P., & Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1).

- Singh, S. P. (1992a). Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 202–207 San Jose, CA. AAAI Press.
- Singh, S. P. (1992b). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3), 323–340.
- Singh, S. P. (1993). *Learning to Solve Markovian Decision Processes*. Ph.D. thesis, Department of Computer Science, University of Massachusetts. Also, CMPSCI Technical Report 93-77.
- Stengel, R. F. (1986). *Stochastic Optimal Control*. John Wiley and Sons.
- Sutton, R. S. (1996). Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In Touretzky, D., Mozer, M., & Hasselmo, M. (Eds.), *Neural Information Processing Systems 8*.
- Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning*. Ph.D. thesis, University of Massachusetts, Amherst, MA.
- Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3(1), 9–44.
- Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning* Austin, TX. Morgan Kaufmann.
- Sutton, R. S. (1991). Planning by incremental dynamic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 353–357. Morgan Kaufmann.
- Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8, 257–277.
- Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2), 215–219.
- Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), 58–67.
- Tham, C.-K., & Prager, R. W. (1994). A modular q-learning architecture for manipulator task decomposition. In *Proceedings of the Eleventh International Conference on Machine Learning* San Francisco, CA. Morgan Kaufmann.
- Thrun, S. (1995). Learning to play the game of chess. In Tesauro, G., Touretzky, D. S., & Leen, T. K. (Eds.), *Advances in Neural Information Processing Systems 7* Cambridge, MA. The MIT Press.

- Thrun, S., & Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. In Mozer, M., Smolensky, P., Touretzky, D., Elman, J., & Weigend, A. (Eds.), *Proceedings of the 1993 Connectionist Models Summer School* Hillsdale, NJ. Lawrence Erlbaum.
- Thrun, S. B. (1992). The role of exploration in learning control. In White, D. A., & Sofge, D. A. (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. Van Nostrand Reinhold, New York, NY.
- Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16(3).
- Tsitsiklis, J. N., & Van Roy, B. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1).
- Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11), 1134–1142.
- Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Cambridge, UK.
- Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3), 279–292.
- Whitehead, S. D. (1991). Complexity and cooperation in Q-learning. In *Proceedings of the Eighth International Workshop on Machine Learning* Evanston, IL. Morgan Kaufmann.
- Williams, R. J. (1987). A class of gradient-estimating algorithms for reinforcement learning in neural networks. In *Proceedings of the IEEE First International Conference on Neural Networks* San Diego, CA.
- Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3), 229–256.
- Williams, R. J., & Baird, III, L. C. (1993a). Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems. Tech. rep. NU-CCS-93-11, Northeastern University, College of Computer Science, Boston, MA.
- Williams, R. J., & Baird, III, L. C. (1993b). Tight performance bounds on greedy policies based on imperfect value functions. Tech. rep. NU-CCS-93-14, Northeastern University, College of Computer Science, Boston, MA.
- Wilson, S. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2), 147–173.
- Zhang, W., & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Reinforcement Learning

An Introduction

second edition

Richard S. Sutton and Andrew G. Barto



Adaptive Computation and Machine Learning

Francis Bach, series editor

A complete list of books published in the Adaptive Computation and Machine Learning series appears at the back of this book.

Reinforcement Learning:

An Introduction

second edition

Richard S. Sutton and Andrew G. Barto

The MIT Press

Cambridge, Massachusetts

London, England

© 2018, 2020 Richard S. Sutton and Andrew G. Barto

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the copyright holder. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 2.0 Generic License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/2.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

This book was set in 10/12, CMR by Westchester Publishing Services. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Names: Sutton, Richard S., author. | Barto, Andrew G., author.

Title: Reinforcement learning : an introduction / Richard S. Sutton and Andrew G. Barto.

Description: Second edition. | Cambridge, MA : The MIT Press, [2018] | Series: Adaptive computation and machine learning series | Includes bibliographical references and index.

Identifiers: LCCN 2018023826 | ISBN 9780262039246 (hardcover : alk. paper)

Subjects: LCSH: Reinforcement learning.

Classification: LCC Q325.6 .R45 2018 | DDC 006.3/1--dc23 LC record available

at <https://lccn.loc.gov/2018023826>

10 9 8 7 6 5 4 3 2 1

In memory of A. Harry Klopf

Contents

Preface to the Second Edition	xiii
Preface to the First Edition	xvii
Summary of Notation	xix
1 Introduction	1
1.1 Reinforcement Learning	1
1.2 Examples	4
1.3 Elements of Reinforcement Learning	6
1.4 Limitations and Scope	7
1.5 An Extended Example: Tic-Tac-Toe	8
1.6 Summary	13
1.7 Early History of Reinforcement Learning	13
I Tabular Solution Methods	23
2 Multi-armed Bandits	25
2.1 A k -armed Bandit Problem	25
2.2 Action-value Methods	27
2.3 The 10-armed Testbed	28
2.4 Incremental Implementation	30
2.5 Tracking a Nonstationary Problem	32
2.6 Optimistic Initial Values	34
2.7 Upper-Confidence-Bound Action Selection	35
2.8 Gradient Bandit Algorithms	37
2.9 Associative Search (Contextual Bandits)	41
2.10 Summary	42

3 Finite Markov Decision Processes	47
3.1 The Agent–Environment Interface	47
3.2 Goals and Rewards	53
3.3 Returns and Episodes	54
3.4 Unified Notation for Episodic and Continuing Tasks	57
3.5 Policies and Value Functions	58
3.6 Optimal Policies and Optimal Value Functions	62
3.7 Optimality and Approximation	67
3.8 Summary	68
4 Dynamic Programming	73
4.1 Policy Evaluation (Prediction)	74
4.2 Policy Improvement	76
4.3 Policy Iteration	80
4.4 Value Iteration	82
4.5 Asynchronous Dynamic Programming	85
4.6 Generalized Policy Iteration	86
4.7 Efficiency of Dynamic Programming	87
4.8 Summary	88
5 Monte Carlo Methods	91
5.1 Monte Carlo Prediction	92
5.2 Monte Carlo Estimation of Action Values	96
5.3 Monte Carlo Control	97
5.4 Monte Carlo Control without Exploring Starts	100
5.5 Off-policy Prediction via Importance Sampling	103
5.6 Incremental Implementation	109
5.7 Off-policy Monte Carlo Control	110
5.8 *Discounting-aware Importance Sampling	112
5.9 *Per-decision Importance Sampling	114
5.10 Summary	115
6 Temporal-Difference Learning	119
6.1 TD Prediction	119
6.2 Advantages of TD Prediction Methods	124
6.3 Optimality of TD(0)	126
6.4 Sarsa: On-policy TD Control	129
6.5 Q-learning: Off-policy TD Control	131
6.6 Expected Sarsa	133
6.7 Maximization Bias and Double Learning	134
6.8 Games, Afterstates, and Other Special Cases	136
6.9 Summary	138

7 n-step Bootstrapping	141
7.1 n-step TD Prediction	142
7.2 n-step Sarsa	145
7.3 n-step Off-policy Learning	148
7.4 *Per-decision Methods with Control Variates	150
7.5 Off-policy Learning Without Importance Sampling: The n -step Tree Backup Algorithm	152
7.6 *A Unifying Algorithm: n -step $Q(\sigma)$	154
7.7 Summary	157
8 Planning and Learning with Tabular Methods	159
8.1 Models and Planning	159
8.2 Dyna: Integrated Planning, Acting, and Learning	161
8.3 When the Model Is Wrong	166
8.4 Prioritized Sweeping	168
8.5 Expected vs. Sample Updates	172
8.6 Trajectory Sampling	174
8.7 Real-time Dynamic Programming	177
8.8 Planning at Decision Time	180
8.9 Heuristic Search	181
8.10 Rollout Algorithms	183
8.11 Monte Carlo Tree Search	185
8.12 Summary of the Chapter	188
8.13 Summary of Part I: Dimensions	189
II Approximate Solution Methods	195
9 On-policy Prediction with Approximation	197
9.1 Value-function Approximation	198
9.2 The Prediction Objective (\overline{VE})	199
9.3 Stochastic-gradient and Semi-gradient Methods	200
9.4 Linear Methods	204
9.5 Feature Construction for Linear Methods	210
9.5.1 Polynomials	210
9.5.2 Fourier Basis	211
9.5.3 Coarse Coding	215
9.5.4 Tile Coding	217
9.5.5 Radial Basis Functions	221
9.6 Selecting Step-Size Parameters Manually	222
9.7 Nonlinear Function Approximation: Artificial Neural Networks	223
9.8 Least-Squares TD	228

9.9	Memory-based Function Approximation	230
9.10	Kernel-based Function Approximation	232
9.11	Looking Deeper at On-policy Learning: Interest and Emphasis	234
9.12	Summary	236
10	On-policy Control with Approximation	243
10.1	Episodic Semi-gradient Control	243
10.2	Semi-gradient n -step Sarsa	247
10.3	Average Reward: A New Problem Setting for Continuing Tasks	249
10.4	Deprecating the Discounted Setting	253
10.5	Differential Semi-gradient n -step Sarsa	255
10.6	Summary	256
11	*Off-policy Methods with Approximation	257
11.1	Semi-gradient Methods	258
11.2	Examples of Off-policy Divergence	260
11.3	The Deadly Triad	264
11.4	Linear Value-function Geometry	266
11.5	Gradient Descent in the Bellman Error	269
11.6	The Bellman Error is Not Learnable	274
11.7	Gradient-TD Methods	278
11.8	Emphatic-TD Methods	281
11.9	Reducing Variance	283
11.10	Summary	284
12	Eligibility Traces	287
12.1	The λ -return	288
12.2	TD(λ)	292
12.3	n -step Truncated λ -return Methods	295
12.4	Redoing Updates: Online λ -return Algorithm	297
12.5	True Online TD(λ)	299
12.6	*Dutch Traces in Monte Carlo Learning	301
12.7	Sarsa(λ)	303
12.8	Variable λ and γ	307
12.9	Off-policy Traces with Control Variates	309
12.10	Watkins's Q(λ) to Tree-Backup(λ)	312
12.11	Stable Off-policy Methods with Traces	314
12.12	Implementation Issues	316
12.13	Conclusions	317

13 Policy Gradient Methods	321
13.1 Policy Approximation and its Advantages	322
13.2 The Policy Gradient Theorem	324
13.3 REINFORCE: Monte Carlo Policy Gradient	326
13.4 REINFORCE with Baseline	329
13.5 Actor–Critic Methods	331
13.6 Policy Gradient for Continuing Problems	333
13.7 Policy Parameterization for Continuous Actions	335
13.8 Summary	337
III Looking Deeper	339
14 Psychology	341
14.1 Prediction and Control	342
14.2 Classical Conditioning	343
14.2.1 Blocking and Higher-order Conditioning	345
14.2.2 The Rescorla–Wagner Model	346
14.2.3 The TD Model	349
14.2.4 TD Model Simulations	350
14.3 Instrumental Conditioning	357
14.4 Delayed Reinforcement	361
14.5 Cognitive Maps	363
14.6 Habitual and Goal-directed Behavior	364
14.7 Summary	368
15 Neuroscience	377
15.1 Neuroscience Basics	378
15.2 Reward Signals, Reinforcement Signals, Values, and Prediction Errors	380
15.3 The Reward Prediction Error Hypothesis	381
15.4 Dopamine	383
15.5 Experimental Support for the Reward Prediction Error Hypothesis	387
15.6 TD Error/Dopamine Correspondence	390
15.7 Neural Actor–Critic	395
15.8 Actor and Critic Learning Rules	398
15.9 Hedonistic Neurons	402
15.10 Collective Reinforcement Learning	404
15.11 Model-based Methods in the Brain	407
15.12 Addiction	409
15.13 Summary	410

16 Applications and Case Studies	421
16.1 TD-Gammon	421
16.2 Samuel’s Checkers Player	426
16.3 Watson’s Daily-Double Wagering	429
16.4 Optimizing Memory Control	432
16.5 Human-level Video Game Play	436
16.6 Mastering the Game of Go	441
16.6.1 AlphaGo	444
16.6.2 AlphaGo Zero	447
16.7 Personalized Web Services	450
16.8 Thermal Soaring	453
17 Frontiers	459
17.1 General Value Functions and Auxiliary Tasks	459
17.2 Temporal Abstraction via Options	461
17.3 Observations and State	464
17.4 Designing Reward Signals	469
17.5 Remaining Issues	472
17.6 Reinforcement Learning and the Future of Artificial Intelligence	475
References	481
Index	519

Preface to the Second Edition

The twenty years since the publication of the first edition of this book have seen tremendous progress in artificial intelligence, propelled in large part by advances in machine learning, including advances in reinforcement learning. Although the impressive computational power that became available is responsible for some of these advances, new developments in theory and algorithms have been driving forces as well. In the face of this progress, a second edition of our 1998 book was long overdue, and we finally began the project in 2012. Our goal for the second edition was the same as our goal for the first: to provide a clear and simple account of the key ideas and algorithms of reinforcement learning that is accessible to readers in all the related disciplines. The edition remains an introduction, and we retain a focus on core, online learning algorithms. This edition includes some new topics that rose to importance over the intervening years, and we expanded coverage of topics that we now understand better. But we made no attempt to provide comprehensive coverage of the field, which has exploded in many different directions. We apologize for having to leave out all but a handful of these contributions.

As in the first edition, we chose not to produce a rigorous formal treatment of reinforcement learning, or to formulate it in the most general terms. However, our deeper understanding of some topics since the first edition required a bit more mathematics to explain; we have set off the more mathematical parts in shaded boxes that the non-mathematically-inclined may choose to skip. We also use a slightly different notation than was used in the first edition. In teaching, we have found that the new notation helps to address some common points of confusion. It emphasizes the difference between random variables, denoted with capital letters, and their instantiations, denoted in lower case. For example, the state, action, and reward at time step t are denoted S_t , A_t , and R_t , while their possible values might be denoted s , a , and r . Along with this, it is natural to use lower case for value functions (e.g., v_π) and restrict capitals to their tabular estimates (e.g., $Q_t(s, a)$). Approximate value functions are deterministic functions of random parameters and are thus also in lower case (e.g., $\hat{v}(s, \mathbf{w}_t) \approx v_\pi(s)$). Vectors, such as the weight vector \mathbf{w}_t (formerly $\boldsymbol{\theta}_t$) and the feature vector \mathbf{x}_t (formerly ϕ_t), are bold and written in lowercase even if they are random variables. Uppercase bold is reserved for matrices. In the first edition we used special notations, $\mathcal{P}_{ss'}^a$ and $\mathcal{R}_{ss'}^a$, for the transition probabilities and expected rewards. One weakness of that notation is that it still did not fully characterize the dynamics of the rewards, giving only their expectations, which is sufficient for dynamic programming but not for reinforcement learning. Another weakness

is the excess of subscripts and superscripts. In this edition we use the explicit notation of $p(s', r|s, a)$ for the joint probability for the next state and reward given the current state and action. All the changes in notation are summarized in a table on page xix.

The second edition is significantly expanded, and its top-level organization has been changed. After the introductory first chapter, the second edition is divided into three new parts. The first part (Chapters 2–8) treats as much of reinforcement learning as possible without going beyond the tabular case for which exact solutions can be found. We cover both learning and planning methods for the tabular case, as well as their unification in n -step methods and in Dyna. Many algorithms presented in this part are new to the second edition, including UCB, Expected Sarsa, Double learning, tree-backup, $Q(\sigma)$, RTDP, and MCTS. Doing the tabular case first, and thoroughly, enables core ideas to be developed in the simplest possible setting. The second part of the book (Chapters 9–13) is then devoted to extending the ideas to function approximation. It has new sections on artificial neural networks, the fourier basis, LSTD, kernel-based methods, Gradient-TD and Emphatic-TD methods, average-reward methods, true online TD(λ), and policy-gradient methods. The second edition significantly expands the treatment of off-policy learning, first for the tabular case in Chapters 5–7, then with function approximation in Chapters 11 and 12. Another change is that the second edition separates the forward-view idea of n -step bootstrapping (now treated more fully in Chapter 7) from the backward-view idea of eligibility traces (now treated independently in Chapter 12). The third part of the book has large new chapters on reinforcement learning’s relationships to psychology (Chapter 14) and neuroscience (Chapter 15), as well as an updated case-studies chapter including Atari game playing, Watson’s wagering strategy, and the Go playing programs AlphaGo and AlphaGo Zero (Chapter 16). Still, out of necessity we have included only a small subset of all that has been done in the field. Our choices reflect our long-standing interests in inexpensive model-free methods that should scale well to large applications. The final chapter now includes a discussion of the future societal impacts of reinforcement learning. For better or worse, the second edition is about twice as large as the first.

This book is designed to be used as the primary text for a one- or two-semester course on reinforcement learning. For a one-semester course, the first ten chapters should be covered in order and form a good core, to which can be added material from the other chapters, from other books such as Bertsekas and Tsitsiklis (1996), Wiering and van Otterlo (2012), and Szepesvari (2010), or from the literature, according to taste. Depending of the students’ background, some additional material on online supervised learning may be helpful. The ideas of options and option models are a natural addition (Sutton, Precup and Singh, 1999). A two-semester course can cover all the chapters as well as supplementary material. The book can also be used as part of broader courses on machine learning, artificial intelligence, or neural networks. In this case, it may be desirable to cover only a subset of the material. We recommend covering Chapter 1 for a brief overview, Chapter 2 through Section 2.4, Chapter 3, and then selecting sections from the remaining chapters according to time and interests. Chapter 6 is the most important for the subject and for the rest of the book. A course focusing on machine learning or neural networks should cover Chapters 9 and 10, and a course focusing on artificial intelligence or planning should cover Chapter 8. Throughout the book, sections and chapters that are more difficult and not essential to the rest of the book are marked

with a *. These can be omitted on first reading without creating problems later on. Some exercises are also marked with a * to indicate that they are more advanced and not essential to understanding the basic material of the chapter.

Most chapters end with a section entitled “Bibliographical and Historical Remarks,” wherein we credit the sources of the ideas presented in that chapter, provide pointers to further reading and ongoing research, and describe relevant historical background. Despite our attempts to make these sections authoritative and complete, we have undoubtedly left out some important prior work. For that we again apologize, and we welcome corrections and extensions for incorporation into the electronic version of the book.

Like the first edition, this edition of the book is dedicated to the memory of A. Harry Klopff. It was Harry who introduced us to each other, and it was his ideas about the brain and artificial intelligence that launched our long excursion into reinforcement learning. Trained in neurophysiology and long interested in machine intelligence, Harry was a senior scientist affiliated with the Avionics Directorate of the Air Force Office of Scientific Research (AFOSR) at Wright-Patterson Air Force Base, Ohio. He was dissatisfied with the great importance attributed to equilibrium-seeking processes, including homeostasis and error-correcting pattern classification methods, in explaining natural intelligence and in providing a basis for machine intelligence. He noted that systems that try to maximize something (whatever that might be) are qualitatively different from equilibrium-seeking systems, and he argued that maximizing systems hold the key to understanding important aspects of natural intelligence and for building artificial intelligences. Harry was instrumental in obtaining funding from AFOSR for a project to assess the scientific merit of these and related ideas. This project was conducted in the late 1970s at the University of Massachusetts Amherst (UMass Amherst), initially under the direction of Michael Arbib, William Kilmer, and Nico Spinelli, professors in the Department of Computer and Information Science at UMass Amherst, and founding members of the Cybernetics Center for Systems Neuroscience at the University, a farsighted group focusing on the intersection of neuroscience and artificial intelligence. Barto, a recent PhD from the University of Michigan, was hired as post doctoral researcher on the project. Meanwhile, Sutton, an undergraduate studying computer science and psychology at Stanford, had been corresponding with Harry regarding their mutual interest in the role of stimulus timing in classical conditioning. Harry suggested to the UMass group that Sutton would be a great addition to the project. Thus, Sutton became a UMass graduate student, whose PhD was directed by Barto, who had become an Associate Professor. The study of reinforcement learning as presented in this book is rightfully an outcome of that project instigated by Harry and inspired by his ideas. Further, Harry was responsible for bringing us, the authors, together in what has been a long and enjoyable interaction. By dedicating this book to Harry we honor his essential contributions, not only to the field of reinforcement learning, but also to our collaboration. We also thank Professors Arbib, Kilmer, and Spinelli for the opportunity they provided to us to begin exploring these ideas. Finally, we thank AFOSR for generous support over the early years of our research, and the NSF for its generous support over many of the following years.

We have very many people to thank for their inspiration and help with this second edition. Everyone we acknowledged for their inspiration and help with the first edition

deserve our deepest gratitude for this edition as well, which would not exist were it not for their contributions to edition number one. To that long list we must add many others who contributed specifically to the second edition. Our students over the many years that we have taught this material contributed in countless ways: exposing errors, offering fixes, and—not the least—being confused in places where we could have explained things better. We especially thank Martha Steenstrup for reading and providing detailed comments throughout. The chapters on psychology and neuroscience could not have been written without the help of many experts in those fields. We thank John Moore for his patient tutoring over many years on animal learning experiments, theory, and neuroscience, and for his careful reading of multiple drafts of Chapters 14 and 15. We also thank Matt Botvinick, Nathaniel Daw, Peter Dayan, and Yael Niv for their penetrating comments on drafts of these chapters, their essential guidance through the massive literature, and their interception of many of our errors in early drafts. Of course, the remaining errors in these chapters—and there must still be some—are totally our own. We thank Phil Thomas for helping us make these chapters accessible to non-psychologists and non-neuroscientists, and we thank Peter Sterling for helping us improve the exposition. We are grateful to Jim Houk for introducing us to the subject of information processing in the basal ganglia and for alerting us to other relevant aspects of neuroscience. José Martínez, Terry Sejnowski, David Silver, Gerry Tesauro, Georgios Theocharous, and Phil Thomas generously helped us understand details of their reinforcement learning applications for inclusion in the case-studies chapter, and they provided helpful comments on drafts of these sections. Special thanks are owed to David Silver for helping us better understand Monte Carlo Tree Search and the DeepMind Go-playing programs. We thank George Konidaris for his help with the section on the Fourier basis. Emilio Cartoni, Thomas Cederborg, Stefan Dernbach, Clemens Rosenbaum, Patrick Taylor, Thomas Colin, and Pierre-Luc Bacon helped us in a number important ways for which we are most grateful.

Sutton would also like to thank the members of the Reinforcement Learning and Artificial Intelligence laboratory at the University of Alberta for contributions to the second edition. He owes a particular debt to Rupam Mahmood for essential contributions to the treatment of off-policy Monte Carlo methods in Chapter 5, to Hamid Maei for helping develop the perspective on off-policy learning presented in Chapter 11, to Eric Graves for conducting the experiments in Chapter 13, to Shangtong Zhang for replicating and thus verifying almost all the experimental results, to Kris De Asis for improving the new technical content of Chapters 7 and 12, and to Harm van Seijen for insights that led to the separation of n -step methods from eligibility traces and (along with Hado van Hasselt) for the ideas involving exact equivalence of forward and backward views of eligibility traces presented in Chapter 12. Sutton also gratefully acknowledges the support and freedom he was granted by the Government of Alberta and the National Science and Engineering Research Council of Canada throughout the period during which the second edition was conceived and written. In particular, he would like to thank Randy Goebel for creating a supportive and far-sighted environment for research in Alberta. He would also like to thank DeepMind their support in the last six months of writing the book.

Finally, we owe thanks to the many careful readers of drafts of the second edition that we posted on the internet. They found many errors that we had missed and alerted us to potential points of confusion.

Preface to the First Edition

We first came to focus on what is now known as reinforcement learning in late 1979. We were both at the University of Massachusetts, working on one of the earliest projects to revive the idea that networks of neuronlike adaptive elements might prove to be a promising approach to artificial adaptive intelligence. The project explored the “heterostatic theory of adaptive systems” developed by A. Harry Klopf. Harry’s work was a rich source of ideas, and we were permitted to explore them critically and compare them with the long history of prior work in adaptive systems. Our task became one of teasing the ideas apart and understanding their relationships and relative importance. This continues today, but in 1979 we came to realize that perhaps the simplest of the ideas, which had long been taken for granted, had received surprisingly little attention from a computational perspective. This was simply the idea of a learning system that *wants* something, that adapts its behavior in order to maximize a special signal from its environment. This was the idea of a “hedonistic” learning system, or, as we would say now, the idea of reinforcement learning.

Like others, we had a sense that reinforcement learning had been thoroughly explored in the early days of cybernetics and artificial intelligence. On closer inspection, though, we found that it had been explored only slightly. While reinforcement learning had clearly motivated some of the earliest computational studies of learning, most of these researchers had gone on to other things, such as pattern classification, supervised learning, and adaptive control, or they had abandoned the study of learning altogether. As a result, the special issues involved in learning how to get something from the environment received relatively little attention. In retrospect, focusing on this idea was the critical step that set this branch of research in motion. Little progress could be made in the computational study of reinforcement learning until it was recognized that such a fundamental idea had not yet been thoroughly explored.

The field has come a long way since then, evolving and maturing in several directions. Reinforcement learning has gradually become one of the most active research areas in machine learning, artificial intelligence, and neural network research. The field has developed strong mathematical foundations and impressive applications. The computational study of reinforcement learning is now a large field, with hundreds of active researchers around the world in diverse disciplines such as psychology, control theory, artificial intelligence, and neuroscience. Particularly important have been the contributions establishing and developing the relationships to the theory of optimal control and dynamic programming.

The overall problem of learning from interaction to achieve goals is still far from being solved, but our understanding of it has improved significantly. We can now place component ideas, such as temporal-difference learning, dynamic programming, and function approximation, within a coherent perspective with respect to the overall problem.

Our goal in writing this book was to provide a clear and simple account of the key ideas and algorithms of reinforcement learning. We wanted our treatment to be accessible to readers in all of the related disciplines, but we could not cover all of these perspectives in detail. For the most part, our treatment takes the point of view of artificial intelligence and engineering. Coverage of connections to other fields we leave to others or to another time. We also chose not to produce a rigorous formal treatment of reinforcement learning. We did not reach for the highest possible level of mathematical abstraction and did not rely on a theorem–proof format. We tried to choose a level of mathematical detail that points the mathematically inclined in the right directions without distracting from the simplicity and potential generality of the underlying ideas.

In some sense we have been working toward this book for thirty years, and we have lots of people to thank. First, we thank those who have personally helped us develop the overall view presented in this book: Harry Klopf, for helping us recognize that reinforcement learning needed to be revived; Chris Watkins, Dimitri Bertsekas, John Tsitsiklis, and Paul Werbos, for helping us see the value of the relationships to dynamic programming; John Moore and Jim Kehoe, for insights and inspirations from animal learning theory; Oliver Selfridge, for emphasizing the breadth and importance of adaptation; and, more generally, our colleagues and students who have contributed in countless ways: Ron Williams, Charles Anderson, Satinder Singh, Sridhar Mahadevan, Steve Bradtke, Bob Crites, Peter Dayan, and Leemon Baird. Our view of reinforcement learning has been significantly enriched by discussions with Paul Cohen, Paul Utgoff, Martha Steenstrup, Gerry Tesauro, Mike Jordan, Leslie Kaelbling, Andrew Moore, Chris Atkeson, Tom Mitchell, Nils Nilsson, Stuart Russell, Tom Dietterich, Tom Dean, and Bob Narendra. We thank Michael Littman, Gerry Tesauro, Bob Crites, Satinder Singh, and Wei Zhang for providing specifics of Sections 4.7, 15.1, 15.4, 15.5, and 15.6 respectively. We thank the Air Force Office of Scientific Research, the National Science Foundation, and GTE Laboratories for their long and farsighted support.

We also wish to thank the many people who have read drafts of this book and provided valuable comments, including Tom Kalt, John Tsitsiklis, Paweł Cichosz, Olle Gällmo, Chuck Anderson, Stuart Russell, Ben Van Roy, Paul Steenstrup, Paul Cohen, Sridhar Mahadevan, Jette Randlov, Brian Sheppard, Thomas O’Connell, Richard Coggins, Cristina Versino, John H. Hiett, Andreas Badelt, Jay Ponte, Joe Beck, Justus Piater, Martha Steenstrup, Satinder Singh, Tommi Jaakkola, Dimitri Bertsekas, Torbjörn Ekman, Christina Björkman, Jakob Carlström, and Olle Palmgren. Finally, we thank Gwyn Mitchell for helping in many ways, and Harry Stanton and Bob Prior for being our champions at MIT Press.

Summary of Notation

Capital letters are used for random variables, whereas lower case letters are used for the values of random variables and for scalar functions. Quantities that are required to be real-valued vectors are written in bold and in lower case (even if random variables). Matrices are bold capitals.

\doteq	equality relationship that is true by definition
\approx	approximately equal
\propto	proportional to
$\Pr\{X=x\}$	probability that a random variable X takes on the value x
$X \sim p$	random variable X selected from distribution $p(x) \doteq \Pr\{X=x\}$
$\mathbb{E}[X]$	expectation of a random variable X , i.e., $\mathbb{E}[X] \doteq \sum_x p(x)x$
$\arg \max_a f(a)$	a value of a at which $f(a)$ takes its maximal value
$\ln x$	natural logarithm of x
$e^x, \exp(x)$	the base of the natural logarithm, $e \approx 2.71828$, carried to power x ; $e^{\ln x} = x$
\mathbb{R}	set of real numbers
$f : \mathcal{X} \rightarrow \mathcal{Y}$	function f from elements of set \mathcal{X} to elements of set \mathcal{Y}
\leftarrow	assignment
$(a, b]$	the real interval between a and b including b but not including a
ε	probability of taking a random action in an ε -greedy policy
α, β	step-size parameters
γ	discount-rate parameter
λ	decay-rate parameter for eligibility traces
$\mathbb{1}_{\textit{predicate}}$	indicator function ($\mathbb{1}_{\textit{predicate}} \doteq 1$ if the <i>predicate</i> is true, else 0)

In a multi-arm bandit problem:

k	number of actions (arms)
t	discrete time step or play number
$q_*(a)$	true value (expected reward) of action a
$Q_t(a)$	estimate at time t of $q_*(a)$
$N_t(a)$	number of times action a has been selected up prior to time t
$H_t(a)$	learned preference for selecting action a at time t
$\pi_t(a)$	probability of selecting action a at time t
\bar{R}_t	estimate at time t of the expected reward given π_t

In a Markov Decision Process:

s, s'	states
a	an action
r	a reward
\mathcal{S}	set of all nonterminal states
\mathcal{S}^+	set of all states, including the terminal state
$\mathcal{A}(s)$	set of all actions available in state s
\mathcal{R}	set of all possible rewards, a finite subset of \mathbb{R}
\subset	subset of (e.g., $\mathcal{R} \subset \mathbb{R}$)
\in	is an element of; e.g. ($s \in \mathcal{S}, r \in \mathcal{R}$)
$ \mathcal{S} $	number of elements in set \mathcal{S}
t	discrete time step
$T, T(t)$	final time step of an episode, or of the episode including time step t
A_t	action at time t
S_t	state at time t , typically due, stochastically, to S_{t-1} and A_{t-1}
R_t	reward at time t , typically due, stochastically, to S_{t-1} and A_{t-1}
π	policy (decision-making rule)
$\pi(s)$	action taken in state s under <i>deterministic</i> policy π
$\pi(a s)$	probability of taking action a in state s under <i>stochastic</i> policy π
G_t	return following time t
h	horizon, the time step one looks up to in a forward view
$G_{t:t+n}, G_{t:h}$	n -step return from $t + 1$ to $t + n$, or to h (discounted and corrected)
$\tilde{G}_{t:h}$	flat return (undiscounted and uncorrected) from $t + 1$ to h (Section 5.8)
G_t^λ	λ -return (Section 12.1)
$G_{t:h}^\lambda$	truncated, corrected λ -return (Section 12.3)
$G_t^{\lambda_s}, G_t^{\lambda_a}$	λ -return, corrected by estimated state, or action, values (Section 12.8)
$p(s', r s, a)$	probability of transition to state s' with reward r , from state s and action a
$p(s' s, a)$	probability of transition to state s' , from state s taking action a
$r(s, a)$	expected immediate reward from state s after action a
$r(s, a, s')$	expected immediate reward on transition from s to s' under action a
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy
$q_\pi(s, a)$	value of taking action a in state s under policy π
$q_*(s, a)$	value of taking action a in state s under the optimal policy
V, V_t	array estimates of state-value function v_π or v_*
Q, Q_t	array estimates of action-value function q_π or q_*
$\bar{V}_t(s)$	expected approximate action value; for example, $\bar{V}_t(s) \doteq \sum_a \pi(a s)Q_t(s, a)$
U_t	target for estimate at time t

δ_t	temporal-difference (TD) error at t (a random variable) (Section 6.1)
δ_t^s, δ_t^a	state- and action-specific forms of the TD error (Section 12.9)
n	in n -step methods, n is the number of steps of bootstrapping
d	dimensionality—the number of components of \mathbf{w}
d'	alternate dimensionality—the number of components of $\boldsymbol{\theta}$
\mathbf{w}, \mathbf{w}_t	d -vector of weights underlying an approximate value function
$w_i, w_{t,i}$	i th component of learnable weight vector
$\hat{v}(s, \mathbf{w})$	approximate value of state s given weight vector \mathbf{w}
$v_{\mathbf{w}}(s)$	alternate notation for $\hat{v}(s, \mathbf{w})$
$\hat{q}(s, a, \mathbf{w})$	approximate value of state-action pair s, a given weight vector \mathbf{w}
$\nabla \hat{v}(s, \mathbf{w})$	column vector of partial derivatives of $\hat{v}(s, \mathbf{w})$ with respect to \mathbf{w}
$\nabla \hat{q}(s, a, \mathbf{w})$	column vector of partial derivatives of $\hat{q}(s, a, \mathbf{w})$ with respect to \mathbf{w}
$\mathbf{x}(s)$	vector of features visible when in state s
$\mathbf{x}(s, a)$	vector of features visible when in state s taking action a
$x_i(s), x_i(s, a)$	i th component of vector $\mathbf{x}(s)$ or $\mathbf{x}(s, a)$
\mathbf{x}_t	shorthand for $\mathbf{x}(S_t)$ or $\mathbf{x}(S_t, A_t)$
$\mathbf{w}^\top \mathbf{x}$	inner product of vectors, $\mathbf{w}^\top \mathbf{x} \doteq \sum_i w_i x_i$; for example, $\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s)$
\mathbf{v}, \mathbf{v}_t	secondary d -vector of weights, used to learn \mathbf{w} (Chapter 11)
\mathbf{z}_t	d -vector of eligibility traces at time t (Chapter 12)
$\boldsymbol{\theta}, \boldsymbol{\theta}_t$	parameter vector of target policy (Chapter 13)
$\pi(a s, \boldsymbol{\theta})$	probability of taking action a in state s given parameter vector $\boldsymbol{\theta}$
$\pi_{\boldsymbol{\theta}}$	policy corresponding to parameter $\boldsymbol{\theta}$
$\nabla \pi(a s, \boldsymbol{\theta})$	column vector of partial derivatives of $\pi(a s, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$
$J(\boldsymbol{\theta})$	performance measure for the policy $\pi_{\boldsymbol{\theta}}$
$\nabla J(\boldsymbol{\theta})$	column vector of partial derivatives of $J(\boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$
$h(s, a, \boldsymbol{\theta})$	preference for selecting action a in state s based on $\boldsymbol{\theta}$
$b(a s)$	behavior policy used to select actions while learning about target policy π
$b(s)$	a baseline function $b : \mathcal{S} \mapsto \mathbb{R}$ for policy-gradient methods
b	branching factor for an MDP or search tree
$\rho_{t:h}$	importance sampling ratio for time t through time h (Section 5.5)
ρ_t	importance sampling ratio for time t alone, $\rho_t \doteq \rho_{t:t}$
$r(\pi)$	average reward (reward rate) for policy π (Section 10.3)
\bar{R}_t	estimate of $r(\pi)$ at time t
$\mu(s)$	on-policy distribution over states (Section 9.2)
$\boldsymbol{\mu}$	$ \mathcal{S} $ -vector of the $\mu(s)$ for all $s \in \mathcal{S}$
$\ v\ _{\mu}^2$	μ -weighted squared norm of value function v , i.e., $\ v\ _{\mu}^2 \doteq \sum_{s \in \mathcal{S}} \mu(s) v(s)^2$
$\eta(s)$	expected number of visits to state s per episode (page 199)
Π	projection operator for value functions (page 268)
B_{π}	Bellman operator for value functions (Section 11.4)

A	$d \times d$ matrix $\mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]$
b	d -dimensional vector $\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t]$
w_{TD}	TD fixed point $\mathbf{w}_{\text{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}$ (a d -vector, Section 9.4)
I	identity matrix
P	$ \mathcal{S} \times \mathcal{S} $ matrix of state-transition probabilities under π
D	$ \mathcal{S} \times \mathcal{S} $ diagonal matrix with μ on its diagonal
X	$ \mathcal{S} \times d$ matrix with the $\mathbf{x}(s)$ as its rows
$\bar{\delta}_{\mathbf{w}}(s)$	Bellman error (expected TD error) for $v_{\mathbf{w}}$ at state s (Section 11.4)
$\bar{\delta}_{\mathbf{w}}$, BE	Bellman error vector, with components $\bar{\delta}_{\mathbf{w}}(s)$
$\overline{\text{VE}}(\mathbf{w})$	mean square value error $\overline{\text{VE}}(\mathbf{w}) \doteq \ v_{\mathbf{w}} - v_{\pi}\ _{\mu}^2$ (Section 9.2)
$\overline{\text{BE}}(\mathbf{w})$	mean square Bellman error $\overline{\text{BE}}(\mathbf{w}) \doteq \ \bar{\delta}_{\mathbf{w}}\ _{\mu}^2$
$\overline{\text{PBE}}(\mathbf{w})$	mean square projected Bellman error $\overline{\text{PBE}}(\mathbf{w}) \doteq \ \Pi \bar{\delta}_{\mathbf{w}}\ _{\mu}^2$
$\overline{\text{TDE}}(\mathbf{w})$	mean square temporal-difference error $\overline{\text{TDE}}(\mathbf{w}) \doteq \mathbb{E}_b[\rho_t \delta_t^2]$ (Section 11.5)
$\overline{\text{RE}}(\mathbf{w})$	mean square return error (Section 11.6)

Chapter 1

Introduction

The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

In this book we explore a *computational* approach to learning from interaction. Rather than directly theorizing about how people or animals learn, we primarily explore idealized learning situations and evaluate the effectiveness of various learning methods. That is, we adopt the perspective of an artificial intelligence researcher or engineer. We explore designs for machines that are effective in solving learning problems of scientific or economic interest, evaluating the designs through mathematical analysis or computational experiments. The approach we explore, called *reinforcement learning*, is much more focused on goal-directed learning from interaction than are other approaches to machine learning.

1.1 Reinforcement Learning

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning.

Reinforcement learning, like many topics whose names end with “ing,” such as machine learning and mountaineering, is simultaneously a problem, a class of solution methods that work well on the problem, and the field that studies this problem and its solution methods. It is convenient to use a single name for all three things, but at the same time essential to keep the three conceptually separate. In particular, the distinction between problems and solution methods is very important in reinforcement learning; failing to make this distinction is the source of many confusions.

We formalize the problem of reinforcement learning using ideas from dynamical systems theory, specifically, as the optimal control of incompletely-known Markov decision processes. The details of this formalization must wait until Chapter 3, but the basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting over time with its environment to achieve a goal. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. Markov decision processes are intended to include just these three aspects—sensation, action, and goal—in their simplest possible forms without trivializing any of them. Any method that is well suited to solving such problems we consider to be a reinforcement learning method.

Reinforcement learning is different from *supervised learning*, the kind of learning studied in most current research in the field of machine learning. Supervised learning is learning from a training set of labeled examples provided by a knowledgeable external supervisor. Each example is a description of a situation together with a specification—the label—of the correct action the system should take in that situation, which is often to identify a category to which the situation belongs. The object of this kind of learning is for the system to extrapolate, or generalize, its responses so that it acts correctly in situations not present in the training set. This is an important kind of learning, but alone it is not adequate for learning from interaction. In interactive problems it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent has to act. In uncharted territory—where one would expect learning to be most beneficial—an agent must be able to learn from its own experience.

Reinforcement learning is also different from what machine learning researchers call *unsupervised learning*, which is typically about finding structure hidden in collections of unlabeled data. The terms supervised learning and unsupervised learning would seem to exhaustively classify machine learning paradigms, but they do not. Although one might be tempted to think of reinforcement learning as a kind of unsupervised learning because it does not rely on examples of correct behavior, reinforcement learning is trying to maximize a reward signal instead of trying to find hidden structure. Uncovering structure in an agent’s experience can certainly be useful in reinforcement learning, but by itself does not address the reinforcement learning problem of maximizing a reward signal. We therefore consider reinforcement learning to be a third machine learning paradigm, alongside supervised learning and unsupervised learning and perhaps other paradigms.

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to *exploit* what it has already experienced in order to obtain reward, but it also has to *explore* in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions *and* progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate of its expected reward. The exploration–exploitation dilemma has been intensively studied by mathematicians for many decades, yet remains unresolved. For now, we simply note that the entire issue of balancing exploration and exploitation does not even arise in supervised and unsupervised learning, at least in the purest forms of these paradigms.

Another key feature of reinforcement learning is that it explicitly considers the *whole* problem of a goal-directed agent interacting with an uncertain environment. This is in contrast to many approaches that consider subproblems without addressing how they might fit into a larger picture. For example, we have mentioned that many machine learning researchers have studied supervised learning without specifying how such an ability would ultimately be useful. Other researchers have developed theories of planning with general goals, but without considering planning’s role in real-time decision making, or the question of where the predictive models necessary for planning would come from. Although these approaches have yielded many useful results, their focus on isolated subproblems is a significant limitation.

Reinforcement learning takes the opposite tack, starting with a complete, interactive, goal-seeking agent. All reinforcement learning agents have explicit goals, can sense aspects of their environments, and can choose actions to influence their environments. Moreover, it is usually assumed from the beginning that the agent has to operate despite significant uncertainty about the environment it faces. When reinforcement learning involves planning, it has to address the interplay between planning and real-time action selection, as well as the question of how environment models are acquired and improved. When reinforcement learning involves supervised learning, it does so for specific reasons that determine which capabilities are critical and which are not. For learning research to make progress, important subproblems have to be isolated and studied, but they should be subproblems that play clear roles in complete, interactive, goal-seeking agents, even if all the details of the complete agent cannot yet be filled in.

By a complete, interactive, goal-seeking agent we do not always mean something like a complete organism or robot. These are clearly examples, but a complete, interactive, goal-seeking agent can also be a component of a larger behaving system. In this case, the agent directly interacts with the rest of the larger system and indirectly interacts with the larger system’s environment. A simple example is an agent that monitors the charge level of robot’s battery and sends commands to the robot’s control architecture. This agent’s environment is the rest of the robot together with the robot’s environment. It is

important to look beyond the most obvious examples of agents and their environments to appreciate the generality of the reinforcement learning framework.

One of the most exciting aspects of modern reinforcement learning is its substantive and fruitful interactions with other engineering and scientific disciplines. Reinforcement learning is part of a decades-long trend within artificial intelligence and machine learning toward greater integration with statistics, optimization, and other mathematical subjects. For example, the ability of some reinforcement learning methods to learn with parameterized approximators addresses the classical “curse of dimensionality” in operations research and control theory. More distinctively, reinforcement learning has also interacted strongly with psychology and neuroscience, with substantial benefits going both ways. Of all the forms of machine learning, reinforcement learning is the closest to the kind of learning that humans and other animals do, and many of the core algorithms of reinforcement learning were originally inspired by biological learning systems. Reinforcement learning has also given back, both through a psychological model of animal learning that better matches some of the empirical data, and through an influential model of parts of the brain’s reward system. The body of this book develops the ideas of reinforcement learning that pertain to engineering and artificial intelligence, with connections to psychology and neuroscience summarized in Chapters 14 and 15.

Finally, reinforcement learning is also part of a larger trend in artificial intelligence back toward simple general principles. Since the late 1960s, many artificial intelligence researchers presumed that there are no general principles to be discovered, that intelligence is instead due to the possession of a vast number of special purpose tricks, procedures, and heuristics. It was sometimes said that if we could just get enough relevant facts into a machine, say one million, or one billion, then it would become intelligent. Methods based on general principles, such as search or learning, were characterized as “weak methods,” whereas those based on specific knowledge were called “strong methods.” This view is uncommon today. From our point of view, it was premature: too little effort had been put into the search for general principles to conclude that there were none. Modern artificial intelligence now includes much research looking for general principles of learning, search, and decision making. It is not clear how far back the pendulum will swing, but reinforcement learning research is certainly part of the swing back toward simpler and fewer general principles of artificial intelligence.

1.2 Examples

A good way to understand reinforcement learning is to consider some of the examples and possible applications that have guided its development.

- A master chess player makes a move. The choice is informed both by planning—anticipating possible replies and counterreplies—and by immediate, intuitive judgments of the desirability of particular positions and moves.
- An adaptive controller adjusts parameters of a petroleum refinery’s operation in real time. The controller optimizes the yield/cost/quality trade-off on the basis

of specified marginal costs without sticking strictly to the set points originally suggested by engineers.

- A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 20 miles per hour.
- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station. It makes its decision based on the current charge level of its battery and how quickly and easily it has been able to find the recharger in the past.
- Phil prepares his breakfast. Closely examined, even this apparently mundane activity reveals a complex web of conditional behavior and interlocking goal–subgoal relationships: walking to the cupboard, opening it, selecting a cereal box, then reaching for, grasping, and retrieving the box. Other complex, tuned, interactive sequences of behavior are required to obtain a bowl, spoon, and milk carton. Each step involves a series of eye movements to obtain information and to guide reaching and locomotion. Rapid judgments are continually made about how to carry the objects or whether it is better to ferry some of them to the dining table before obtaining others. Each step is guided by goals, such as grasping a spoon or getting to the refrigerator, and is in service of other goals, such as having the spoon to eat with once the cereal is prepared and ultimately obtaining nourishment. Whether he is aware of it or not, Phil is accessing information about the state of his body that determines his nutritional needs, level of hunger, and food preferences.

These examples share features that are so basic that they are easy to overlook. All involve *interaction* between an active decision-making agent and its environment, within which the agent seeks to achieve a *goal* despite *uncertainty* about its environment. The agent’s actions are permitted to affect the future state of the environment (e.g., the next chess position, the level of reservoirs of the refinery, the robot’s next location and the future charge level of its battery), thereby affecting the actions and opportunities available to the agent at later times. Correct choice requires taking into account indirect, delayed consequences of actions, and thus may require foresight or planning.

At the same time, in all of these examples the effects of actions cannot be fully predicted; thus the agent must monitor its environment frequently and react appropriately. For example, Phil must watch the milk he pours into his cereal bowl to keep it from overflowing. All these examples involve goals that are explicit in the sense that the agent can judge progress toward its goal based on what it can sense directly. The chess player knows whether or not he wins, the refinery controller knows how much petroleum is being produced, the gazelle calf knows when it falls, the mobile robot knows when its batteries run down, and Phil knows whether or not he is enjoying his breakfast.

In all of these examples the agent can use its experience to improve its performance over time. The chess player refines the intuition he uses to evaluate positions, thereby improving his play; the gazelle calf improves the efficiency with which it can run; Phil learns to streamline making his breakfast. The knowledge the agent brings to the task at the start—either from previous experience with related tasks or built into it by design or

evolution— influences what is useful or easy to learn, but interaction with the environment is essential for adjusting behavior to exploit specific features of the task.

1.3 Elements of Reinforcement Learning

Beyond the agent and the environment, one can identify four main subelements of a reinforcement learning system: a *policy*, a *reward signal*, a *value function*, and, optionally, a *model* of the environment.

A *policy* defines the learning agent’s way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. It corresponds to what in psychology would be called a set of stimulus–response rules or associations. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic, specifying probabilities for each action.

A *reward signal* defines the goal of a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the *reward*. The agent’s sole objective is to maximize the total reward it receives over the long run. The reward signal thus defines what are the good and bad events for the agent. In a biological system, we might think of rewards as analogous to the experiences of pleasure or pain. They are the immediate and defining features of the problem faced by the agent. The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. In general, reward signals may be stochastic functions of the state of the environment and the actions taken.

Whereas the reward signal indicates what is good in an immediate sense, a *value function* specifies what is good in the long run. Roughly speaking, the *value* of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the *long-term* desirability of states after taking into account the states that are likely to follow and the rewards available in those states. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards. Or the reverse could be true. To make a human analogy, rewards are somewhat like pleasure (if high) and pain (if low), whereas values correspond to a more refined and farsighted judgment of how pleased or displeased we are that our environment is in a particular state.

Rewards are in a sense primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no values, and the only purpose of estimating values is to achieve more reward. Nevertheless, it is values with which we are most concerned when making and evaluating decisions. Action choices are made based on value judgments. We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run. Unfortunately, it is much harder to determine values than it is to determine rewards. Rewards are basically given directly by the environment, but values must be estimated and re-estimated from

the sequences of observations an agent makes over its entire lifetime. In fact, the most important component of almost all reinforcement learning algorithms we consider is a method for efficiently estimating values. The central role of value estimation is arguably the most important thing that has been learned about reinforcement learning over the last six decades.

The fourth and final element of some reinforcement learning systems is a *model* of the environment. This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for *planning*, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called *model-based* methods, as opposed to simpler *model-free* methods that are explicitly trial-and-error learners—viewed as almost the *opposite* of planning. In Chapter 8 we explore reinforcement learning systems that simultaneously learn by trial and error, learn a model of the environment, and use the model for planning. Modern reinforcement learning spans the spectrum from low-level, trial-and-error learning to high-level, deliberative planning.

1.4 Limitations and Scope

Reinforcement learning relies heavily on the concept of state—as input to the policy and value function, and as both input to and output from the model. Informally, we can think of the state as a signal conveying to the agent some sense of “how the environment is” at a particular time. The formal definition of state as we use it here is given by the framework of Markov decision processes presented in Chapter 3. More generally, however, we encourage the reader to follow the informal meaning and think of the state as whatever information is available to the agent about its environment. In effect, we assume that the state signal is produced by some preprocessing system that is nominally part of the agent’s environment. We do not address the issues of constructing, changing, or learning the state signal in this book (other than briefly in Section 17.3). We take this approach not because we consider state representation to be unimportant, but in order to focus fully on the decision-making issues. In other words, our concern in this book is not with designing the state signal, but with deciding what action to take as a function of whatever state signal is available.

Most of the reinforcement learning methods we consider in this book are structured around estimating value functions, but it is not strictly necessary to do this to solve reinforcement learning problems. For example, solution methods such as genetic algorithms, genetic programming, simulated annealing, and other optimization methods never estimate value functions. These methods apply multiple static policies each interacting over an extended period of time with a separate instance of the environment. The policies that obtain the most reward, and random variations of them, are carried over to the next generation of policies, and the process repeats. We call these *evolutionary* methods because their operation is analogous to the way biological evolution produces organisms

with skilled behavior even if they do not learn during their individual lifetimes. If the space of policies is sufficiently small, or can be structured so that good policies are common or easy to find—or if a lot of time is available for the search—then evolutionary methods can be effective. In addition, evolutionary methods have advantages on problems in which the learning agent cannot sense the complete state of its environment.

Our focus is on reinforcement learning methods that learn while interacting with the environment, which evolutionary methods do not do. Methods able to take advantage of the details of individual behavioral interactions can be much more efficient than evolutionary methods in many cases. Evolutionary methods ignore much of the useful structure of the reinforcement learning problem: they do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects. In some cases such information can be misleading (e.g., when states are misperceived), but more often it should enable more efficient search. Although evolution and learning share many features and naturally work together, we do not consider evolutionary methods by themselves to be especially well suited to reinforcement learning problems and, accordingly, we do not cover them in this book.

1.5 An Extended Example: Tic-Tac-Toe

To illustrate the general idea of reinforcement learning and contrast it with other approaches, we next consider a single example in more detail.

Consider the familiar child’s game of tic-tac-toe. Two players take turns playing on a three-by-three board. One player plays Xs and the other Os until one player wins by placing three marks in a row, horizontally, vertically, or diagonally, as the X player has in the game shown to the right. If the board fills up with neither player getting three in a row, then the game is a draw. Because a skilled player can play so as never to lose, let us assume that we are playing against an imperfect player, one whose play is sometimes incorrect and allows us to win. For the moment, in fact, let us consider draws and losses to be equally bad for us. How might we construct a player that will find the imperfections in its opponent’s play and learn to maximize its chances of winning?

X	O	O
O	X	X
		X

Although this is a simple problem, it cannot readily be solved in a satisfactory way through classical techniques. For example, the classical “minimax” solution from game theory is not correct here because it assumes a particular way of playing by the opponent. For example, a minimax player would never reach a game state from which it could lose, even if in fact it always won from that state because of incorrect play by the opponent. Classical optimization methods for sequential decision problems, such as dynamic programming, can *compute* an optimal solution for any opponent, but require as input a complete specification of that opponent, including the probabilities with which the opponent makes each move in each board state. Let us assume that this information is not available *a priori* for this problem, as it is not for the vast majority of problems of

practical interest. On the other hand, such information can be estimated from experience, in this case by playing many games against the opponent. About the best one can do on this problem is first to learn a model of the opponent’s behavior, up to some level of confidence, and then apply dynamic programming to compute an optimal solution given the approximate opponent model. In the end, this is not that different from some of the reinforcement learning methods we examine later in this book.

An evolutionary method applied to this problem would directly search the space of possible policies for one with a high probability of winning against the opponent. Here, a policy is a rule that tells the player what move to make for every state of the game—every possible configuration of Xs and Os on the three-by-three board. For each policy considered, an estimate of its winning probability would be obtained by playing some number of games against the opponent. This evaluation would then direct which policy or policies were considered next. A typical evolutionary method would hill-climb in policy space, successively generating and evaluating policies in an attempt to obtain incremental improvements. Or, perhaps, a genetic-style algorithm could be used that would maintain and evaluate a population of policies. Literally hundreds of different optimization methods could be applied.

Here is how the tic-tac-toe problem would be approached with a method making use of a value function. First we would set up a table of numbers, one for each possible state of the game. Each number will be the latest estimate of the probability of our winning from that state. We treat this estimate as the state’s *value*, and the whole table is the learned value function. State A has higher value than state B, or is considered “better” than state B, if the current estimate of the probability of our winning from A is higher than it is from B. Assuming we always play Xs, then for all states with three Xs in a row the probability of winning is 1, because we have already won. Similarly, for all states with three Os in a row, or that are filled up, the correct probability is 0, as we cannot win from them. We set the initial values of all the other states to 0.5, representing a guess that we have a 50% chance of winning.

We then play many games against the opponent. To select our moves we examine the states that would result from each of our possible moves (one for each blank space on the board) and look up their current values in the table. Most of the time we move *greedily*, selecting the move that leads to the state with greatest value, that is, with the highest estimated probability of winning. Occasionally, however, we select randomly from among the other moves instead. These are called *exploratory* moves because they cause us to experience states that we might otherwise never see. A sequence of moves made and considered during a game can be diagrammed as in Figure 1.1.

While we are playing, we change the values of the states in which we find ourselves during the game. We attempt to make them more accurate estimates of the probabilities of winning. To do this, we “back up” the value of the state after each greedy move to the state before the move, as suggested by the arrows in Figure 1.1. More precisely, the current value of the earlier state is updated to be closer to the value of the later state. This can be done by moving the earlier state’s value a fraction of the way toward the value of the later state. If we let S_t denote the state before the greedy move, and S_{t+1} the state after that move, then the update to the estimated value of S_t , denoted $V(S_t)$,

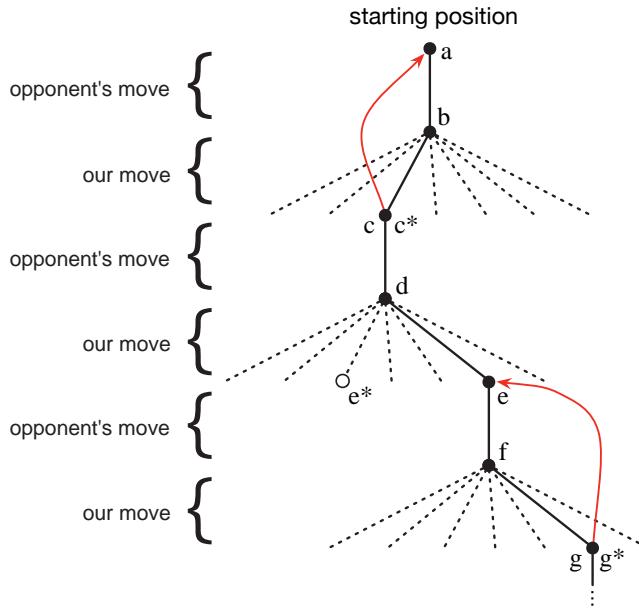


Figure 1.1: A sequence of tic-tac-toe moves. The solid black lines represent the moves taken during a game; the dashed lines represent moves that we (our reinforcement learning player) considered but did not make. The * indicates the move currently estimated to be the best. Our second move was an exploratory move, meaning that it was taken even though another sibling move, the one leading to e^* , was ranked higher. Exploratory moves do not result in any learning, but each of our other moves does, causing updates as suggested by the red arrows in which estimated values are moved up the tree from later nodes to earlier nodes as detailed in the text.

can be written as

$$V(S_t) \leftarrow V(S_t) + \alpha [V(S_{t+1}) - V(S_t)],$$

where α is a small positive fraction called the *step-size parameter*, which influences the rate of learning. This update rule is an example of a *temporal-difference* learning method, so called because its changes are based on a difference, $V(S_{t+1}) - V(S_t)$, between estimates at two successive times.

The method described above performs quite well on this task. For example, if the step-size parameter is reduced properly over time, then this method converges, for any fixed opponent, to the true probabilities of winning from each state given optimal play by our player. Furthermore, the moves then taken (except on exploratory moves) are in fact the optimal moves against this (imperfect) opponent. In other words, the method converges to an optimal policy for playing the game against this opponent. If the step-size parameter is not reduced all the way to zero over time, then this player also plays well against opponents that slowly change their way of playing.

This example illustrates the differences between evolutionary methods and methods that learn value functions. To evaluate a policy, an evolutionary method holds the policy fixed and plays many games against the opponent or simulates many games using a model of the opponent. The frequency of wins gives an unbiased estimate of the probability of winning with that policy, and can be used to direct the next policy selection. But each policy change is made only after many games, and only the final outcome of each game is used: what happens *during* the games is ignored. For example, if the player wins, then *all* of its behavior in the game is given credit, independently of how specific moves might have been critical to the win. Credit is even given to moves that never occurred! Value function methods, in contrast, allow individual states to be evaluated. In the end, evolutionary and value function methods both search the space of policies, but learning a value function takes advantage of information available during the course of play.

This simple example illustrates some of the key features of reinforcement learning methods. First, there is the emphasis on learning while interacting with an environment, in this case with an opponent player. Second, there is a clear goal, and correct behavior requires planning or foresight that takes into account delayed effects of one's choices. For example, the simple reinforcement learning player would learn to set up multi-move traps for a shortsighted opponent. It is a striking feature of the reinforcement learning solution that it can achieve the effects of planning and lookahead without using a model of the opponent and without conducting an explicit search over possible sequences of future states and actions.

While this example illustrates some of the key features of reinforcement learning, it is so simple that it might give the impression that reinforcement learning is more limited than it really is. Although tic-tac-toe is a two-person game, reinforcement learning also applies in the case in which there is no external adversary, that is, in the case of a “game against nature.” Reinforcement learning also is not restricted to problems in which behavior breaks down into separate episodes, like the separate games of tic-tac-toe, with reward only at the end of each episode. It is just as applicable when behavior continues indefinitely and when rewards of various magnitudes can be received at any time. Reinforcement learning is also applicable to problems that do not even break down into discrete time steps like the plays of tic-tac-toe. The general principles apply to continuous-time problems as well, although the theory gets more complicated and we omit it from this introductory treatment.

Tic-tac-toe has a relatively small, finite state set, whereas reinforcement learning can be used when the state set is very large, or even infinite. For example, Gerry Tesauro (1992, 1995) combined the algorithm described above with an artificial neural network to learn to play backgammon, which has approximately 10^{20} states. With this many states it is impossible ever to experience more than a small fraction of them. Tesauro’s program learned to play far better than any previous program and eventually better than the world’s best human players (see Section 16.1). The artificial neural network provides the program with the ability to generalize from its experience, so that in new states it selects moves based on information saved from similar states faced in the past, as determined by the network. How well a reinforcement learning system can work in problems with such large state sets is intimately tied to how appropriately it can generalize from past

experience. It is in this role that we have the greatest need for supervised learning methods within reinforcement learning. Artificial neural networks and deep learning (Section 9.7) are not the only, or necessarily the best, way to do this.

In this tic-tac-toe example, learning started with no prior knowledge beyond the rules of the game, but reinforcement learning by no means entails a tabula rasa view of learning and intelligence. On the contrary, prior information can be incorporated into reinforcement learning in a variety of ways that can be critical for efficient learning (e.g., see Sections 9.5, 17.4, and 13.1). We also have access to the true state in the tic-tac-toe example, whereas reinforcement learning can also be applied when part of the state is hidden, or when different states appear to the learner to be the same.

Finally, the tic-tac-toe player was able to look ahead and know the states that would result from each of its possible moves. To do this, it had to have a model of the game that allowed it to foresee how its environment would change in response to moves that it might never make. Many problems are like this, but in others even a short-term model of the effects of actions is lacking. Reinforcement learning can be applied in either case. A model is not required, but models can easily be used if they are available or can be learned (Chapter 8).

On the other hand, there are reinforcement learning methods that do not need any kind of environment model at all. Model-free systems cannot even think about how their environments will change in response to a single action. The tic-tac-toe player is model-free in this sense with respect to its opponent: it has no model of its opponent of any kind. Because models have to be reasonably accurate to be useful, model-free methods can have advantages over more complex methods when the real bottleneck in solving a problem is the difficulty of constructing a sufficiently accurate environment model. Model-free methods are also important building blocks for model-based methods. In this book we devote several chapters to model-free methods before we discuss how they can be used as components of more complex model-based methods.

Reinforcement learning can be used at both high and low levels in a system. Although the tic-tac-toe player learned only about the basic moves of the game, nothing prevents reinforcement learning from working at higher levels where each of the “actions” may itself be the application of a possibly elaborate problem-solving method. In hierarchical learning systems, reinforcement learning can work simultaneously on several levels.

Exercise 1.1: Self-Play Suppose, instead of playing against a random opponent, the reinforcement learning algorithm described above played against itself, with both sides learning. What do you think would happen in this case? Would it learn a different policy for selecting moves? □

Exercise 1.2: Symmetries Many tic-tac-toe positions appear different but are really the same because of symmetries. How might we amend the learning process described above to take advantage of this? In what ways would this change improve the learning process? Now think again. Suppose the opponent did not take advantage of symmetries. In that case, should we? Is it true, then, that symmetrically equivalent positions should necessarily have the same value? □

Exercise 1.3: Greedy Play Suppose the reinforcement learning player was *greedy*, that is, it always played the move that brought it to the position that it rated the best. Might it

learn to play better, or worse, than a nongreedy player? What problems might occur? \square

Exercise 1.4: Learning from Exploration Suppose learning updates occurred after *all* moves, including exploratory moves. If the step-size parameter is appropriately reduced over time (but not the tendency to explore), then the state values would converge to a different set of probabilities. What (conceptually) are the two sets of probabilities computed when we do, and when we do not, learn from exploratory moves? Assuming that we do continue to make exploratory moves, which set of probabilities might be better to learn? Which would result in more wins? \square

Exercise 1.5: Other Improvements Can you think of other ways to improve the reinforcement learning player? Can you think of any better way to solve the tic-tac-toe problem as posed? \square

1.6 Summary

Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment. In our opinion, reinforcement learning is the first field to seriously address the computational issues that arise when learning from interaction with an environment in order to achieve long-term goals.

Reinforcement learning uses the formal framework of Markov decision processes to define the interaction between a learning agent and its environment in terms of states, actions, and rewards. This framework is intended to be a simple way of representing essential features of the artificial intelligence problem. These features include a sense of cause and effect, a sense of uncertainty and nondeterminism, and the existence of explicit goals.

The concepts of value and value function are key to most of the reinforcement learning methods that we consider in this book. We take the position that value functions are important for efficient search in the space of policies. The use of value functions distinguishes reinforcement learning methods from evolutionary methods that search directly in policy space guided by evaluations of entire policies.

1.7 Early History of Reinforcement Learning

The early history of reinforcement learning has two main threads, both long and rich, that were pursued independently before intertwining in modern reinforcement learning. One thread concerns learning by trial and error, and originated in the psychology of animal learning. This thread runs through some of the earliest work in artificial intelligence and led to the revival of reinforcement learning in the early 1980s. The second thread concerns the problem of optimal control and its solution using value functions and dynamic programming. For the most part, this thread did not involve learning. The two threads were mostly independent, but became interrelated to some extent around a

third, less distinct thread concerning temporal-difference methods such as that used in the tic-tac-toe example in this chapter. All three threads came together in the late 1980s to produce the modern field of reinforcement learning as we present it in this book.

The thread focusing on trial-and-error learning is the one with which we are most familiar and about which we have the most to say in this brief history. Before doing that, however, we briefly discuss the optimal control thread.

The term “optimal control” came into use in the late 1950s to describe the problem of designing a controller to minimize or maximize a measure of a dynamical system’s behavior over time. One of the approaches to this problem was developed in the mid-1950s by Richard Bellman and others through extending a nineteenth century theory of Hamilton and Jacobi. This approach uses the concepts of a dynamical system’s state and of a value function, or “optimal return function,” to define a functional equation, now often called the Bellman equation. The class of methods for solving optimal control problems by solving this equation came to be known as dynamic programming (Bellman, 1957a). Bellman (1957b) also introduced the discrete stochastic version of the optimal control problem known as Markov decision processes (MDPs). Ronald Howard (1960) devised the policy iteration method for MDPs. All of these are essential elements underlying the theory and algorithms of modern reinforcement learning.

Dynamic programming is widely considered the only feasible way of solving general stochastic optimal control problems. It suffers from what Bellman called “the curse of dimensionality,” meaning that its computational requirements grow exponentially with the number of state variables, but it is still far more efficient and more widely applicable than any other general method. Dynamic programming has been extensively developed since the late 1950s, including extensions to partially observable MDPs (surveyed by Lovejoy, 1991), many applications (surveyed by White, 1985, 1988, 1993), approximation methods (surveyed by Rust, 1996), and asynchronous methods (Bertsekas, 1982, 1983). Many excellent modern treatments of dynamic programming are available (e.g., Bertsekas, 2005, 2012; Puterman, 1994; Ross, 1983; Whittle, 1982, 1983). Bryson (1996) provides an authoritative history of optimal control.

Connections between optimal control and dynamic programming, on the one hand, and learning, on the other, were slow to be recognized. We cannot be sure about what accounted for this separation, but its main cause was likely the separation between the disciplines involved and their different goals. Also contributing may have been the prevalent view of dynamic programming as an off-line computation depending essentially on accurate system models and analytic solutions to the Bellman equation. Further, the simplest form of dynamic programming is a computation that proceeds backwards in time, making it difficult to see how it could be involved in a learning process that must proceed in a forward direction. Some of the earliest work in dynamic programming, such as that by Bellman and Dreyfus (1959), might now be classified as following a learning approach. Witten’s (1977) work (discussed below) certainly qualifies as a combination of learning and dynamic-programming ideas. Werbos (1987) argued explicitly for greater interrelation of dynamic programming and learning methods and for dynamic programming’s relevance to understanding neural and cognitive mechanisms. For us the full integration of dynamic programming methods with online learning did not occur

until the work of Chris Watkins in 1989, whose treatment of reinforcement learning using the MDP formalism has been widely adopted. Since then these relationships have been extensively developed by many researchers, most particularly by Dimitri Bertsekas and John Tsitsiklis (1996), who coined the term “neurodynamic programming” to refer to the combination of dynamic programming and artificial neural networks. Another term currently in use is “approximate dynamic programming.” These various approaches emphasize different aspects of the subject, but they all share with reinforcement learning an interest in circumventing the classical shortcomings of dynamic programming.

We consider all of the work in optimal control also to be, in a sense, work in reinforcement learning. We define a reinforcement learning method as any effective way of solving reinforcement learning problems, and it is now clear that these problems are closely related to optimal control problems, particularly stochastic optimal control problems such as those formulated as MDPs. Accordingly, we must consider the solution methods of optimal control, such as dynamic programming, also to be reinforcement learning methods. Because almost all of the conventional methods require complete knowledge of the system to be controlled, it feels a little unnatural to say that they are part of reinforcement *learning*. On the other hand, many dynamic programming algorithms are incremental and iterative. Like learning methods, they gradually reach the correct answer through successive approximations. As we show in the rest of this book, these similarities are far more than superficial. The theories and solution methods for the cases of complete and incomplete knowledge are so closely related that we feel they must be considered together as part of the same subject matter.

Let us return now to the other major thread leading to the modern field of reinforcement learning, the thread centered on the idea of trial-and-error learning. We only touch on the major points of contact here, taking up this topic in more detail in Section 14.3. According to American psychologist R. S. Woodworth (1938) the idea of trial-and-error learning goes as far back as the 1850s to Alexander Bain’s discussion of learning by “groping and experiment” and more explicitly to the British ethologist and psychologist Conway Lloyd Morgan’s 1894 use of the term to describe his observations of animal behavior. Perhaps the first to succinctly express the essence of trial-and-error learning as a principle of learning was Edward Thorndike:

Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. (Thorndike, 1911, p. 244)

Thorndike called this the “Law of Effect” because it describes the effect of reinforcing events on the tendency to select actions. Thorndike later modified the law to better account for subsequent data on animal learning (such as differences between the effects of reward and punishment), and the law in its various forms has generated considerable controversy among learning theorists (e.g., see Gallistel, 2005; Herrnstein, 1970; Kimble,

1961, 1967; Mazur, 1994). Despite this, the Law of Effect—in one form or another—is widely regarded as a basic principle underlying much behavior (e.g., Hilgard and Bower, 1975; Dennett, 1978; Campbell, 1960; Cziko, 1995). It is the basis of the influential learning theories of Clark Hull (1943, 1952) and the influential experimental methods of B. F. Skinner (1938).

The term “reinforcement” in the context of animal learning came into use well after Thorndike’s expression of the Law of Effect, first appearing in this context (to the best of our knowledge) in the 1927 English translation of Pavlov’s monograph on conditioned reflexes. Pavlov described reinforcement as the strengthening of a pattern of behavior due to an animal receiving a stimulus—a reinforcer—in an appropriate temporal relationship with another stimulus or with a response. Some psychologists extended the idea of reinforcement to include weakening as well as strengthening of behavior, and extended the idea of a reinforcer to include possibly the omission or termination of stimulus. To be considered a reinforcer, the strengthening or weakening must persist after the reinforcer is withdrawn; a stimulus that merely attracts an animal’s attention or that energizes its behavior without producing lasting changes would not be considered a reinforcer.

The idea of implementing trial-and-error learning in a computer appeared among the earliest thoughts about the possibility of artificial intelligence. In a 1948 report, Alan Turing described a design for a “pleasure-pain system” that worked along the lines of the Law of Effect:

When a configuration is reached for which the action is undetermined, a random choice for the missing data is made and the appropriate entry is made in the description, tentatively, and is applied. When a pain stimulus occurs all tentative entries are cancelled, and when a pleasure stimulus occurs they are all made permanent. (Turing, 1948)

Many ingenious electro-mechanical machines were constructed that demonstrated trial-and-error learning. The earliest may have been a machine built by Thomas Ross (1933) that was able to find its way through a simple maze and remember the path through the settings of switches. In 1951 W. Grey Walter built a version of his “mechanical tortoise” (Walter, 1950) capable of a simple form of learning. In 1952 Claude Shannon demonstrated a maze-running mouse named Theseus that used trial and error to find its way through a maze, with the maze itself remembering the successful directions via magnets and relays under its floor (see also Shannon, 1951). J. A. Deutsch (1954) described a maze-solving machine based on his behavior theory (Deutsch, 1953) that has some properties in common with model-based reinforcement learning (Chapter 8). In his PhD dissertation, Marvin Minsky (1954) discussed computational models of reinforcement learning and described his construction of an analog machine composed of components he called SNARCs (Stochastic Neural-Analog Reinforcement Calculators) meant to resemble modifiable synaptic connections in the brain (Chapter 15). The web site cyberneticzoo.com contains a wealth of information on these and many other electro-mechanical learning machines.

Building electro-mechanical learning machines gave way to programming digital computers to perform various types of learning, some of which implemented trial-and-error learning. Farley and Clark (1954) described a digital simulation of a neural-network

learning machine that learned by trial and error. But their interests soon shifted from trial-and-error learning to generalization and pattern recognition, that is, from reinforcement learning to supervised learning (Clark and Farley, 1955). This began a pattern of confusion about the relationship between these types of learning. Many researchers seemed to believe that they were studying reinforcement learning when they were actually studying supervised learning. For example, artificial neural network pioneers such as Rosenblatt (1962) and Widrow and Hoff (1960) were clearly motivated by reinforcement learning—they used the language of rewards and punishments—but the systems they studied were supervised learning systems suitable for pattern recognition and perceptual learning. Even today, some researchers and textbooks minimize or blur the distinction between these types of learning. For example, some textbooks have used the term “trial-and-error” to describe artificial neural networks that learn from training examples. This is an understandable confusion because these networks use error information to update connection weights, but this misses the essential character of trial-and-error learning as selecting actions on the basis of evaluative feedback that does not rely on knowledge of what the correct action should be.

Partly as a result of these confusions, research into genuine trial-and-error learning became rare in the 1960s and 1970s, although there were notable exceptions. In the 1960s the terms “reinforcement” and “reinforcement learning” were used in the engineering literature for the first time to describe engineering uses of trial-and-error learning (e.g., Waltz and Fu, 1965; Mendel, 1966; Fu, 1970; Mendel and McLaren, 1970). Particularly influential was Minsky’s paper “Steps Toward Artificial Intelligence” (Minsky, 1961), which discussed several issues relevant to trial-and-error learning, including prediction, expectation, and what he called the *basic credit-assignment problem for complex reinforcement learning systems*: How do you distribute credit for success among the many decisions that may have been involved in producing it? All of the methods we discuss in this book are, in a sense, directed toward solving this problem. Minsky’s paper is well worth reading today.

In the next few paragraphs we discuss some of the other exceptions and partial exceptions to the relative neglect of computational and theoretical study of genuine trial-and-error learning in the 1960s and 1970s.

One exception was the work of the New Zealand researcher John Andreae, who developed a system called STeLLA that learned by trial and error in interaction with its environment. This system included an internal model of the world and, later, an “internal monologue” to deal with problems of hidden state (Andreae, 1963, 1969; Andreae and Cashin, 1969). Andreae’s later work (1977) placed more emphasis on learning from a teacher, but still included learning by trial and error, with the generation of novel events being one of the system’s goals. A feature of this work was a “leakback process,” elaborated more fully in Andreae (1998), that implemented a credit-assignment mechanism similar to the backing-up update operations that we describe. Unfortunately, his pioneering research was not well known and did not greatly impact subsequent reinforcement learning research. Recent summaries are available (Andreae, 2017a,b).

More influential was the work of Donald Michie. In 1961 and 1963 he described a simple trial-and-error learning system for learning how to play tic-tac-toe (or naughts

and crosses) called MENACE (for Matchbox Educable Naughts and Crosses Engine). It consisted of a matchbox for each possible game position, each matchbox containing a number of colored beads, a different color for each possible move from that position. By drawing a bead at random from the matchbox corresponding to the current game position, one could determine MENACE's move. When a game was over, beads were added to or removed from the boxes used during play to reward or punish MENACE's decisions. Michie and Chambers (1968) described another tic-tac-toe reinforcement learner called GLEE (Game Learning Expectimaxing Engine) and a reinforcement learning controller called BOXES. They applied BOXES to the task of learning to balance a pole hinged to a movable cart on the basis of a failure signal occurring only when the pole fell or the cart reached the end of a track. This task was adapted from the earlier work of Widrow and Smith (1964), who used supervised learning methods, assuming instruction from a teacher already able to balance the pole. Michie and Chambers's version of pole-balancing is one of the best early examples of a reinforcement learning task under conditions of incomplete knowledge. It influenced much later work in reinforcement learning, beginning with some of our own studies (Barto, Sutton, and Anderson, 1983; Sutton, 1984). Michie (1974) consistently emphasized trial and error and learning as essential aspects of artificial intelligence.

Widrow, Gupta, and Maitra (1973) modified the Least-Mean-Square (LMS) algorithm of Widrow and Hoff (1960) to produce a reinforcement learning rule that could learn from success and failure signals instead of from training examples. They called this form of learning "selective bootstrap adaptation" and described it as "learning with a critic" instead of "learning with a teacher." They analyzed this rule and showed how it could learn to play blackjack. This was an isolated foray into reinforcement learning by Widrow, whose contributions to supervised learning were much more influential. Our use of the term "critic" is derived from Widrow, Gupta, and Maitra's paper. Buchanan, Mitchell, Smith, and Johnson (1978) independently used the term critic in the context of machine learning (see also Dietterich and Buchanan, 1984), but for them a critic was an expert system able to do more than evaluate performance.

Research on *learning automata* had a more direct influence on the trial-and-error thread leading to modern reinforcement learning research. These are methods for solving a nonassociative, purely selectional learning problem known as the *k*-armed bandit by analogy to a slot machine, or "one-armed bandit," except with *k* levers (see Chapter 2). Learning automata are simple, low-memory machines for improving the probability of reward in these problems. Learning automata originated with work in the 1960s of the Russian mathematician and physicist M. L. Tsetlin and colleagues (published posthumously in Tsetlin, 1973) and has been extensively developed since then within engineering (see Narendra and Thathachar, 1974, 1989). These developments included the study of *stochastic learning automata*, which are methods for updating action probabilities on the basis of reward signals. Although not developed in the tradition of stochastic learning automata, Harth and Tzanakou's (1974) Alopex algorithm (for *Algorithm of pattern extraction*) is a stochastic method for detecting correlations between actions and reinforcement that influenced some of our early research (Barto, Sutton, and Brouwer, 1981). Stochastic learning automata were foreshadowed by earlier work in psychology, beginning with William Estes' (1950) effort toward a statistical theory of learning and further developed by others (e.g., Bush and Mosteller, 1955; Sternberg, 1963).

The statistical learning theories developed in psychology were adopted by researchers in economics, leading to a thread of research in that field devoted to reinforcement learning. This work began in 1973 with the application of Bush and Mosteller’s learning theory to a collection of classical economic models (Cross, 1973). One goal of this research was to study artificial agents that act more like real people than do traditional idealized economic agents (Arthur, 1991). This approach expanded to the study of reinforcement learning in the context of game theory. Reinforcement learning in economics developed largely independently of the early work in reinforcement learning in artificial intelligence, though game theory remains a topic of interest in both fields (beyond the scope of this book). Camerer (2011) discusses the reinforcement learning tradition in economics, and Nowé, Vrancx, and De Hauwere (2012) provide an overview of the subject from the point of view of multi-agent extensions to the approach that we introduce in this book. Reinforcement learning in the context of game theory is a much different subject than reinforcement learning used in programs to play tic-tac-toe, checkers, and other recreational games. See, for example, Szita (2012) for an overview of this aspect of reinforcement learning and games.

John Holland (1975) outlined a general theory of adaptive systems based on selectional principles. His early work concerned trial and error primarily in its nonassociative form, as in evolutionary methods and the *k*-armed bandit. In 1976 and more fully in 1986, he introduced *classifier systems*, true reinforcement learning systems including association and value functions. A key component of Holland’s classifier systems was the “bucket-brigade algorithm” for credit assignment, which is closely related to the temporal difference algorithm used in our tic-tac-toe example and discussed in Chapter 6. Another key component was a *genetic algorithm*, an evolutionary method whose role was to evolve useful representations. Classifier systems have been extensively developed by many researchers to form a major branch of reinforcement learning research (reviewed by Urbanowicz and Moore, 2009), but genetic algorithms—which we do not consider to be reinforcement learning systems by themselves—have received much more attention, as have other approaches to evolutionary computation (e.g., Fogel, Owens and Walsh, 1966; Koza, 1992).

The individual most responsible for reviving the trial-and-error thread of reinforcement learning within artificial intelligence was Harry Klop (1972, 1975, 1982). Klop recognized that essential aspects of adaptive behavior were being lost as learning researchers came to focus almost exclusively on supervised learning. What was missing, according to Klop, were the hedonic aspects of behavior: the drive to achieve some result from the environment, to control the environment toward desired ends and away from undesired ends (see Section 15.9). This is the essential idea of trial-and-error learning. Klop’s ideas were especially influential on the authors because our assessment of them (Barto and Sutton, 1981a) led to our appreciation of the distinction between supervised and reinforcement learning, and to our eventual focus on reinforcement learning. Much of the early work that we and colleagues accomplished was directed toward showing that reinforcement learning and supervised learning were indeed different (Barto, Sutton, and Brouwer, 1981; Barto and Sutton, 1981b; Barto and Anandan, 1985). Other studies showed how reinforcement learning could address important problems in artificial neural

network learning, in particular, how it could produce learning algorithms for multilayer networks (Barto, Anderson, and Sutton, 1982; Barto and Anderson, 1985; Barto, 1985, 1986; Barto and Jordan, 1987; see Section 15.10).

We turn now to the third thread to the history of reinforcement learning, that concerning temporal-difference learning. Temporal-difference learning methods are distinctive in being driven by the difference between temporally successive estimates of the same quantity—for example, of the probability of winning in the tic-tac-toe example. This thread is smaller and less distinct than the other two, but it has played a particularly important role in the field, in part because temporal-difference methods seem to be new and unique to reinforcement learning.

The origins of temporal-difference learning are in part in animal learning psychology, in particular, in the notion of *secondary reinforcers*. A secondary reinforcer is a stimulus that has been paired with a primary reinforcer such as food or pain and, as a result, has come to take on similar reinforcing properties. Minsky (1954) may have been the first to realize that this psychological principle could be important for artificial learning systems. Arthur Samuel (1959) was the first to propose and implement a learning method that included temporal-difference ideas, as part of his celebrated checkers-playing program (Section 16.2).

Samuel made no reference to Minsky's work or to possible connections to animal learning. His inspiration apparently came from Claude Shannon's (1950) suggestion that a computer could be programmed to use an evaluation function to play chess, and that it might be able to improve its play by modifying this function online. (It is possible that these ideas of Shannon's also influenced Bellman, but we know of no evidence for this.) Minsky (1961) extensively discussed Samuel's work in his "Steps" paper, suggesting the connection to secondary reinforcement theories, both natural and artificial.

As we have discussed, in the decade following the work of Minsky and Samuel, little computational work was done on trial-and-error learning, and apparently no computational work at all was done on temporal-difference learning. In 1972, Klopf brought trial-and-error learning together with an important component of temporal-difference learning. Klopf was interested in principles that would scale to learning in large systems, and thus was intrigued by notions of local reinforcement, whereby subcomponents of an overall learning system could reinforce one another. He developed the idea of "generalized reinforcement," whereby every component (nominally, every neuron) views all of its inputs in reinforcement terms: excitatory inputs as rewards and inhibitory inputs as punishments. This is not the same idea as what we now know as temporal-difference learning, and in retrospect it is farther from it than was Samuel's work. On the other hand, Klopf linked the idea with trial-and-error learning and related it to the massive empirical database of animal learning psychology.

Sutton (1978a,b,c) developed Klopf's ideas further, particularly the links to animal learning theories, describing learning rules driven by changes in temporally successive predictions. He and Barto refined these ideas and developed a psychological model of classical conditioning based on temporal-difference learning (Sutton and Barto, 1981a; Barto and Sutton, 1982). There followed several other influential psychological models of classical conditioning based on temporal-difference learning (e.g., Klopf, 1988; Moore et al.,

1986; Sutton and Barto, 1987, 1990). Some neuroscience models developed at this time are well interpreted in terms of temporal-difference learning (Hawkins and Kandel, 1984; Byrne, Gingrich, and Baxter, 1990; Gelperin, Hopfield, and Tank, 1985; Tesauro, 1986; Friston et al., 1994), although in most cases there was no historical connection.

Our early work on temporal-difference learning was strongly influenced by animal learning theories and by Klopf's work. Relationships to Minsky's "Steps" paper and to Samuel's checkers players were recognized only afterward. By 1981, however, we were fully aware of all the prior work mentioned above as part of the temporal-difference and trial-and-error threads. At this time we developed a method for using temporal-difference learning combined with trial-and-error learning, known as the *actor–critic architecture*, and applied this method to Michie and Chambers's pole-balancing problem (Barto, Sutton, and Anderson, 1983). This method was extensively studied in Sutton's (1984) PhD dissertation and extended to use backpropagation neural networks in Anderson's (1986) PhD dissertation. Around this time, Holland (1986) incorporated temporal-difference ideas explicitly into his classifier systems in the form of his bucket-brigade algorithm. A key step was taken by Sutton (1988) by separating temporal-difference learning from control, treating it as a general prediction method. That paper also introduced the $\text{TD}(\lambda)$ algorithm and proved some of its convergence properties.

As we were finalizing our work on the actor–critic architecture in 1981, we discovered a paper by Ian Witten (1977, 1976a) which appears to be the earliest publication of a temporal-difference learning rule. He proposed the method that we now call tabular $\text{TD}(0)$ for use as part of an adaptive controller for solving MDPs. This work was first submitted for journal publication in 1974 and also appeared in Witten's 1976 PhD dissertation. Witten's work was a descendant of Andreae's early experiments with STeLLA and other trial-and-error learning systems. Thus, Witten's 1977 paper spanned both major threads of reinforcement learning research—trial-and-error learning and optimal control—while making a distinct early contribution to temporal-difference learning.

The temporal-difference and optimal control threads were fully brought together in 1989 with Chris Watkins's development of Q-learning. This work extended and integrated prior work in all three threads of reinforcement learning research. Paul Werbos (1987) contributed to this integration by arguing for the convergence of trial-and-error learning and dynamic programming since 1977. By the time of Watkins's work there had been tremendous growth in reinforcement learning research, primarily in the machine learning subfield of artificial intelligence, but also in artificial neural networks and artificial intelligence more broadly. In 1992, the remarkable success of Gerry Tesauro's backgammon playing program, TD-Gammon, brought additional attention to the field.

In the time since publication of the first edition of this book, a flourishing subfield of neuroscience developed that focuses on the relationship between reinforcement learning algorithms and reinforcement learning in the nervous system. Most responsible for this is an uncanny similarity between the behavior of temporal-difference algorithms and the activity of dopamine producing neurons in the brain, as pointed out by a number of researchers (Friston et al., 1994; Barto, 1995a; Houk, Adams, and Barto, 1995; Montague, Dayan, and Sejnowski, 1996; and Schultz, Dayan, and Montague, 1997). Chapter 15 provides an introduction to this exciting aspect of reinforcement learning. Other important

contributions made in the recent history of reinforcement learning are too numerous to mention in this brief account; we cite many of these at the end of the individual chapters in which they arise.

Bibliographical Remarks

For additional general coverage of reinforcement learning, we refer the reader to the books by Szepesvari (2010), Bertsekas and Tsitsiklis (1996), Kaelbling (1993a), and Sugiyama, Hachiya, and Morimura (2013). Books that take a control or operations research perspective include those of Si, Barto, Powell, and Wunsch (2004), Powell (2011), Lewis and Liu (2012), and Bertsekas (2012). Cao’s (2009) review places reinforcement learning in the context of other approaches to learning and optimization of stochastic dynamic systems. Three special issues of the journal *Machine Learning* focus on reinforcement learning: Sutton (1992a), Kaelbling (1996), and Singh (2002). Useful surveys are provided by Barto (1995b); Kaelbling, Littman, and Moore (1996); and Keerthi and Ravindran (1997). The volume edited by Weiring and van Otterlo (2012) provides an excellent overview of recent developments.

- 1.2** The example of Phil’s breakfast in this chapter was inspired by Agre (1988).
- 1.5** The temporal-difference method used in the tic-tac-toe example is developed in Chapter 6.

Part I: Tabular Solution Methods

In this part of the book we describe almost all the core ideas of reinforcement learning algorithms in their simplest forms: that in which the state and action spaces are small enough for the approximate value functions to be represented as arrays, or *tables*. In this case, the methods can often find exact solutions, that is, they can often find exactly the optimal value function and the optimal policy. This contrasts with the approximate methods described in the next part of the book, which only find approximate solutions, but which in return can be applied effectively to much larger problems.

The first chapter of this part of the book describes solution methods for the special case of the reinforcement learning problem in which there is only a single state, called bandit problems. The second chapter describes the general problem formulation that we treat throughout the rest of the book—finite Markov decision processes—and its main ideas including Bellman equations and value functions.

The next three chapters describe three fundamental classes of methods for solving finite Markov decision problems: dynamic programming, Monte Carlo methods, and temporal-difference learning. Each class of methods has its strengths and weaknesses. Dynamic programming methods are well developed mathematically, but require a complete and accurate model of the environment. Monte Carlo methods don't require a model and are conceptually simple, but are not well suited for step-by-step incremental computation. Finally, temporal-difference methods require no model and are fully incremental, but are more complex to analyze. The methods also differ in several ways with respect to their efficiency and speed of convergence.

The remaining two chapters describe how these three classes of methods can be combined to obtain the best features of each of them. In one chapter we describe how the strengths of Monte Carlo methods can be combined with the strengths of temporal-difference methods via multi-step bootstrapping methods. In the final chapter of this part of the book we show how temporal-difference learning methods can be combined with model learning and planning methods (such as dynamic programming) for a complete and unified solution to the tabular reinforcement learning problem.

Chapter 2

Multi-armed Bandits

The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that *evaluates* the actions taken rather than *instructs* by giving correct actions. This is what creates the need for active exploration, for an explicit search for good behavior. Purely evaluative feedback indicates how good the action taken was, but not whether it was the best or the worst action possible. Purely instructive feedback, on the other hand, indicates the correct action to take, independently of the action actually taken. This kind of feedback is the basis of supervised learning, which includes large parts of pattern classification, artificial neural networks, and system identification. In their pure forms, these two kinds of feedback are quite distinct: evaluative feedback depends entirely on the action taken, whereas instructive feedback is independent of the action taken.

In this chapter we study the evaluative aspect of reinforcement learning in a simplified setting, one that does not involve learning to act in more than one situation. This *nonassociative* setting is the one in which most prior work involving evaluative feedback has been done, and it avoids much of the complexity of the full reinforcement learning problem. Studying this case enables us to see most clearly how evaluative feedback differs from, and yet can be combined with, instructive feedback.

The particular nonassociative, evaluative feedback problem that we explore is a simple version of the k -armed bandit problem. We use this problem to introduce a number of basic learning methods which we extend in later chapters to apply to the full reinforcement learning problem. At the end of this chapter, we take a step closer to the full reinforcement learning problem by discussing what happens when the bandit problem becomes associative, that is, when the best action depends on the situation.

2.1 A k -armed Bandit Problem

Consider the following learning problem. You are faced repeatedly with a choice among k different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your

objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or *time steps*.

This is the original form of the *k-armed bandit problem*, so named by analogy to a slot machine, or “one-armed bandit,” except that it has k levers instead of one. Each action selection is like a play of one of the slot machine’s levers, and the rewards are the payoffs for hitting the jackpot. Through repeated action selections you are to maximize your winnings by concentrating your actions on the best levers. Another analogy is that of a doctor choosing between experimental treatments for a series of seriously ill patients. Each action is the selection of a treatment, and each reward is the survival or well-being of the patient. Today the term “bandit problem” is sometimes used for a generalization of the problem described above, but in this book we use it to refer just to this simple case.

In our k -armed bandit problem, each of the k actions has an expected or mean reward given that that action is selected; let us call this the *value* of that action. We denote the action selected on time step t as A_t , and the corresponding reward as R_t . The value then of an arbitrary action a , denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) \doteq \mathbb{E}[R_t | A_t = a].$$

If you knew the value of each action, then it would be trivial to solve the k -armed bandit problem: you would always select the action with highest value. We assume that you do not know the action values with certainty, although you may have estimates. We denote the estimated value of action a at time step t as $Q_t(a)$. We would like $Q_t(a)$ to be close to $q_*(a)$.

If you maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest. We call these the *greedy* actions. When you select one of these actions, we say that you are *exploiting* your current knowledge of the values of the actions. If instead you select one of the nongreedy actions, then we say you are *exploring*, because this enables you to improve your estimate of the nongreedy action’s value. Exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce the greater total reward in the long run. For example, suppose a greedy action’s value is known with certainty, while several other actions are estimated to be nearly as good but with substantial uncertainty. The uncertainty is such that at least one of these other actions probably is actually better than the greedy action, but you don’t know which one. If you have many time steps ahead on which to make action selections, then it may be better to explore the nongreedy actions and discover which of them are better than the greedy action. Reward is lower in the short run, during exploration, but higher in the long run because after you have discovered the better actions, you can exploit them many times. Because it is not possible both to explore and to exploit with any single action selection, one often refers to the “conflict” between exploration and exploitation.

In any specific case, whether it is better to explore or exploit depends in a complex way on the precise values of the estimates, uncertainties, and the number of remaining steps. There are many sophisticated methods for balancing exploration and exploitation for particular mathematical formulations of the k -armed bandit and related problems.

However, most of these methods make strong assumptions about stationarity and prior knowledge that are either violated or impossible to verify in most applications and in the full reinforcement learning problem that we consider in subsequent chapters. The guarantees of optimality or bounded loss for these methods are of little comfort when the assumptions of their theory do not apply.

In this book we do not worry about balancing exploration and exploitation in a sophisticated way; we worry only about balancing them at all. In this chapter we present several simple balancing methods for the k -armed bandit problem and show that they work much better than methods that always exploit. The need to balance exploration and exploitation is a distinctive challenge that arises in reinforcement learning; the simplicity of our version of the k -armed bandit problem enables us to show this in a particularly clear form.

2.2 Action-value Methods

We begin by looking more closely at methods for estimating the values of actions and for using the estimates to make action selection decisions, which we collectively call *action-value methods*. Recall that the true value of an action is the mean reward when that action is selected. One natural way to estimate this is by averaging the rewards actually received:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}, \quad (2.1)$$

where $\mathbb{1}_{\text{predicate}}$ denotes the random variable that is 1 if *predicate* is true and 0 if it is not. If the denominator is zero, then we instead define $Q_t(a)$ as some default value, such as 0. As the denominator goes to infinity, by the law of large numbers, $Q_t(a)$ converges to $q_*(a)$. We call this the *sample-average* method for estimating action values because each estimate is an average of the sample of relevant rewards. Of course this is just one way to estimate action values, and not necessarily the best one. Nevertheless, for now let us stay with this simple estimation method and turn to the question of how the estimates might be used to select actions.

The simplest action selection rule is to select one of the actions with the highest estimated value, that is, one of the greedy actions as defined in the previous section. If there is more than one greedy action, then a selection is made among them in some arbitrary way, perhaps randomly. We write this *greedy* action selection method as

$$A_t \doteq \operatorname{argmax}_a Q_t(a), \quad (2.2)$$

where argmax_a denotes the action a for which the expression that follows is maximized (with ties broken arbitrarily). Greedy action selection always exploits current knowledge to maximize immediate reward; it spends no time at all sampling apparently inferior actions to see if they might really be better. A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ε , instead select randomly

from among all the actions with equal probability, independently of the action-value estimates. We call methods using this near-greedy action selection rule ε -greedy methods. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the $Q_t(a)$ converge to their respective $q_*(a)$. This of course implies that the probability of selecting the optimal action converges to greater than $1 - \varepsilon$, that is, to near certainty. These are just asymptotic guarantees, however, and say little about the practical effectiveness of the methods.

Exercise 2.1 In ε -greedy action selection, for the case of two actions and $\varepsilon = 0.5$, what is the probability that the greedy action is selected? \square

2.3 The 10-armed Testbed

To roughly assess the relative effectiveness of the greedy and ε -greedy action-value methods, we compared them numerically on a suite of test problems. This was a set of 2000 randomly generated k -armed bandit problems with $k = 10$. For each bandit problem, such as the one shown in Figure 2.1, the action values, $q_*(a)$, $a = 1, \dots, 10$,

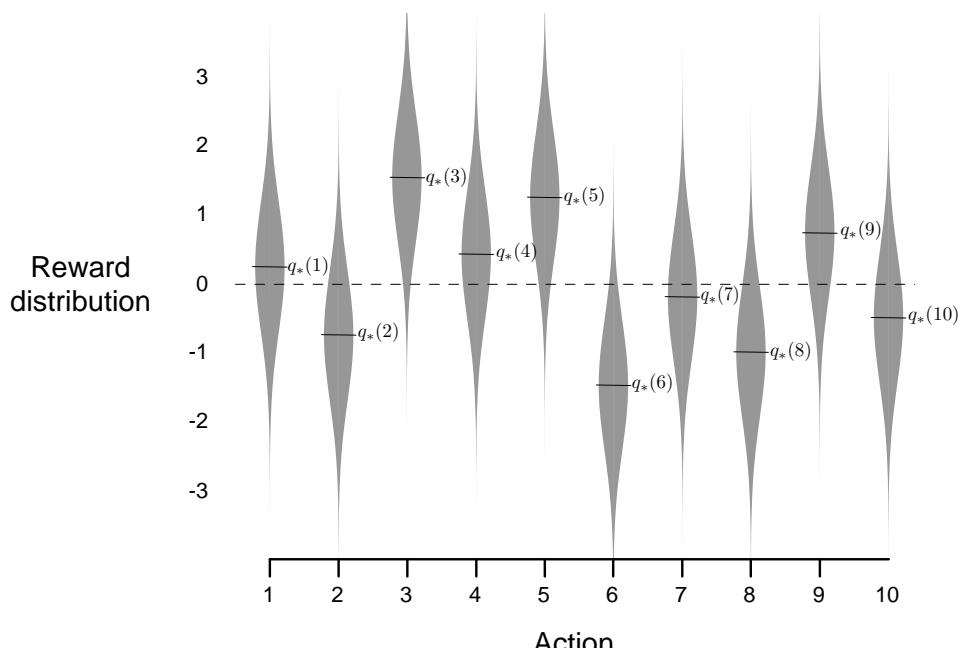


Figure 2.1: An example bandit problem from the 10-armed testbed. The true value $q_*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean $q_*(a)$, unit-variance normal distribution, as suggested by these gray distributions.

were selected according to a normal (Gaussian) distribution with mean 0 and variance 1. Then, when a learning method applied to that problem selected action A_t at time step t , the actual reward, R_t , was selected from a normal distribution with mean $q_*(A_t)$ and variance 1. These distributions are shown in gray in Figure 2.1. We call this suite of test tasks the *10-armed testbed*. For any learning method, we can measure its performance and behavior as it improves with experience over 1000 time steps when applied to one of the bandit problems. This makes up one *run*. Repeating this for 2000 independent runs, each with a different bandit problem, we obtained measures of the learning algorithm's average behavior.

Figure 2.2 compares a greedy method with two ε -greedy methods ($\varepsilon = 0.01$ and $\varepsilon = 0.1$), as described above, on the 10-armed testbed. All the methods formed their action-value estimates using the sample-average technique (with an initial estimate of 0). The upper graph shows the increase in expected reward with experience. The greedy method improved slightly faster than the other methods at the very beginning, but then leveled off at a lower level. It achieved a reward-per-step of only about 1, compared with the best possible of about 1.54 on this testbed. The greedy method performed significantly worse in the long run because it often got stuck performing suboptimal actions. The lower graph

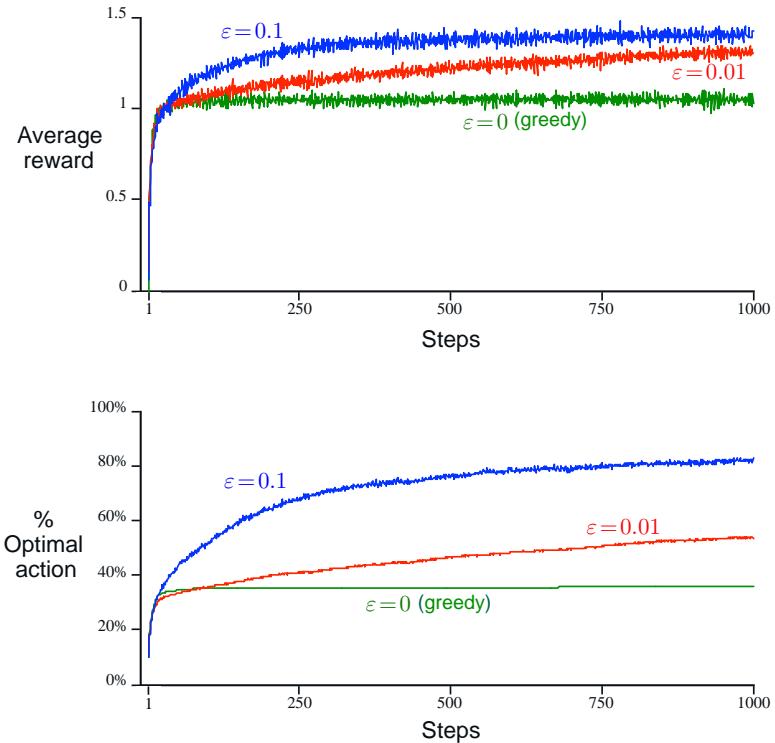


Figure 2.2: Average performance of ε -greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates.

shows that the greedy method found the optimal action in only approximately one-third of the tasks. In the other two-thirds, its initial samples of the optimal action were disappointing, and it never returned to it. The ε -greedy methods eventually performed better because they continued to explore and to improve their chances of recognizing the optimal action. The $\varepsilon = 0.1$ method explored more, and usually found the optimal action earlier, but it never selected that action more than 91% of the time. The $\varepsilon = 0.01$ method improved more slowly, but eventually would perform better than the $\varepsilon = 0.1$ method on both performance measures shown in the figure. It is also possible to reduce ε over time to try to get the best of both high and low values.

The advantage of ε -greedy over greedy methods depends on the task. For example, suppose the reward variance had been larger, say 10 instead of 1. With noisier rewards it takes more exploration to find the optimal action, and ε -greedy methods should fare even better relative to the greedy method. On the other hand, if the reward variances were zero, then the greedy method would know the true value of each action after trying it once. In this case the greedy method might actually perform best because it would soon find the optimal action and then never explore. But even in the deterministic case there is a large advantage to exploring if we weaken some of the other assumptions. For example, suppose the bandit task were nonstationary, that is, the true values of the actions changed over time. In this case exploration is needed even in the deterministic case to make sure one of the nongreedy actions has not changed to become better than the greedy one. As we shall see in the next few chapters, nonstationarity is the case most commonly encountered in reinforcement learning. Even if the underlying task is stationary and deterministic, the learner faces a set of banditlike decision tasks each of which changes over time as learning proceeds and the agent's decision-making policy changes. Reinforcement learning requires a balance between exploration and exploitation.

Exercise 2.2: Bandit example Consider a k -armed bandit problem with $k = 4$ actions, denoted 1, 2, 3, and 4. Consider applying to this problem a bandit algorithm using ε -greedy action selection, sample-average action-value estimates, and initial estimates of $Q_1(a) = 0$, for all a . Suppose the initial sequence of actions and rewards is $A_1 = 1$, $R_1 = -1$, $A_2 = 2$, $R_2 = 1$, $A_3 = 2$, $R_3 = -2$, $A_4 = 2$, $R_4 = 2$, $A_5 = 3$, $R_5 = 0$. On some of these time steps the ε case may have occurred, causing an action to be selected at random. On which time steps did this definitely occur? On which time steps could this possibly have occurred? \square

Exercise 2.3 In the comparison shown in Figure 2.2, which method will perform best in the long run in terms of cumulative reward and probability of selecting the best action? How much better will it be? Express your answer quantitatively. \square

2.4 Incremental Implementation

The action-value methods we have discussed so far all estimate action values as sample averages of observed rewards. We now turn to the question of how these averages can be computed in a computationally efficient manner, in particular, with constant memory and constant per-time-step computation.

To simplify notation we concentrate on a single action. Let R_i now denote the reward received after the i th selection of *this action*, and let Q_n denote the estimate of its action value after it has been selected $n - 1$ times, which we can now write simply as

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n - 1}.$$

The obvious implementation would be to maintain a record of all the rewards and then perform this computation whenever the estimated value was needed. However, if this is done, then the memory and computational requirements would grow over time as more rewards are seen. Each additional reward would require additional memory to store it and additional computation to compute the sum in the numerator.

As you might suspect, this is not really necessary. It is easy to devise incremental formulas for updating averages with small, constant computation required to process each new reward. Given Q_n and the n th reward, R_n , the new average of all n rewards can be computed by

$$\begin{aligned} Q_{n+1} &= \frac{1}{n} \sum_{i=1}^n R_i \\ &= \frac{1}{n} \left(R_n + \sum_{i=1}^{n-1} R_i \right) \\ &= \frac{1}{n} \left(R_n + (n-1) \frac{1}{n-1} \sum_{i=1}^{n-1} R_i \right) \\ &= \frac{1}{n} \left(R_n + (n-1)Q_n \right) \\ &= \frac{1}{n} \left(R_n + nQ_n - Q_n \right) \\ &= Q_n + \frac{1}{n} [R_n - Q_n], \end{aligned} \tag{2.3}$$

which holds even for $n = 1$, obtaining $Q_2 = R_1$ for arbitrary Q_1 . This implementation requires memory only for Q_n and n , and only the small computation (2.3) for each new reward.

This update rule (2.3) is of a form that occurs frequently throughout this book. The general form is

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize} [\text{Target} - \text{OldEstimate}]. \tag{2.4}$$

The expression $[\text{Target} - \text{OldEstimate}]$ is an *error* in the estimate. It is reduced by taking a step toward the “Target.” The target is presumed to indicate a desirable direction in which to move, though it may be noisy. In the case above, for example, the target is the n th reward.

Note that the step-size parameter (*StepSize*) used in the incremental method (2.3) changes from time step to time step. In processing the n th reward for action a , the

method uses the step-size parameter $\frac{1}{n}$. In this book we denote the step-size parameter by α or, more generally, by $\alpha_t(a)$.

Pseudocode for a complete bandit algorithm using incrementally computed sample averages and ε -greedy action selection is shown in the box below. The function $bandit(A)$ is assumed to take an action and return a corresponding reward.

A simple bandit algorithm

Initialize, for $a = 1$ to k :

$$\begin{aligned} Q(a) &\leftarrow 0 \\ N(a) &\leftarrow 0 \end{aligned}$$

Loop forever:

$$\begin{aligned} A &\leftarrow \begin{cases} \arg \max_a Q(a) & \text{with probability } 1 - \varepsilon \\ \text{a random action} & \text{with probability } \varepsilon \end{cases} \quad (\text{breaking ties randomly}) \\ R &\leftarrow bandit(A) \\ N(A) &\leftarrow N(A) + 1 \\ Q(A) &\leftarrow Q(A) + \frac{1}{N(A)} [R - Q(A)] \end{aligned}$$

2.5 Tracking a Nonstationary Problem

The averaging methods discussed so far are appropriate for stationary bandit problems, that is, for bandit problems in which the reward probabilities do not change over time. As noted earlier, we often encounter reinforcement learning problems that are effectively nonstationary. In such cases it makes sense to give more weight to recent rewards than to long-past rewards. One of the most popular ways of doing this is to use a constant step-size parameter. For example, the incremental update rule (2.3) for updating an average Q_n of the $n - 1$ past rewards is modified to be

$$Q_{n+1} \doteq Q_n + \alpha [R_n - Q_n], \quad (2.5)$$

where the step-size parameter $\alpha \in (0, 1]$ is constant. This results in Q_{n+1} being a weighted average of past rewards and the initial estimate Q_1 :

$$\begin{aligned} Q_{n+1} &= Q_n + \alpha [R_n - Q_n] \\ &= \alpha R_n + (1 - \alpha) Q_n \\ &= \alpha R_n + (1 - \alpha) [\alpha R_{n-1} + (1 - \alpha) Q_{n-1}] \\ &= \alpha R_n + (1 - \alpha) \alpha R_{n-1} + (1 - \alpha)^2 Q_{n-1} \\ &= \alpha R_n + (1 - \alpha) \alpha R_{n-1} + (1 - \alpha)^2 \alpha R_{n-2} + \\ &\quad \cdots + (1 - \alpha)^{n-1} \alpha R_1 + (1 - \alpha)^n Q_1 \\ &= (1 - \alpha)^n Q_1 + \sum_{i=1}^n \alpha (1 - \alpha)^{n-i} R_i. \end{aligned} \quad (2.6)$$

We call this a weighted average because the sum of the weights is $(1 - \alpha)^n + \sum_{i=1}^n \alpha(1 - \alpha)^{n-i} = 1$, as you can check for yourself. Note that the weight, $\alpha(1 - \alpha)^{n-i}$, given to the reward R_i depends on how many rewards ago, $n - i$, it was observed. The quantity $1 - \alpha$ is less than 1, and thus the weight given to R_i decreases as the number of intervening rewards increases. In fact, the weight decays exponentially according to the exponent on $1 - \alpha$. (If $1 - \alpha = 0$, then all the weight goes on the very last reward, R_n , because of the convention that $0^0 = 1$.) Accordingly, this is sometimes called an *exponential recency-weighted average*.

Sometimes it is convenient to vary the step-size parameter from step to step. Let $\alpha_n(a)$ denote the step-size parameter used to process the reward received after the n th selection of action a . As we have noted, the choice $\alpha_n(a) = \frac{1}{n}$ results in the sample-average method, which is guaranteed to converge to the true action values by the law of large numbers. But of course convergence is not guaranteed for all choices of the sequence $\{\alpha_n(a)\}$. A well-known result in stochastic approximation theory gives us the conditions required to assure convergence with probability 1:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty. \quad (2.7)$$

The first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence.

Note that both convergence conditions are met for the sample-average case, $\alpha_n(a) = \frac{1}{n}$, but not for the case of constant step-size parameter, $\alpha_n(a) = \alpha$. In the latter case, the second condition is not met, indicating that the estimates never completely converge but continue to vary in response to the most recently received rewards. As we mentioned above, this is actually desirable in a nonstationary environment, and problems that are effectively nonstationary are the most common in reinforcement learning. In addition, sequences of step-size parameters that meet the conditions (2.7) often converge very slowly or need considerable tuning in order to obtain a satisfactory convergence rate. Although sequences of step-size parameters that meet these convergence conditions are often used in theoretical work, they are seldom used in applications and empirical research.

Exercise 2.4 If the step-size parameters, α_n , are not constant, then the estimate Q_n is a weighted average of previously received rewards with a weighting different from that given by (2.6). What is the weighting on each prior reward for the general case, analogous to (2.6), in terms of the sequence of step-size parameters? \square

Exercise 2.5 (programming) Design and conduct an experiment to demonstrate the difficulties that sample-average methods have for nonstationary problems. Use a modified version of the 10-armed testbed in which all the $q_*(a)$ start out equal and then take independent random walks (say by adding a normally distributed increment with mean 0 and standard deviation 0.01 to all the $q_*(a)$ on each step). Prepare plots like Figure 2.2 for an action-value method using sample averages, incrementally computed, and another action-value method using a constant step-size parameter, $\alpha = 0.1$. Use $\varepsilon = 0.1$ and longer runs, say of 10,000 steps. \square

2.6 Optimistic Initial Values

All the methods we have discussed so far are dependent to some extent on the initial action-value estimates, $Q_1(a)$. In the language of statistics, these methods are *biased* by their initial estimates. For the sample-average methods, the bias disappears once all actions have been selected at least once, but for methods with constant α , the bias is permanent, though decreasing over time as given by (2.6). In practice, this kind of bias is usually not a problem and can sometimes be very helpful. The downside is that the initial estimates become, in effect, a set of parameters that must be picked by the user, if only to set them all to zero. The upside is that they provide an easy way to supply some prior knowledge about what level of rewards can be expected.

Initial action values can also be used as a simple way to encourage exploration. Suppose that instead of setting the initial action values to zero, as we did in the 10-armed testbed, we set them all to +5. Recall that the $q_*(a)$ in this problem are selected from a normal distribution with mean 0 and variance 1. An initial estimate of +5 is thus wildly optimistic. But this optimism encourages action-value methods to explore. Whichever actions are initially selected, the reward is less than the starting estimates; the learner switches to other actions, being “disappointed” with the rewards it is receiving. The result is that all actions are tried several times before the value estimates converge. The system does a fair amount of exploration even if greedy actions are selected all the time.

Figure 2.3 shows the performance on the 10-armed bandit testbed of a greedy method using $Q_1(a) = +5$, for all a . For comparison, also shown is an ε -greedy method with $Q_1(a) = 0$. Initially, the optimistic method performs worse because it explores more, but eventually it performs better because its exploration decreases with time. We call this technique for encouraging exploration *optimistic initial values*. We regard it as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration. For example, it is not well suited to nonstationary problems because its drive for exploration is inherently

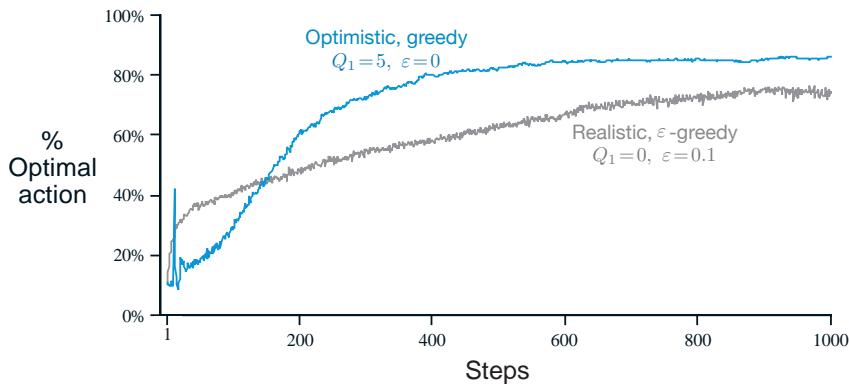


Figure 2.3: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$.

temporary. If the task changes, creating a renewed need for exploration, this method cannot help. Indeed, any method that focuses on the initial conditions in any special way is unlikely to help with the general nonstationary case. The beginning of time occurs only once, and thus we should not focus on it too much. This criticism applies as well to the sample-average methods, which also treat the beginning of time as a special event, averaging all subsequent rewards with equal weights. Nevertheless, all of these methods are very simple, and one of them—or some simple combination of them—is often adequate in practice. In the rest of this book we make frequent use of several of these simple exploration techniques.

Exercise 2.6: Mysterious Spikes The results shown in Figure 2.3 should be quite reliable because they are averages over 2000 individual, randomly chosen 10-armed bandit tasks. Why, then, are there oscillations and spikes in the early part of the curve for the optimistic method? In other words, what might make this method perform particularly better or worse, on average, on particular early steps? \square

Exercise 2.7: Unbiased Constant-Step-Size Trick In most of this chapter we have used sample averages to estimate action values because sample averages do not produce the initial bias that constant step sizes do (see the analysis leading to (2.6)). However, sample averages are not a completely satisfactory solution because they may perform poorly on nonstationary problems. Is it possible to avoid the bias of constant step sizes while retaining their advantages on nonstationary problems? One way is to use a step size of

$$\beta_n \doteq \alpha/\bar{o}_n, \quad (2.8)$$

to process the n th reward for a particular action, where $\alpha > 0$ is a conventional constant step size, and \bar{o}_n is a trace of one that starts at 0:

$$\bar{o}_n \doteq \bar{o}_{n-1} + \alpha(1 - \bar{o}_{n-1}), \quad \text{for } n > 0, \quad \text{with } \bar{o}_0 \doteq 0. \quad (2.9)$$

Carry out an analysis like that in (2.6) to show that Q_n is an exponential recency-weighted average *without initial bias*. \square

2.7 Upper-Confidence-Bound Action Selection

Exploration is needed because there is always uncertainty about the accuracy of the action-value estimates. The greedy actions are those that look best at present, but some of the other actions may actually be better. ε -greedy action selection forces the non-greedy actions to be tried, but indiscriminately, with no preference for those that are nearly greedy or particularly uncertain. It would be better to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates. One effective way of doing this is to select actions according to

$$A_t \doteq \arg \max_a \left[Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}} \right], \quad (2.10)$$

where $\ln t$ denotes the natural logarithm of t (the number that $e \approx 2.71828$ would have to be raised to in order to equal t), $N_t(a)$ denotes the number of times that action a has

been selected prior to time t (the denominator in (2.1)), and the number $c > 0$ controls the degree of exploration. If $N_t(a) = 0$, then a is considered to be a maximizing action.

The idea of this *upper confidence bound* (UCB) action selection is that the square-root term is a measure of the uncertainty or variance in the estimate of a 's value. The quantity being max'ed over is thus a sort of upper bound on the possible true value of action a , with c determining the confidence level. Each time a is selected the uncertainty is presumably reduced: $N_t(a)$ increments, and, as it appears in the denominator, the uncertainty term decreases. On the other hand, each time an action other than a is selected, t increases but $N_t(a)$ does not; because t appears in the numerator, the uncertainty estimate increases. The use of the natural logarithm means that the increases get smaller over time, but are unbounded; all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time.

Results with UCB on the 10-armed testbed are shown in Figure 2.4. UCB often performs well, as shown here, but is more difficult than ε -greedy to extend beyond bandits to the more general reinforcement learning settings considered in the rest of this book. One difficulty is in dealing with nonstationary problems; methods more complex than those presented in Section 2.5 would be needed. Another difficulty is dealing with large state spaces, particularly when using function approximation as developed in Part II of this book. In these more advanced settings the idea of UCB action selection is usually not practical.

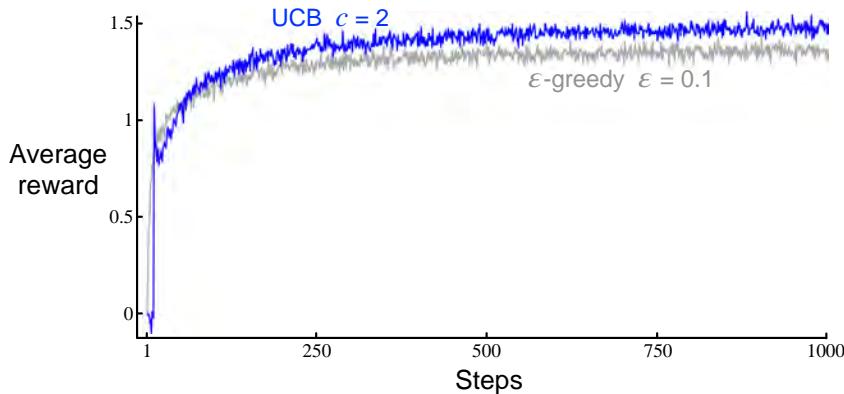


Figure 2.4: Average performance of UCB action selection on the 10-armed testbed. As shown, UCB generally performs better than ε -greedy action selection, except in the first k steps, when it selects randomly among the as-yet-untried actions.

Exercise 2.8: UCB Spikes In Figure 2.4 the UCB algorithm shows a distinct spike in performance on the 11th step. Why is this? Note that for your answer to be fully satisfactory it must explain both why the reward increases on the 11th step and why it decreases on the subsequent steps. Hint: If $c = 1$, then the spike is less prominent. \square

2.8 Gradient Bandit Algorithms

So far in this chapter we have considered methods that estimate action values and use those estimates to select actions. This is often a good approach, but it is not the only one possible. In this section we consider learning a numerical *preference* for each action a , which we denote $H_t(a) \in \mathbb{R}$. The larger the preference, the more often that action is taken, but the preference has no interpretation in terms of reward. Only the relative preference of one action over another is important; if we add 1000 to all the action preferences there is no effect on the action probabilities, which are determined according to a *soft-max distribution* (i.e., Gibbs or Boltzmann distribution) as follows:

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^k e^{H_t(b)}} \doteq \pi_t(a), \quad (2.11)$$

where here we have also introduced a useful new notation, $\pi_t(a)$, for the probability of taking action a at time t . Initially all action preferences are the same (e.g., $H_1(a) = 0$, for all a) so that all actions have an equal probability of being selected.

Exercise 2.9 Show that in the case of two actions, the soft-max distribution is the same as that given by the logistic, or sigmoid, function often used in statistics and artificial neural networks. \square

There is a natural learning algorithm for soft-max action preferences based on the idea of stochastic gradient ascent. On each step, after selecting action A_t and receiving the reward R_t , the action preferences are updated by:

$$\begin{aligned} H_{t+1}(A_t) &\doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), && \text{and} \\ H_{t+1}(a) &\doteq H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a), && \text{for all } a \neq A_t, \end{aligned} \quad (2.12)$$

where $\alpha > 0$ is a step-size parameter, and $\bar{R}_t \in \mathbb{R}$ is the average of the rewards up to but not including time t (with $\bar{R}_1 \doteq R_1$), which can be computed incrementally as described in Section 2.4 (or Section 2.5 if the problem is nonstationary).¹ The \bar{R}_t term serves as a baseline with which the reward is compared. If the reward is higher than the baseline, then the probability of taking A_t in the future is increased, and if the reward is below baseline, then the probability is decreased. The non-selected actions move in the opposite direction.

Figure 2.5 shows results with the gradient bandit algorithm on a variant of the 10-armed testbed in which the true expected rewards were selected according to a normal distribution with a mean of +4 instead of zero (and with unit variance as before). This shifting up of all the rewards has absolutely no effect on the gradient bandit algorithm because of the reward baseline term, which instantaneously adapts to the new level. But if the baseline were omitted (that is, if \bar{R}_t was taken to be constant zero in (2.12)), then performance would be significantly degraded, as shown in the figure.

¹In the empirical results in this chapter, the baseline \bar{R}_t also included R_t .

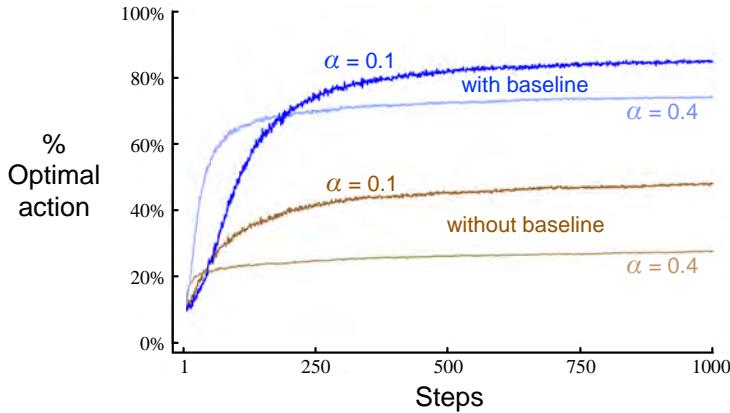


Figure 2.5: Average performance of the gradient bandit algorithm with and without a reward baseline on the 10-armed testbed when the $q_*(a)$ are chosen to be near +4 rather than near zero.

The Bandit Gradient Algorithm as Stochastic Gradient Ascent

One can gain a deeper insight into the gradient bandit algorithm by understanding it as a stochastic approximation to gradient ascent. In exact *gradient ascent*, each action preference $H_t(a)$ would be incremented in proportion to the increment's effect on performance:

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \quad (2.13)$$

where the measure of performance here is the expected reward:

$$\mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x),$$

and the measure of the increment's effect is the *partial derivative* of this performance measure with respect to the action preference. Of course, it is not possible to implement gradient ascent exactly in our case because by assumption we do not know the $q_*(x)$, but in fact the updates of our algorithm (2.12) are equal to (2.13) in expected value, making the algorithm an instance of *stochastic gradient ascent*. The calculations showing this require only beginning calculus, but take several

steps. First we take a closer look at the exact performance gradient:

$$\begin{aligned}\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \left[\sum_x \pi_t(x) q_*(x) \right] \\ &= \sum_x q_*(x) \frac{\partial \pi_t(x)}{\partial H_t(a)} \\ &= \sum_x (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)},\end{aligned}$$

where B_t , called the *baseline*, can be any scalar that does not depend on x . We can include a baseline here without changing the equality because the gradient sums to zero over all the actions, $\sum_x \frac{\partial \pi_t(x)}{\partial H_t(a)} = 0$. As $H_t(a)$ is changed, some actions' probabilities go up and some go down, but the sum of the changes must be zero because the sum of the probabilities is always one.

Next we multiply each term of the sum by $\pi_t(x)/\pi_t(x)$:

$$\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} = \sum_x \pi_t(x) (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)} / \pi_t(x).$$

The equation is now in the form of an expectation, summing over all possible values x of the random variable A_t , then multiplying by the probability of taking those values. Thus:

$$\begin{aligned}&= \mathbb{E} \left[(q_*(A_t) - B_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t) \right] \\ &= \mathbb{E} \left[(R_t - \bar{R}_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t) \right],\end{aligned}$$

where here we have chosen the baseline $B_t = \bar{R}_t$ and substituted R_t for $q_*(A_t)$, which is permitted because $\mathbb{E}[R_t|A_t] = q_*(A_t)$. Shortly we will establish that $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$, where $\mathbb{1}_{a=x}$ is defined to be 1 if $a = x$, else 0. Assuming that for now, we have

$$\begin{aligned}&= \mathbb{E} [(R_t - \bar{R}_t) \pi_t(A_t) (\mathbb{1}_{a=A_t} - \pi_t(a)) / \pi_t(A_t)] \\ &= \mathbb{E} [(R_t - \bar{R}_t) (\mathbb{1}_{a=A_t} - \pi_t(a))].\end{aligned}$$

Recall that our plan has been to write the performance gradient as an expectation of something that we can sample on each step, as we have just done, and then update on each step in proportion to the sample. Substituting a sample of the expectation above for the performance gradient in (2.13) yields:

$$H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a)), \quad \text{for all } a,$$

which you may recognize as being equivalent to our original algorithm (2.12).

Thus it remains only to show that $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$, as we assumed. Recall the standard quotient rule for derivatives:

$$\frac{\partial}{\partial x} \left[\frac{f(x)}{g(x)} \right] = \frac{\frac{\partial f(x)}{\partial x} g(x) - f(x) \frac{\partial g(x)}{\partial x}}{g(x)^2}.$$

Using this, we can write

$$\begin{aligned} \frac{\partial \pi_t(x)}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \pi_t(x) \\ &= \frac{\partial}{\partial H_t(a)} \left[\frac{e^{H_t(x)}}{\sum_{y=1}^k e^{H_t(y)}} \right] \\ &= \frac{\frac{\partial e^{H_t(x)}}{\partial H_t(a)} \sum_{y=1}^k e^{H_t(y)} - e^{H_t(x)} \frac{\partial \sum_{y=1}^k e^{H_t(y)}}{\partial H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)} \right)^2} \quad (\text{by the quotient rule}) \\ &= \frac{\mathbb{1}_{a=x} e^{H_t(x)} \sum_{y=1}^k e^{H_t(y)} - e^{H_t(x)} e^{H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)} \right)^2} \quad (\text{because } \frac{\partial e^x}{\partial x} = e^x) \\ &= \frac{\mathbb{1}_{a=x} e^{H_t(x)}}{\sum_{y=1}^k e^{H_t(y)}} - \frac{e^{H_t(x)} e^{H_t(a)}}{\left(\sum_{y=1}^k e^{H_t(y)} \right)^2} \\ &= \mathbb{1}_{a=x} \pi_t(x) - \pi_t(x) \pi_t(a) \\ &= \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a)). \end{aligned}$$

Q.E.D.

We have just shown that the expected update of the gradient bandit algorithm is equal to the gradient of expected reward, and thus that the algorithm is an instance of stochastic gradient ascent. This assures us that the algorithm has robust convergence properties.

Note that we did not require any properties of the reward baseline other than that it does not depend on the selected action. For example, we could have set it to zero, or to 1000, and the algorithm would still be an instance of stochastic gradient ascent. The choice of the baseline does not affect the expected update of the algorithm, but it does affect the variance of the update and thus the rate of convergence (as shown, for example, in Figure 2.5). Choosing it as the average of the rewards may not be the very best, but it is simple and works well in practice.

2.9 Associative Search (Contextual Bandits)

So far in this chapter we have considered only nonassociative tasks, that is, tasks in which there is no need to associate different actions with different situations. In these tasks the learner either tries to find a single best action when the task is stationary, or tries to track the best action as it changes over time when the task is nonstationary. However, in a general reinforcement learning task there is more than one situation, and the goal is to learn a policy: a mapping from situations to the actions that are best in those situations. To set the stage for the full problem, we briefly discuss the simplest way in which nonassociative tasks extend to the associative setting.

As an example, suppose there are several different k -armed bandit tasks, and that on each step you confront one of these chosen at random. Thus, the bandit task changes randomly from step to step. If the probabilities with which each task is selected for you do not change over time, this would appear as a single stationary k -armed bandit task, and you could use one of the methods described in this chapter. Now suppose, however, that when a bandit task is selected for you, you are given some distinctive clue about its identity (but not its action values). Maybe you are facing an actual slot machine that changes the color of its display as it changes its action values. Now you can learn a policy associating each task, signaled by the color you see, with the best action to take when facing that task—for instance, if red, select arm 1; if green, select arm 2. With the right policy you can usually do much better than you could in the absence of any information distinguishing one bandit task from another.

This is an example of an *associative search* task, so called because it involves both trial-and-error learning to *search* for the best actions, and *association* of these actions with the situations in which they are best. Associative search tasks are often now called *contextual bandits* in the literature. Associative search tasks are intermediate between the k -armed bandit problem and the full reinforcement learning problem. They are like the full reinforcement learning problem in that they involve learning a policy, but they are also like our version of the k -armed bandit problem in that each action affects only the immediate reward. If actions are allowed to affect the *next situation* as well as the reward, then we have the full reinforcement learning problem. We present this problem in the next chapter and consider its ramifications throughout the rest of the book.

Exercise 2.10 Suppose you face a 2-armed bandit task whose true action values change randomly from time step to time step. Specifically, suppose that, for any time step, the true values of actions 1 and 2 are respectively 10 and 20 with probability 0.5 (case A), and 90 and 80 with probability 0.5 (case B). If you are not able to tell which case you face at any step, what is the best expected reward you can achieve and how should you behave to achieve it? Now suppose that on each step you are told whether you are facing case A or case B (although you still don't know the true action values). This is an associative search task. What is the best expected reward you can achieve in this task, and how should you behave to achieve it? \square

2.10 Summary

We have presented in this chapter several simple ways of balancing exploration and exploitation. The ϵ -greedy methods choose randomly a small fraction of the time, whereas UCB methods choose deterministically but achieve exploration by subtly favoring at each step the actions that have so far received fewer samples. Gradient bandit algorithms estimate not action values, but action preferences, and favor the more preferred actions in a graded, probabilistic manner using a soft-max distribution. The simple expedient of initializing estimates optimistically causes even greedy methods to explore significantly.

It is natural to ask which of these methods is best. Although this is a difficult question to answer in general, we can certainly run them all on the 10-armed testbed that we have used throughout this chapter and compare their performances. A complication is that they all have a parameter; to get a meaningful comparison we have to consider their performance as a function of their parameter. Our graphs so far have shown the course of learning over time for each algorithm and parameter setting, to produce a *learning curve* for that algorithm and parameter setting. If we plotted learning curves for all algorithms and all parameter settings, then the graph would be too complex and crowded to make clear comparisons. Instead we summarize a complete learning curve by its average value over the 1000 steps; this value is proportional to the area under the learning curve. Figure 2.6 shows this measure for the various bandit algorithms from this chapter, each as a function of its own parameter shown on a single scale on the x-axis. This kind of graph is called a *parameter study*. Note that the parameter values are varied by factors of two and presented on a log scale. Note also the characteristic inverted-U shapes of each algorithm's performance; all the algorithms perform best at an intermediate value of their parameter, neither too large nor too small. In assessing

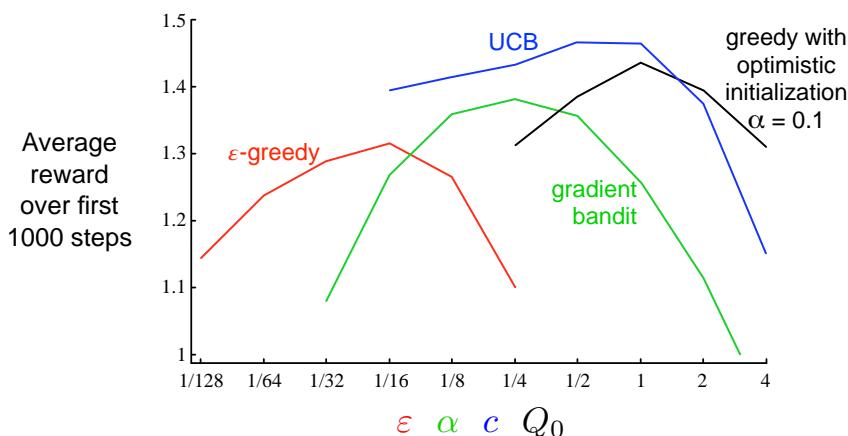


Figure 2.6: A parameter study of the various bandit algorithms presented in this chapter. Each point is the average reward obtained over 1000 steps with a particular algorithm at a particular setting of its parameter.

a method, we should attend not just to how well it does at its best parameter setting, but also to how sensitive it is to its parameter value. All of these algorithms are fairly insensitive, performing well over a range of parameter values varying by about an order of magnitude. Overall, on this problem, UCB seems to perform best.

Despite their simplicity, in our opinion the methods presented in this chapter can fairly be considered the state of the art. There are more sophisticated methods, but their complexity and assumptions make them impractical for the full reinforcement learning problem that is our real focus. Starting in Chapter 5 we present learning methods for solving the full reinforcement learning problem that use in part the simple methods explored in this chapter.

Although the simple methods explored in this chapter may be the best we can do at present, they are far from a fully satisfactory solution to the problem of balancing exploration and exploitation.

One well-studied approach to balancing exploration and exploitation in k -armed bandit problems is to compute a special kind of action value called a *Gittins index*. In certain important special cases, this computation is tractable and leads directly to optimal solutions, although it does require complete knowledge of the prior distribution of possible problems, which we generally assume is not available. In addition, neither the theory nor the computational tractability of this approach appear to generalize to the full reinforcement learning problem that we consider in the rest of the book.

The Gittins-index approach is an instance of *Bayesian* methods, which assume a known initial distribution over the action values and then update the distribution exactly after each step (assuming that the true action values are stationary). In general, the update computations can be very complex, but for certain special distributions (called *conjugate priors*) they are easy. One possibility is to then select actions at each step according to their posterior probability of being the best action. This method, sometimes called *posterior sampling* or *Thompson sampling*, often performs similarly to the best of the distribution-free methods we have presented in this chapter.

In the Bayesian setting it is even conceivable to compute the *optimal* balance between exploration and exploitation. One can compute for any possible action the probability of each possible immediate reward and the resultant posterior distributions over action values. This evolving distribution becomes the *information state* of the problem. Given a horizon, say of 1000 steps, one can consider all possible actions, all possible resulting rewards, all possible next actions, all next rewards, and so on for all 1000 steps. Given the assumptions, the rewards and probabilities of each possible chain of events can be determined; one need only pick the best. But the tree of possibilities grows extremely rapidly; even if there were only two actions and two rewards, the tree would have 2^{2000} leaves. It is generally not feasible to perform this immense computation exactly, but perhaps it could be approximated efficiently. This approach would effectively turn the bandit problem into an instance of the full reinforcement learning problem. In the end, we may be able to use approximate reinforcement learning methods such as those presented in Part II of this book to approach this optimal solution. But that is a topic for research and beyond the scope of this introductory book.

Exercise 2.11 (programming) Make a figure analogous to Figure 2.6 for the nonstationary case outlined in Exercise 2.5. Include the constant-step-size ε -greedy algorithm with $\alpha=0.1$. Use runs of 200,000 steps and, as a performance measure for each algorithm and parameter setting, use the average reward over the last 100,000 steps. \square

Bibliographical and Historical Remarks

- 2.1** Bandit problems have been studied in statistics, engineering, and psychology. In statistics, bandit problems fall under the heading “sequential design of experiments,” introduced by Thompson (1933, 1934) and Robbins (1952), and studied by Bellman (1956). Berry and Fristedt (1985) provide an extensive treatment of bandit problems from the perspective of statistics. Narendra and Thathachar (1989) treat bandit problems from the engineering perspective, providing a good discussion of the various theoretical traditions that have focused on them. In psychology, bandit problems have played roles in statistical learning theory (e.g., Bush and Mosteller, 1955; Estes, 1950).
- The term *greedy* is often used in the heuristic search literature (e.g., Pearl, 1984). The conflict between exploration and exploitation is known in control engineering as the conflict between identification (or estimation) and control (e.g., Witten, 1976b). Feldbaum (1965) called it the *dual control* problem, referring to the need to solve the two problems of identification and control simultaneously when trying to control a system under uncertainty. In discussing aspects of genetic algorithms, Holland (1975) emphasized the importance of this conflict, referring to it as the conflict between the need to exploit and the need for new information.
- 2.2** Action-value methods for our k -armed bandit problem were first proposed by Thathachar and Sastry (1985). These are often called *estimator algorithms* in the learning automata literature. The term *action value* is due to Watkins (1989). The first to use ε -greedy methods may also have been Watkins (1989, p. 187), but the idea is so simple that some earlier use seems likely.
- 2.4–5** This material falls under the general heading of stochastic iterative algorithms, which is well covered by Bertsekas and Tsitsiklis (1996).
- 2.6** Optimistic initialization was used in reinforcement learning by Sutton (1996).
- 2.7** Early work on using estimates of the upper confidence bound to select actions was done by Lai and Robbins (1985), Kaelbling (1993b), and Agrawal (1995). The UCB algorithm we present here is called UCB1 in the literature and was first developed by Auer, Cesa-Bianchi and Fischer (2002).
- 2.8** Gradient bandit algorithms are a special case of the gradient-based reinforcement learning algorithms introduced by Williams (1992) that later developed into the actor-critic and policy-gradient algorithms that we treat later in this book. Our development here was influenced by that by Balaraman Ravindran (personal

communication). Further discussion of the choice of baseline is provided by Greensmith, Bartlett, and Baxter (2002, 2004) and by Dick (2015). Early systematic studies of algorithms like this were done by Sutton (1984).

The term *soft-max* for the action selection rule (2.11) is due to Bridle (1990). This rule appears to have been first proposed by Luce (1959).

- 2.9** The term *associative search* and the corresponding problem were introduced by Barto, Sutton, and Brouwer (1981). The term *associative reinforcement learning* has also been used for associative search (Barto and Anandan, 1985), but we prefer to reserve that term as a synonym for the full reinforcement learning problem (as in Sutton, 1984). (And, as we noted, the modern literature also uses the term “contextual bandits” for this problem.) We note that Thorndike’s Law of Effect (quoted in Chapter 1) describes associative search by referring to the formation of associative links between situations (states) and actions. According to the terminology of operant, or instrumental, conditioning (e.g., Skinner, 1938), a discriminative stimulus is a stimulus that signals the presence of a particular reinforcement contingency. In our terms, different discriminative stimuli correspond to different states.

- 2.10** Bellman (1956) was the first to show how dynamic programming could be used to compute the optimal balance between exploration and exploitation within a Bayesian formulation of the problem. The Gittins index approach is due to Gittins and Jones (1974). Duff (1995) showed how it is possible to learn Gittins indices for bandit problems through reinforcement learning. The survey by Kumar (1985) provides a good discussion of Bayesian and non-Bayesian approaches to these problems. The term *information state* comes from the literature on partially observable MDPs; see, for example, Lovejoy (1991).

Other theoretical research focuses on the efficiency of exploration, usually expressed as how quickly an algorithm can approach an optimal decision-making policy. One way to formalize exploration efficiency is by adapting to reinforcement learning the notion of *sample complexity* for a supervised learning algorithm, which is the number of training examples the algorithm needs to attain a desired degree of accuracy in learning the target function. A definition of the sample complexity of exploration for a reinforcement learning algorithm is the number of time steps in which the algorithm does not select near-optimal actions (Kakade, 2003). Li (2012) discusses this and several other approaches in a survey of theoretical approaches to exploration efficiency in reinforcement learning. A thorough modern treatment of Thompson sampling is provided by Russo et al. (2018).

Chapter 3

Finite Markov Decision Processes

In this chapter we introduce the formal problem of finite Markov decision processes, or finite MDPs, which we try to solve in the rest of the book. This problem involves evaluative feedback, as in bandits, but also an associative aspect—choosing different actions in different situations. MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to trade off immediate and delayed reward. Whereas in bandit problems we estimated the value $q_*(a)$ of each action a , in MDPs we estimate the value $q_*(s, a)$ of each action a in each state s , or we estimate the value $v_*(s)$ of each state given optimal action selections. These state-dependent quantities are essential to accurately assigning credit for long-term consequences to individual action selections.

MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. We introduce key elements of the problem’s mathematical structure, such as returns, value functions, and Bellman equations. We try to convey the wide range of applications that can be formulated as finite MDPs. As in all of artificial intelligence, there is a tension between breadth of applicability and mathematical tractability. In this chapter we introduce this tension and discuss some of the trade-offs and challenges that it implies. Some ways in which reinforcement learning can be taken beyond MDPs are treated in Chapter 17.

3.1 The Agent–Environment Interface

MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The learner and decision maker is called the *agent*. The thing it interacts with, comprising everything outside the agent, is called the *environment*. These interact continually, the agent selecting actions and the environment responding to

these actions and presenting new situations to the agent.¹ The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions.

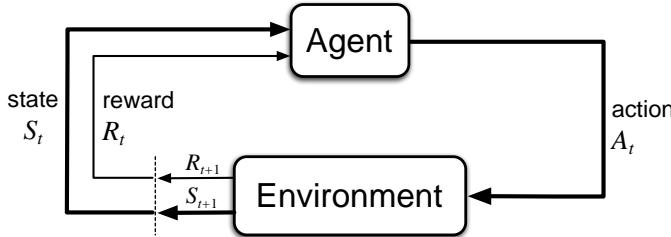


Figure 3.1: The agent–environment interaction in a Markov decision process.

More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \dots$.² At each time step t , the agent receives some representation of the environment’s *state*, $S_t \in \mathcal{S}$, and on that basis selects an *action*, $A_t \in \mathcal{A}(s)$.³ One time step later, in part as a consequence of its action, the agent receives a numerical *reward*, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, S_{t+1} .⁴ The MDP and agent together thereby give rise to a sequence or *trajectory* that begins like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots \quad (3.1)$$

In a *finite* MDP, the sets of states, actions, and rewards (\mathcal{S} , \mathcal{A} , and \mathcal{R}) all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t , given particular values of the preceding state and action:

$$p(s', r | s, a) \doteq \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}, \quad (3.2)$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. The function p defines the *dynamics* of the MDP. The dot over the equals sign in the equation reminds us that it is a definition (in this case of the function p) rather than a fact that follows from previous definitions. The dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is an ordinary deterministic function of four arguments. The ‘|’ in the middle of it comes from the notation for conditional probability,

¹We use the terms *agent*, *environment*, and *action* instead of the engineers’ terms *controller*, *controlled system* (or *plant*), and *control signal* because they are meaningful to a wider audience.

²We restrict attention to discrete time to keep things as simple as possible, even though many of the ideas can be extended to the continuous-time case (e.g., see Bertsekas and Tsitsiklis, 1996; Doya, 1996).

³To simplify notation, we sometimes assume the special case in which the action set is the same in all states and write it simply as \mathcal{A} .

⁴We use R_{t+1} instead of R_t to denote the reward due to A_t because it emphasizes that the next reward and next state, R_{t+1} and S_{t+1} , are jointly determined. Unfortunately, both conventions are widely used in the literature.

but here it just reminds us that p specifies a probability distribution for each choice of s and a , that is, that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s). \quad (3.3)$$

In a *Markov* decision process, the probabilities given by p completely characterize the environment’s dynamics. That is, the probability of each possible value for S_t and R_t depends on the immediately preceding state and action, S_{t-1} and A_{t-1} , and, given them, not at all on earlier states and actions. This is best viewed as a restriction not on the decision process, but on the *state*. The state must include information about all aspects of the past agent–environment interaction that make a difference for the future. If it does, then the state is said to have the *Markov property*. We will assume the Markov property throughout this book, though starting in Part II we will consider approximation methods that do not rely on it, and in Chapter 17 we consider how a Markov state can be efficiently learned and constructed from non-Markov observations.

From the four-argument dynamics function, p , one can compute anything else one might want to know about the environment, such as the *state-transition probabilities* (which we denote, with a slight abuse of notation, as a three-argument function $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$),

$$p(s' | s, a) \doteq \Pr\{S_t = s' | S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r | s, a). \quad (3.4)$$

We can also compute the expected rewards for state–action pairs as a two-argument function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$:

$$r(s, a) \doteq \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r | s, a), \quad (3.5)$$

and the expected rewards for state–action–next-state triples as a three-argument function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$,

$$r(s, a, s') \doteq \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r | s, a)}{p(s' | s, a)}. \quad (3.6)$$

In this book, we usually use the four-argument p function (3.2), but each of these other notations are also occasionally convenient.

The MDP framework is abstract and flexible and can be applied to many different problems in many different ways. For example, the time steps need not refer to fixed intervals of real time; they can refer to arbitrary successive stages of decision making and acting. The actions can be low-level controls, such as the voltages applied to the motors of a robot arm, or high-level decisions, such as whether or not to have lunch or to go to graduate school. Similarly, the states can take a wide variety of forms. They can be completely determined by low-level sensations, such as direct sensor readings, or they can be more high-level and abstract, such as symbolic descriptions of objects in a room. Some of what makes up a state could be based on memory of past sensations or

even be entirely mental or subjective. For example, an agent could be in the state of not being sure where an object is, or of having just been surprised in some clearly defined sense. Similarly, some actions might be totally mental or computational. For example, some actions might control what an agent chooses to think about, or where it focuses its attention. In general, actions can be any decisions we want to learn how to make, and states can be anything we can know that might be useful in making them.

In particular, the boundary between agent and environment is typically not the same as the physical boundary of a robot's or an animal's body. Usually, the boundary is drawn closer to the agent than that. For example, the motors and mechanical linkages of a robot and its sensing hardware should usually be considered parts of the environment rather than parts of the agent. Similarly, if we apply the MDP framework to a person or animal, the muscles, skeleton, and sensory organs should be considered part of the environment. Rewards, too, presumably are computed inside the physical bodies of natural and artificial learning systems, but are considered external to the agent.

The general rule we follow is that anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of its environment. We do not assume that everything in the environment is unknown to the agent. For example, the agent often knows quite a bit about how its rewards are computed as a function of its actions and the states in which they are taken. But we always consider the reward computation to be external to the agent because it defines the task facing the agent and thus must be beyond its ability to change arbitrarily. In fact, in some cases the agent may know *everything* about how its environment works and still face a difficult reinforcement learning task, just as we may know exactly how a puzzle like Rubik's cube works, but still be unable to solve it. The agent–environment boundary represents the limit of the agent's *absolute control*, not of its knowledge.

The agent–environment boundary can be located at different places for different purposes. In a complicated robot, many different agents may be operating at once, each with its own boundary. For example, one agent may make high-level decisions which form part of the states faced by a lower-level agent that implements the high-level decisions. In practice, the agent–environment boundary is determined once one has selected particular states, actions, and rewards, and thus has identified a specific decision-making task of interest.

The MDP framework is a considerable abstraction of the problem of goal-directed learning from interaction. It proposes that whatever the details of the sensory, memory, and control apparatus, and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal to represent the choices made by the agent (the actions), one signal to represent the basis on which the choices are made (the states), and one signal to define the agent's goal (the rewards). This framework may not be sufficient to represent all decision-learning problems usefully, but it has proved to be widely useful and applicable.

Of course, the particular states and actions vary greatly from task to task, and how they are represented can strongly affect performance. In reinforcement learning, as in other kinds of learning, such representational choices are at present more art than science.

In this book we offer some advice and examples regarding good ways of representing states and actions, but our primary focus is on general principles for learning how to behave once the representations have been selected.

Example 3.1: Bioreactor Suppose reinforcement learning is being applied to determine moment-by-moment temperatures and stirring rates for a bioreactor (a large vat of nutrients and bacteria used to produce useful chemicals). The actions in such an application might be target temperatures and target stirring rates that are passed to lower-level control systems that, in turn, directly activate heating elements and motors to attain the targets. The states are likely to be thermocouple and other sensory readings, perhaps filtered and delayed, plus symbolic inputs representing the ingredients in the vat and the target chemical. The rewards might be moment-by-moment measures of the rate at which the useful chemical is produced by the bioreactor. Notice that here each state is a list, or vector, of sensor readings and symbolic inputs, and each action is a vector consisting of a target temperature and a stirring rate. It is typical of reinforcement learning tasks to have states and actions with such structured representations. Rewards, on the other hand, are always single numbers. ■

Example 3.2: Pick-and-Place Robot Consider using reinforcement learning to control the motion of a robot arm in a repetitive pick-and-place task. If we want to learn movements that are fast and smooth, the learning agent will have to control the motors directly and have low-latency information about the current positions and velocities of the mechanical linkages. The actions in this case might be the voltages applied to each motor at each joint, and the states might be the latest readings of joint angles and velocities. The reward might be +1 for each object successfully picked up and placed. To encourage smooth movements, on each time step a small, negative reward could be given as a function of the moment-to-moment jerkiness of the motion. ■

Exercise 3.1 Devise three example tasks of your own that fit into the MDP framework, identifying for each its states, actions, and rewards. Make the three examples as *different* from each other as possible. The framework is abstract and flexible and can be applied in many different ways. Stretch its limits in some way in at least one of your examples. □

Exercise 3.2 Is the MDP framework adequate to usefully represent *all* goal-directed learning tasks? Can you think of any clear exceptions? □

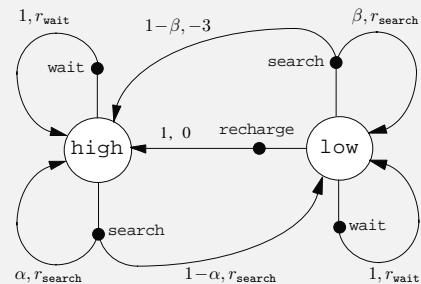
Exercise 3.3 Consider the problem of driving. You could define the actions in terms of the accelerator, steering wheel, and brake, that is, where your body meets the machine. Or you could define them farther out—say, where the rubber meets the road, considering your actions to be tire torques. Or you could define them farther in—say, where your brain meets your body, the actions being muscle twitches to control your limbs. Or you could go to a really high level and say that your actions are your choices of *where* to drive. What is the right level, the right place to draw the line between agent and environment? On what basis is one location of the line to be preferred over another? Is there any fundamental reason for preferring one location over another, or is it a free choice? □

Example 3.3 Recycling Robot

A mobile robot has the job of collecting empty soda cans in an office environment. It has sensors for detecting cans, and an arm and gripper that can pick them up and place them in an onboard bin; it runs on a rechargeable battery. The robot's control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper. High-level decisions about how to search for cans are made by a reinforcement learning agent based on the current charge level of the battery. To make a simple example, we assume that only two charge levels can be distinguished, comprising a small state set $S = \{\text{high}, \text{low}\}$. In each state, the agent can decide whether to (1) actively **search** for a can for a certain period of time, (2) remain stationary and **wait** for someone to bring it a can, or (3) head back to its home base to **recharge** its battery. When the energy level is **high**, recharging would always be foolish, so we do not include it in the action set for this state. The action sets are then $\mathcal{A}(\text{high}) = \{\text{search}, \text{wait}\}$ and $\mathcal{A}(\text{low}) = \{\text{search}, \text{wait}, \text{recharge}\}$.

The rewards are zero most of the time, but become positive when the robot secures an empty can, or large and negative if the battery runs all the way down. The best way to find cans is to actively search for them, but this runs down the robot's battery, whereas waiting does not. Whenever the robot is searching, the possibility exists that its battery will become depleted. In this case the robot must shut down and wait to be rescued (producing a low reward). If the energy level is **high**, then a period of active search can always be completed without risk of depleting the battery. A period of searching that begins with a **high** energy level leaves the energy level **high** with probability α and reduces it to **low** with probability $1 - \alpha$. On the other hand, a period of searching undertaken when the energy level is **low** leaves it **low** with probability β and depletes the battery with probability $1 - \beta$. In the latter case, the robot must be rescued, and the battery is then recharged back to **high**. Each can collected by the robot counts as a unit reward, whereas a reward of -3 results whenever the robot has to be rescued. Let r_{search} and r_{wait} , with $r_{\text{search}} > r_{\text{wait}}$, denote the expected number of cans the robot will collect (and hence the expected reward) while searching and while waiting respectively. Finally, suppose that no cans can be collected during a run home for recharging, and that no cans can be collected on a step in which the battery is depleted. This system is then a finite MDP, and we can write down the transition probabilities and the expected rewards, with dynamics as indicated in the table on the left:

s	a	s'	$p(s' s, a)$	$r(s, a, s')$
high	search	high	α	r_{search}
high	search	low	$1 - \alpha$	r_{search}
low	search	high	$1 - \beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-



Note that there is a row in the table for each possible combination of current state, s , action, $a \in \mathcal{A}(s)$, and next state, s' . Some transitions have zero probability of occurring, so no expected reward is specified for them. Shown on the right is another useful way of

summarizing the dynamics of a finite MDP, as a *transition graph*. There are two kinds of nodes: *state nodes* and *action nodes*. There is a state node for each possible state (a large open circle labeled by the name of the state), and an action node for each state-action pair (a small solid circle labeled by the name of the action and connected by a line to the state node). Starting in state s and taking action a moves you along the line from state node s to action node (s, a) . Then the environment responds with a transition to the next state's node via one of the arrows leaving action node (s, a) . Each arrow corresponds to a triple (s, s', a) , where s' is the next state, and we label the arrow with the transition probability, $p(s'|s, a)$, and the expected reward for that transition, $r(s, a, s')$. Note that the transition probabilities labeling the arrows leaving an action node always sum to 1.

Exercise 3.4 Give a table analogous to that in Example 3.3, but for $p(s', r|s, a)$. It should have columns for s , a , s' , r , and $p(s', r|s, a)$, and a row for every 4-tuple for which $p(s', r|s, a) > 0$. \square

3.2 Goals and Rewards

In reinforcement learning, the purpose or goal of the agent is formalized in terms of a special signal, called the *reward*, passing from the environment to the agent. At each time step, the reward is a simple number, $R_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run. We can clearly state this informal idea as the *reward hypothesis*:

That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning.

Although formulating goals in terms of reward signals might at first appear limiting, in practice it has proved to be flexible and widely applicable. The best way to see this is to consider examples of how it has been, or could be, used. For example, to make a robot learn to walk, researchers have provided reward on each time step proportional to the robot's forward motion. In making a robot learn how to escape from a maze, the reward is often -1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible. To make a robot learn to find and collect empty soda cans for recycling, one might give it a reward of zero most of the time, and then a reward of $+1$ for each can collected. One might also want to give the robot negative rewards when it bumps into things or when somebody yells at it. For an agent to learn to play checkers or chess, the natural rewards are $+1$ for winning, -1 for losing, and 0 for drawing and for all nonterminal positions.

You can see what is happening in all of these examples. The agent always learns to maximize its reward. If we want it to do something for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It

is thus critical that the rewards we set up truly indicate what we want accomplished. In particular, the reward signal is not the place to impart to the agent prior knowledge about *how* to achieve what we want it to do.⁵ For example, a chess-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponent’s pieces or gaining control of the center of the board. If achieving these sorts of subgoals were rewarded, then the agent might find a way to achieve them without achieving the real goal. For example, it might find a way to take the opponent’s pieces even at the cost of losing the game. The reward signal is your way of communicating to the agent *what* you want achieved, not *how* you want it achieved.⁶

3.3 Returns and Episodes

So far we have discussed informally the objective of learning. We have said that the agent’s goal is to maximize the cumulative reward it receives in the long run. How might this be defined formally? If the sequence of rewards received after time step t is denoted $R_{t+1}, R_{t+2}, R_{t+3}, \dots$, then what precise aspect of this sequence do we wish to maximize? In general, we seek to maximize the *expected return*, where the return, denoted G_t , is defined as some specific function of the reward sequence. In the simplest case the return is the sum of the rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \quad (3.7)$$

where T is a final time step. This approach makes sense in applications in which there is a natural notion of final time step, that is, when the agent–environment interaction breaks naturally into subsequences, which we call *episodes*,⁷ such as plays of a game, trips through a maze, or any sort of repeated interaction. Each episode ends in a special state called the *terminal state*, followed by a reset to a standard starting state or to a sample from a standard distribution of starting states. Even if you think of episodes as ending in different ways, such as winning and losing a game, the next episode begins independently of how the previous one ended. Thus the episodes can all be considered to end in the same terminal state, with different rewards for the different outcomes. Tasks with episodes of this kind are called *episodic tasks*. In episodic tasks we sometimes need to distinguish the set of all nonterminal states, denoted \mathcal{S} , from the set of all states plus the terminal state, denoted \mathcal{S}^+ . The time of termination, T , is a random variable that normally varies from episode to episode.

On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. For example, this would be the natural way to formulate an on-going process-control task, or an application to a robot with a long life span. We call these *continuing tasks*. The return formulation (3.7) is problematic for continuing tasks because the final time step would be $T = \infty$, and the return, which is what we are trying to maximize, could easily be infinite.

⁵Better places for imparting this kind of prior knowledge are the initial policy or initial value function.

⁶Section 17.4 delves further into the issue of designing effective reward signals.

⁷Episodes are sometimes called “trials” in the literature.

(For example, suppose the agent receives a reward of +1 at each time step.) Thus, in this book we usually use a definition of return that is slightly more complex conceptually but much simpler mathematically.

The additional concept that we need is that of *discounting*. According to this approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses A_t to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad (3.8)$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.

The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the infinite sum in (3.8) has a finite value as long as the reward sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent is “myopic” in being concerned only with maximizing immediate rewards: its objective in this case is to learn how to choose A_t so as to maximize only R_{t+1} . If each of the agent’s actions happened to influence only the immediate reward, not future rewards as well, then a myopic agent could maximize (3.8) by separately maximizing each immediate reward. But in general, acting to maximize immediate reward can reduce access to future rewards so that the return is reduced. As γ approaches 1, the return objective takes future rewards into account more strongly; the agent becomes more farsighted.

Returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \quad (3.9)$$

Note that this works for all time steps $t < T$, even if termination occurs at $t+1$, provided we define $G_T = 0$. This often makes it easy to compute returns from reward sequences.

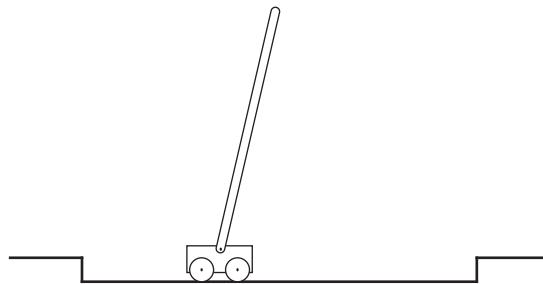
Note that although the return (3.8) is a sum of an infinite number of terms, it is still finite if the reward is nonzero and constant—if $\gamma < 1$. For example, if the reward is a constant +1, then the return is

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}. \quad (3.10)$$

Exercise 3.5 The equations in Section 3.1 are for the continuing case and need to be modified (very slightly) to apply to episodic tasks. Show that you know the modifications needed by giving the modified version of (3.3). \square

Example 3.4: Pole-Balancing

The objective in this task is to apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over: A failure is said to occur if the pole falls past a given angle from vertical or if the cart runs off the track. The pole is reset to vertical after each failure. This task could be treated as episodic, where the natural episodes are the repeated attempts to balance the pole. The reward in this case could be +1 for every time step on which failure did not occur, so that the return at each time would be the number of steps until failure. In this case, successful balancing forever would mean a return of infinity. Alternatively, we could treat pole-balancing as a continuing task, using discounting. In this case the reward would be -1 on each failure and zero at all other times. The return at each time would then be related to $-\gamma^{K-1}$, where K is the number of time steps before failure (as well as to the times of later failures). In either case, the return is maximized by keeping the pole balanced for as long as possible. ■



Exercise 3.6 Suppose you treated pole-balancing as an episodic task but also used discounting, with all rewards zero except for -1 upon failure. What then would the return be at each time? How does this return differ from that in the discounted, continuing formulation of this task? □

Exercise 3.7 Imagine that you are designing a robot to run a maze. You decide to give it a reward of +1 for escaping from the maze and a reward of zero at all other times. The task seems to break down naturally into episodes—the successive runs through the maze—so you decide to treat it as an episodic task, where the goal is to maximize expected total reward (3.7). After running the learning agent for a while, you find that it is showing no improvement in escaping from the maze. What is going wrong? Have you effectively communicated to the agent what you want it to achieve? □

Exercise 3.8 Suppose $\gamma = 0.5$ and the following sequence of rewards is received $R_1 = -1$, $R_2 = 2$, $R_3 = 6$, $R_4 = 3$, and $R_5 = 2$, with $T = 5$. What are G_0 , G_1 , ..., G_5 ? Hint: Work backwards. □

Exercise 3.9 Suppose $\gamma = 0.9$ and the reward sequence is $R_1 = 2$ followed by an infinite sequence of 7s. What are G_1 and G_0 ? □

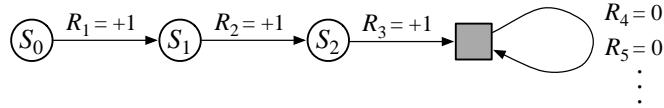
Exercise 3.10 Prove the second equality in (3.10). □

3.4 Unified Notation for Episodic and Continuing Tasks

In the preceding section we described two kinds of reinforcement learning tasks, one in which the agent–environment interaction naturally breaks down into a sequence of separate episodes (episodic tasks), and one in which it does not (continuing tasks). The former case is mathematically easier because each action affects only the finite number of rewards subsequently received during the episode. In this book we consider sometimes one kind of problem and sometimes the other, but often both. It is therefore useful to establish one notation that enables us to talk precisely about both cases simultaneously.

To be precise about episodic tasks requires some additional notation. Rather than one long sequence of time steps, we need to consider a series of episodes, each of which consists of a finite sequence of time steps. We number the time steps of each episode starting anew from zero. Therefore, we have to refer not just to S_t , the state representation at time t , but to $S_{t,i}$, the state representation at time t of episode i (and similarly for $A_{t,i}$, $R_{t,i}$, $\pi_{t,i}$, T_i , etc.). However, it turns out that when we discuss episodic tasks we almost never have to distinguish between different episodes. We are almost always considering a particular episode, or stating something that is true for all episodes. Accordingly, in practice we almost always abuse notation slightly by dropping the explicit reference to episode number. That is, we write S_t to refer to $S_{t,i}$, and so on.

We need one other convention to obtain a single notation that covers both episodic and continuing tasks. We have defined the return as a sum over a finite number of terms in one case (3.7) and as a sum over an infinite number of terms in the other (3.8). These two can be unified by considering episode termination to be the entering of a special *absorbing state* that transitions only to itself and that generates only rewards of zero. For example, consider the state transition diagram:



Here the solid square represents the special absorbing state corresponding to the end of an episode. Starting from S_0 , we get the reward sequence $+1, +1, +1, 0, 0, 0, \dots$. Summing these, we get the same return whether we sum over the first T rewards (here $T = 3$) or over the full infinite sequence. This remains true even if we introduce discounting. Thus, we can define the return, in general, according to (3.8), using the convention of omitting episode numbers when they are not needed, and including the possibility that $\gamma = 1$ if the sum remains defined (e.g., because all episodes terminate). Alternatively, we can write

$$G_t \doteq \sum_{k=t+1}^T \gamma^{k-t-1} R_k, \quad (3.11)$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both). We use these conventions throughout the rest of the book to simplify notation and to express the close parallels

between episodic and continuing tasks. (Later, in Chapter 10, we will introduce a formulation that is both continuing and undiscounted.)

3.5 Policies and Value Functions

Almost all reinforcement learning algorithms involve estimating *value functions*—functions of states (or of state–action pairs) that estimate *how good* it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of “how good” here is defined in terms of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, called policies.

Formally, a *policy* is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t , then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Like p , π is an ordinary function; the “|” in the middle of $\pi(a|s)$ merely reminds us that it defines a probability distribution over $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$. Reinforcement learning methods specify how the agent’s policy is changed as a result of its experience.

Exercise 3.11 If the current state is S_t , and actions are selected according to a stochastic policy π , then what is the expectation of R_{t+1} in terms of π and the four-argument function p (3.2)? \square

The *value function* of a state s under a policy π , denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \text{ for all } s \in \mathcal{S}, \quad (3.12)$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π , and t is any time step. Note that the value of the terminal state, if any, is always zero. We call the function v_π the *state-value function for policy π* .

Similarly, we define the value of taking action a in state s under a policy π , denoted $q_\pi(s, a)$, as the expected return starting from s , taking the action a , and thereafter following policy π :

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right]. \quad (3.13)$$

We call q_π the *action-value function for policy π* .

Exercise 3.12 Give an equation for v_π in terms of q_π and π . \square

Exercise 3.13 Give an equation for q_π in terms of v_π and the four-argument p . \square

The value functions v_π and q_π can be estimated from experience. For example, if an agent follows policy π and maintains an average, for each state encountered, of the actual returns that have followed that state, then the average will converge to the state’s value, $v_\pi(s)$, as the number of times that state is encountered approaches infinity. If separate

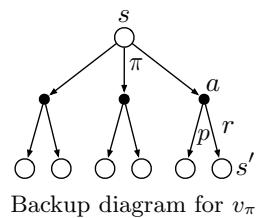
averages are kept for each action taken in each state, then these averages will similarly converge to the action values, $q_\pi(s, a)$. We call estimation methods of this kind *Monte Carlo methods* because they involve averaging over many random samples of actual returns. These kinds of methods are presented in Chapter 5. Of course, if there are very many states, then it may not be practical to keep separate averages for each state individually. Instead, the agent would have to maintain v_π and q_π as parameterized functions (with fewer parameters than states) and adjust the parameters to better match the observed returns. This can also produce accurate estimates, although much depends on the nature of the parameterized function approximator. These possibilities are discussed in Part II of the book.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return (3.9). For any policy π and any state s , the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \tag{by (3.9)} \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')], \quad \text{for all } s \in \mathcal{S}, \end{aligned} \tag{3.14}$$

where it is implicit that the actions, a , are taken from the set $\mathcal{A}(s)$, that the next states, s' , are taken from the set \mathcal{S} (or from \mathcal{S}^+ in the case of an episodic problem), and that the rewards, r , are taken from the set \mathcal{R} . Note also how in the last equation we have merged the two sums, one over all the values of s' and the other over all the values of r , into one sum over all the possible values of both. We use this kind of merged sum often to simplify formulas. Note how the final expression can be read easily as an expected value. It is really a sum over all values of the three variables, a , s' , and r . For each triple, we compute its probability, $\pi(a|s)p(s', r | s, a)$, weight the quantity in brackets by that probability, then sum over all possibilities to get an expected value.

Equation (3.14) is the *Bellman equation* for v_π . It expresses a relationship between the value of a state and the values of its successor states. Think of looking ahead from a state to its possible successor states, as suggested by the diagram to the right. Each open circle represents a state and each solid circle represents a state-action pair. Starting from state s , the root node at the top, the agent could take any of some set of actions—three are shown in the diagram—based on its policy π . From each of these, the environment could respond with one of several next states, s' (two are shown in the figure), along with a reward, r , depending on its dynamics given by the function p . The Bellman equation (3.14) averages over all the possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.



Backup diagram for v_π

The value function v_π is the unique solution to its Bellman equation. We show in subsequent chapters how this Bellman equation forms the basis of a number of ways to compute, approximate, and learn v_π . We call diagrams like that above *backup diagrams* because they diagram relationships that form the basis of the update or *backup* operations that are at the heart of reinforcement learning methods. These operations transfer value information *back* to a state (or a state–action pair) from its successor states (or state–action pairs). We use backup diagrams throughout the book to provide graphical summaries of the algorithms we discuss. (Note that, unlike transition graphs, the state nodes of backup diagrams do not necessarily represent distinct states; for example, a state might be its own successor.)

Example 3.5: Gridworld Figure 3.2 (left) shows a rectangular gridworld representation of a simple finite MDP. The cells of the grid correspond to the states of the environment. At each cell, four actions are possible: **north**, **south**, **east**, and **west**, which deterministically cause the agent to move one cell in the respective direction on the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of -1 . Other actions result in a reward of 0 , except those that move the agent out of the special states **A** and **B**. From state **A**, all four actions yield a reward of $+10$ and take the agent to **A'**. From state **B**, all actions yield a reward of $+5$ and take the agent to **B'**.

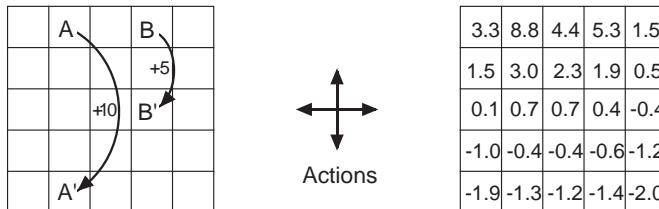


Figure 3.2: Gridworld example: exceptional reward dynamics (left) and state-value function for the equiprobable random policy (right).

Suppose the agent selects all four actions with equal probability in all states. Figure 3.2 (right) shows the value function, v_π , for this policy, for the discounted reward case with $\gamma = 0.9$. This value function was computed by solving the system of linear equations (3.14). Notice the negative values near the lower edge; these are the result of the high probability of hitting the edge of the grid there under the random policy. State **A** is the best state to be in under this policy. Note that **A**'s expected return is *less* than its immediate reward of 10 , because from **A** the agent is taken to state **A'** from which it is likely to run into the edge of the grid. State **B**, on the other hand, is valued *more* than its immediate reward of 5 , because from **B** the agent is taken to **B'** which has a positive value. From **B'** the expected penalty (negative reward) for possibly running into an edge is more than compensated for by the expected gain for possibly stumbling onto **A** or **B**. ■

Exercise 3.14 The Bellman equation (3.14) must hold for each state for the value function v_π shown in Figure 3.2 (right) of Example 3.5. Show numerically that this equation holds for the center state, valued at $+0.7$, with respect to its four neighboring states, valued at $+2.3$, $+0.4$, -0.4 , and $+0.7$. (These numbers are accurate only to one decimal place.) □

Exercise 3.15 In the gridworld example, rewards are positive for goals, negative for running into the edge of the world, and zero the rest of the time. Are the signs of these rewards important, or only the intervals between them? Prove, using (3.8), that adding a constant c to all the rewards adds a constant, v_c , to the values of all states, and thus does not affect the relative values of any states under any policies. What is v_c in terms of c and γ ? \square

Exercise 3.16 Now consider adding a constant c to all the rewards in an episodic task, such as maze running. Would this have any effect, or would it leave the task unchanged as in the continuing task above? Why or why not? Give an example. \square

Example 3.6: Golf To formulate playing a hole of golf as a reinforcement learning task, we count a penalty (negative reward) of -1 for each stroke until we hit the ball into the hole. The state is the location of the ball. The value of a state is the negative of the number of strokes to the hole from that location. Our actions are how we aim and swing at the ball, of course, and which club we select. Let us take the former as given and consider just the choice of club, which we assume is either a putter or a driver. The upper part of Figure 3.3 shows a possible state-value function, $v_{\text{putt}}(s)$, for the policy that always uses the putter. The terminal state *in-the-hole* has a value of 0 . From anywhere on the green we assume we can make a putt; these states have value -1 . Off the green we cannot reach the hole by putting, and the value is lower. If we can reach the green from a state by putting, then that state must have value one less than the green's value, that is, -2 . For simplicity, let us assume we can putt very precisely and deterministically, but with a limited range. This gives us the sharp contour line labeled -2 in the figure; all locations between that line and the green require exactly two strokes to complete the hole. Similarly, any location within putting range of the -2 contour line must have a value of -3 , and so on to get all the contour lines shown in the figure. Putting doesn't get us out of sand traps, so they have a value of $-\infty$. Overall, it takes us six strokes to get from the tee to the hole by putting.

Exercise 3.17 What is the Bellman equation for action values, that is, for q_π ? It must give the action value $q_\pi(s, a)$ in terms of the action values, $q_\pi(s', a')$, of possible successors to the state-action pair (s, a) . Hint: The backup diagram to the right corresponds to this equation. Show the sequence of equations analogous to (3.14), but for action values. \square

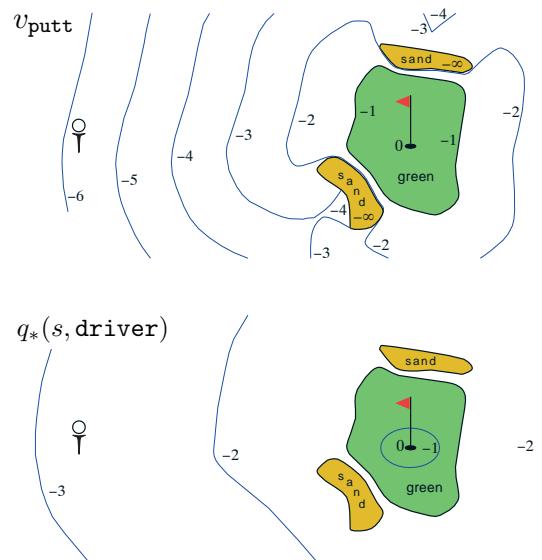
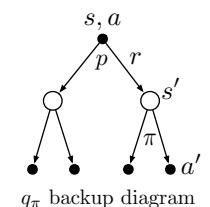
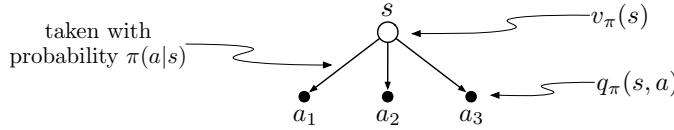


Figure 3.3: A golf example: the state-value function for putting (upper) and the optimal action-value function for using the driver (lower). \blacksquare

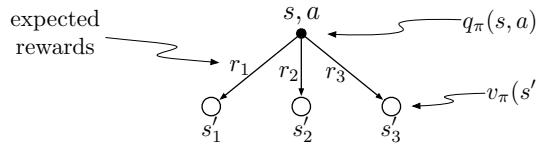


Exercise 3.18 The value of a state depends on the values of the actions possible in that state and on how likely each action is to be taken under the current policy. We can think of this in terms of a small backup diagram rooted at the state and considering each possible action:



Give the equation corresponding to this intuition and diagram for the value at the root node, $v_\pi(s)$, in terms of the value at the expected leaf node, $q_\pi(s, a)$, given $S_t = s$. This equation should include an expectation conditioned on following the policy, π . Then give a second equation in which the expected value is written out explicitly in terms of $\pi(a|s)$ such that no expected value notation appears in the equation. \square

Exercise 3.19 The value of an action, $q_\pi(s, a)$, depends on the expected next reward and the expected sum of the remaining rewards. Again we can think of this in terms of a small backup diagram, this one rooted at an action (state-action pair) and branching to the possible next states:



Give the equation corresponding to this intuition and diagram for the action value, $q_\pi(s, a)$, in terms of the expected next reward, R_{t+1} , and the expected next state value, $v_\pi(S_{t+1})$, given that $S_t = s$ and $A_t = a$. This equation should include an expectation but *not* one conditioned on following the policy. Then give a second equation, writing out the expected value explicitly in terms of $p(s', r | s, a)$ defined by (3.2), such that no expected value notation appears in the equation. \square

3.6 Optimal Policies and Optimal Value Functions

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs, we can precisely define an optimal policy in the following way. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy*. Although there may be more than one, we denote all the optimal policies by π_* . They share the same state-value function, called the *optimal state-value function*, denoted v_* , and defined as

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s), \tag{3.15}$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action-value function*, denoted q_* , and defined as

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a), \quad (3.16)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. For the state-action pair (s, a) , this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Thus, we can write q_* in terms of v_* as follows:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]. \quad (3.17)$$

Example 3.7: Optimal Value Functions for Golf The lower part of Figure 3.3 shows the contours of a possible optimal action-value function $q_*(s, \text{driver})$. These are the values of each state if we first play a stroke with the driver and afterward select either the driver or the putter, whichever is better. The driver enables us to hit the ball farther, but with less accuracy. We can reach the hole in one shot using the driver only if we are already very close; thus the -1 contour for $q_*(s, \text{driver})$ covers only a small portion of the green. If we have two strokes, however, then we can reach the hole from much farther away, as shown by the -2 contour. In this case we don't have to drive all the way to within the small -1 contour, but only to anywhere on the green; from there we can use the putter. The optimal action-value function gives the values after committing to a particular *first* action, in this case, to the driver, but afterward using whichever actions are best. The -3 contour is still farther out and includes the starting tee. From the tee, the best sequence of actions is two drives and one putt, sinking the ball in three strokes. ■

Because v_* is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values (3.14). Because it is the optimal value function, however, v_* 's consistency condition can be written in a special form without reference to any specific policy. This is the Bellman equation for v_* , or the *Bellman optimality equation*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \end{aligned} \quad (\text{by (3.9)})$$

$$\begin{aligned} &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \end{aligned} \quad (3.18)$$

$$= \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_*(s')]. \quad (3.19)$$

The last two equations are two forms of the Bellman optimality equation for v_* . The Bellman optimality equation for q_* is

$$\begin{aligned} q_*(s, a) &= \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \\ &= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right]. \end{aligned} \quad (3.20)$$

The backup diagrams in the figure below show graphically the spans of future states and actions considered in the Bellman optimality equations for v_* and q_* . These are the same as the backup diagrams for v_π and q_π presented earlier except that arcs have been added at the agent's choice points to represent that the maximum over that choice is taken rather than the expected value given some policy. The backup diagram on the left graphically represents the Bellman optimality equation (3.19) and the backup diagram on the right graphically represents (3.20).

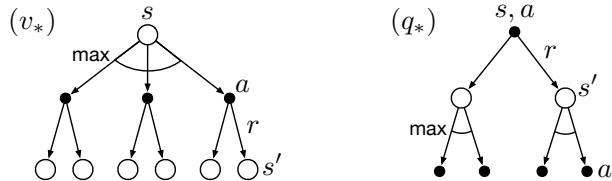


Figure 3.4: Backup diagrams for v_* and q_*

For finite MDPs, the Bellman optimality equation for v_* (3.19) has a unique solution. The Bellman optimality equation is actually a system of equations, one for each state, so if there are n states, then there are n equations in n unknowns. If the dynamics p of the environment are known, then in principle one can solve this system of equations for v_* using any one of a variety of methods for solving systems of nonlinear equations. One can solve a related set of equations for q_* .

Once one has v_* , it is relatively easy to determine an optimal policy. For each state s , there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy. You can think of this as a one-step search. If you have the optimal value function, v_* , then the actions that appear best after a one-step search will be optimal actions. Another way of saying this is that any policy that is *greedy* with respect to the optimal evaluation function v_* is an optimal policy. The term *greedy* is used in computer science to describe any search or decision procedure that selects alternatives based only on local or immediate considerations, without considering the possibility that such a selection may prevent future access to even better alternatives. Consequently, it describes policies that select actions based only on their short-term consequences. The beauty of v_* is that if one uses it to evaluate the short-term consequences of actions—specifically, the one-step consequences—then a greedy policy is actually optimal in the long-term sense in which we are interested because v_* already takes into account the reward consequences of all possible future behavior. By means of v_* , the optimal expected long-term return is

turned into a quantity that is locally and immediately available for each state. Hence, a one-step-ahead search yields the long-term optimal actions.

Having q_* makes choosing optimal actions even easier. With q_* , the agent does not even have to do a one-step-ahead search: for any state s , it can simply find any action that maximizes $q_*(s, a)$. The action-value function effectively caches the results of all one-step-ahead searches. It provides the optimal expected long-term return as a value that is locally and immediately available for each state-action pair. Hence, at the cost of representing a function of state-action pairs, instead of just of states, the optimal action-value function allows optimal actions to be selected without having to know anything about possible successor states and their values, that is, without having to know anything about the environment's dynamics.

Example 3.8: Solving the Gridworld Suppose we solve the Bellman equation for v_* for the simple grid task introduced in Example 3.5 and shown again in Figure 3.5 (left). Recall that state A is followed by a reward of +10 and transition to state A', while state B is followed by a reward of +5 and transition to state B'. Figure 3.5 (middle) shows the optimal value function, and Figure 3.5 (right) shows the corresponding optimal policies. Where there are multiple arrows in a cell, all of the corresponding actions are optimal.

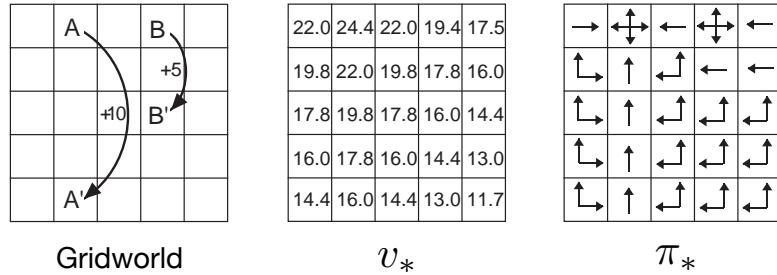


Figure 3.5: Optimal solutions to the gridworld example. ■

Example 3.9: Bellman Optimality Equations for the Recycling Robot Using (3.19), we can explicitly give the Bellman optimality equation for the recycling robot example. To make things more compact, we abbreviate the states `high` and `low`, and the actions `search`, `wait`, and `recharge` respectively by h , l , s , w , and re . Because there are only two states, the Bellman optimality equation consists of two equations. The equation for $v_*(h)$ can be written as follows:

$$\begin{aligned}
 v_*(h) &= \max \left\{ \begin{array}{l} p(h|h, s)[r(h, s, h) + \gamma v_*(h)] + p(l|h, s)[r(h, s, l) + \gamma v_*(l)], \\ p(h|w, h)[r(h, w, h) + \gamma v_*(h)] + p(l|w, h)[r(h, w, l) + \gamma v_*(l)] \end{array} \right\} \\
 &= \max \left\{ \begin{array}{l} \alpha[r_s + \gamma v_*(h)] + (1 - \alpha)[r_s + \gamma v_*(l)], \\ 1[r_w + \gamma v_*(h)] + 0[r_w + \gamma v_*(l)] \end{array} \right\} \\
 &= \max \left\{ \begin{array}{l} r_s + \gamma[\alpha v_*(h) + (1 - \alpha)v_*(l)], \\ r_w + \gamma v_*(h) \end{array} \right\}.
 \end{aligned}$$

Following the same procedure for $v_*(1)$ yields the equation

$$v_*(1) = \max \left\{ \begin{array}{l} \beta r_s - 3(1-\beta) + \gamma[(1-\beta)v_*(h) + \beta v_*(1)], \\ r_w + \gamma v_*(1), \\ \gamma v_*(h) \end{array} \right\}.$$

For any choice of r_s , r_w , α , β , and γ , with $0 \leq \gamma < 1$, $0 \leq \alpha, \beta \leq 1$, there is exactly one pair of numbers, $v_*(h)$ and $v_*(1)$, that simultaneously satisfy these two nonlinear equations. ■

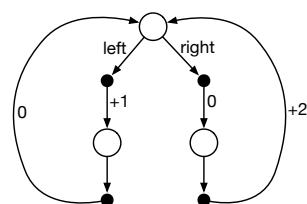
Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the reinforcement learning problem. However, this solution is rarely directly useful. It is akin to an exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. This solution relies on at least three assumptions that are rarely true in practice: (1) the dynamics of the environment are accurately known; (2) computational resources are sufficient to complete the calculation; and (3) the states have the Markov property. For the kinds of tasks in which we are interested, one is generally not able to implement this solution exactly because various combinations of these assumptions are violated. For example, although the first and third assumptions present no problems for the game of backgammon, the second is a major impediment. Because the game has about 10^{20} states, it would take thousands of years on today's fastest computers to solve the Bellman equation for v_* , and the same is true for finding q_* . In reinforcement learning one typically has to settle for approximate solutions.

Many different decision-making methods can be viewed as ways of approximately solving the Bellman optimality equation. For example, heuristic search methods can be viewed as expanding the right-hand side of (3.19) several times, up to some depth, forming a “tree” of possibilities, and then using a heuristic evaluation function to approximate v_* at the “leaf” nodes. (Heuristic search methods such as A* are almost always based on the episodic case.) The methods of dynamic programming can be related even more closely to the Bellman optimality equation. Many reinforcement learning methods can be clearly understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions. We consider a variety of such methods in the following chapters.

Exercise 3.20 Draw or describe the optimal state-value function for the golf example. □

Exercise 3.21 Draw or describe the contours of the optimal action-value function for putting, $q_*(s, \text{putter})$, for the golf example. □

Exercise 3.22 Consider the continuing MDP shown to the right. The only decision to be made is that in the top state, where two actions are available, left and right. The numbers show the rewards that are received deterministically after each action. There are exactly two deterministic policies, π_{left} and π_{right} . What policy is optimal if $\gamma = 0$? If $\gamma = 0.9$? If $\gamma = 0.5$? □



Exercise 3.23 Give the Bellman equation for q_* for the recycling robot. □

Exercise 3.24 Figure 3.5 gives the optimal value of the best state of the gridworld as 24.4, to one decimal place. Use your knowledge of the optimal policy and (3.8) to express this value symbolically, and then to compute it to three decimal places. □

Exercise 3.25 Give an equation for v_* in terms of q_* . □

Exercise 3.26 Give an equation for q_* in terms of v_* and the four-argument p . □

Exercise 3.27 Give an equation for π_* in terms of q_* . □

Exercise 3.28 Give an equation for π_* in terms of v_* and the four-argument p . □

Exercise 3.29 Rewrite the four Bellman equations for the four value functions (v_π , v_* , q_π , and q_*) in terms of the three argument function p (3.4) and the two-argument function r (3.5). □

3.7 Optimality and Approximation

We have defined optimal value functions and optimal policies. Clearly, an agent that learns an optimal policy has done very well, but in practice this rarely happens. For the kinds of tasks in which we are interested, optimal policies can be generated only with extreme computational cost. A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that agents can only approximate. As we discussed above, even if we have a complete and accurate model of the environment's dynamics, it is usually not possible to simply compute an optimal policy by solving the Bellman optimality equation. For example, board games such as chess are a tiny fraction of human experience, yet large, custom-designed computers still cannot compute the optimal moves. A critical aspect of the problem facing the agent is always the computational power available to it, in particular, the amount of computation it can perform in a single time step.

The memory available is also an important constraint. A large amount of memory is often required to build up approximations of value functions, policies, and models. In tasks with small, finite state sets, it is possible to form these approximations using arrays or tables with one entry for each state (or state-action pair). This we call the *tabular* case, and the corresponding methods we call tabular methods. In many cases of practical interest, however, there are far more states than could possibly be entries in a table. In these cases the functions must be approximated, using some sort of more compact parameterized function representation.

Our framing of the reinforcement learning problem forces us to settle for approximations. However, it also presents us with some unique opportunities for achieving useful approximations. For example, in approximating optimal behavior, there may be many states that the agent faces with such a low probability that selecting suboptimal actions for them has little impact on the amount of reward the agent receives. Tesauro's backgammon player, for example, plays with exceptional skill even though it might make

very bad decisions on board configurations that never occur in games against experts. In fact, it is possible that TD-Gammon makes bad decisions for a large fraction of the game's state set. The online nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states. This is one key property that distinguishes reinforcement learning from other approaches to approximately solving MDPs.

3.8 Summary

Let us summarize the elements of the reinforcement learning problem that we have presented in this chapter. Reinforcement learning is about learning from interaction how to behave in order to achieve a goal. The reinforcement learning *agent* and its *environment* interact over a sequence of discrete time steps. The specification of their interface defines a particular task: the *actions* are the choices made by the agent; the *states* are the basis for making the choices; and the *rewards* are the basis for evaluating the choices. Everything inside the agent is known and controllable. Its environment, on the other hand, is incompletely controllable and may or may not be completely known. A *policy* is a stochastic rule by which the agent selects actions as a function of states. The agent's objective is to maximize the amount of reward it receives over time.

When the reinforcement learning setup described above is formulated with well defined transition probabilities it constitutes a *Markov decision process* (MDP). A *finite MDP* is an MDP with finite state, action, and (as we formulate it here) reward sets. Much of the current theory of reinforcement learning is restricted to finite MDPs, but the methods and ideas apply more generally.

The *return* is the function of future rewards that the agent seeks to maximize (in expected value). It has several different definitions depending upon the nature of the task and whether one wishes to *discount* delayed reward. The undiscounted formulation is appropriate for *episodic tasks*, in which the agent–environment interaction breaks naturally into *episodes*; the discounted formulation is appropriate for tabular *continuing tasks*, in which the interaction does not naturally break into episodes but continues without limit (but see Sections 10.3–4). We try to define the returns for the two kinds of tasks such that one set of equations can apply to both the episodic and continuing cases.

A policy's *value functions* (v_π and q_π) assign to each state, or state–action pair, the expected return from that state, or state–action pair, given that the agent uses the policy. The *optimal value functions* (v_* and q_*) assign to each state, or state–action pair, the largest expected return achievable by any policy. A policy whose value functions are optimal is an *optimal policy*. Whereas the optimal value functions for states and state–action pairs are unique for a given MDP, there can be many optimal policies. Any policy that is *greedy* with respect to the optimal value functions must be an optimal policy. The *Bellman optimality equations* are special consistency conditions that the optimal value functions must satisfy and that can, in principle, be solved for the optimal value functions, from which an optimal policy can be determined with relative ease.

A reinforcement learning problem can be posed in a variety of different ways depending on assumptions about the level of knowledge initially available to the agent. In problems of *complete knowledge*, the agent has a complete and accurate model of the environment's dynamics. If the environment is an MDP, then such a model consists of the complete four-argument dynamics function p (3.2). In problems of *incomplete knowledge*, a complete and perfect model of the environment is not available.

Even if the agent had a complete and accurate environment model, the agent would typically be unable to fully use it because of limitations on its memory and computation per time step. In particular, extensive memory may be required to build up accurate approximations of value functions, policies, and models. In most cases of practical interest there are far more states than could possibly be entries in a table, and approximations must be made.

A well-defined notion of optimality organizes the approach to learning we describe in this book and provides a way to understand the theoretical properties of various learning algorithms, but it is an ideal that reinforcement learning agents can only approximate to varying degrees. In reinforcement learning we are very much concerned with cases in which optimal solutions cannot be found but must be approximated in some way.

Bibliographical and Historical Remarks

The reinforcement learning problem is deeply indebted to the idea of Markov decision processes (MDPs) from the field of optimal control. These historical influences and other major influences from psychology are described in the brief history given in Chapter 1. Reinforcement learning adds to MDPs a focus on approximation and incomplete information for realistically large problems. MDPs and the reinforcement learning problem are only weakly linked to traditional learning and decision-making problems in artificial intelligence. However, artificial intelligence is now vigorously exploring MDP formulations for planning and decision making from a variety of perspectives. MDPs are more general than previous formulations used in artificial intelligence in that they permit more general kinds of goals and uncertainty.

The theory of MDPs is treated by, for example, Bertsekas (2005), White (1969), Whittle (1982, 1983), and Puterman (1994). A particularly compact treatment of the finite case is given by Ross (1983). MDPs are also studied under the heading of stochastic optimal control, where *adaptive* optimal control methods are most closely related to reinforcement learning (e.g., Kumar, 1985; Kumar and Varaiya, 1986).

The theory of MDPs evolved from efforts to understand the problem of making sequences of decisions under uncertainty, where each decision can depend on the previous decisions and their outcomes. It is sometimes called the theory of multistage decision processes, or sequential decision processes, and has roots in the statistical literature on sequential sampling beginning with the papers by Thompson (1933, 1934) and Robbins (1952) that we cited in Chapter 2 in connection with bandit problems (which are prototypical MDPs if formulated as multiple-situation problems).

The earliest instance (that we are aware of) in which reinforcement learning was discussed using the MDP formalism is Andreae's (1969) description of a unified view of

learning machines. Witten and Corbin (1973) experimented with a reinforcement learning system later analyzed by Witten (1977, 1976a) using the MDP formalism. Although he did not explicitly mention MDPs, Werbos (1977) suggested approximate solution methods for stochastic optimal control problems that are related to modern reinforcement learning methods (see also Werbos, 1982, 1987, 1988, 1989, 1992). Although Werbos's ideas were not widely recognized at the time, they were prescient in emphasizing the importance of approximately solving optimal control problems in a variety of domains, including artificial intelligence. The most influential integration of reinforcement learning and MDPs is due to Watkins (1989).

- 3.1** Our characterization of the dynamics of an MDP in terms of $p(s', r|s, a)$ is slightly unusual. It is more common in the MDP literature to describe the dynamics in terms of the state transition probabilities $p(s'|s, a)$ and expected next rewards $r(s, a)$. In reinforcement learning, however, we more often have to refer to individual actual or sample rewards (rather than just their expected values). Our notation also makes it plainer that S_t and R_t are in general jointly determined, and thus must have the same time index. In teaching reinforcement learning, we have found our notation to be more straightforward conceptually and easier to understand.

For a good intuitive discussion of the system-theoretic concept of state, see Minsky (1967).

The bioreactor example is based on the work of Ungar (1990) and Miller and Williams (1992). The recycling robot example was inspired by the can-collecting robot built by Jonathan Connell (1989). Kober and Peters (2012) present a collection of robotics applications of reinforcement learning.

- 3.2** An explicit statement of the reward hypothesis was suggested by Michael Littman (personal communication).

- 3.3–4** The terminology of *episodic* and *continuing* tasks is different from that usually used in the MDP literature. In that literature it is common to distinguish three types of tasks: (1) finite-horizon tasks, in which interaction terminates after a particular *fixed* number of time steps; (2) indefinite-horizon tasks, in which interaction can last arbitrarily long but must eventually terminate; and (3) infinite-horizon tasks, in which interaction does not terminate. Our episodic and continuing tasks are similar to indefinite-horizon and infinite-horizon tasks, respectively, but we prefer to emphasize the difference in the nature of the interaction. This difference seems more fundamental than the difference in the objective functions emphasized by the usual terms. Often episodic tasks use an indefinite-horizon objective function and continuing tasks an infinite-horizon objective function, but we see this as a common coincidence rather than a fundamental difference.

The pole-balancing example is from Michie and Chambers (1968) and Barto, Sutton, and Anderson (1983).

3.5–6 Assigning value on the basis of what is good or bad in the long run has ancient roots. In control theory, mapping states to numerical values representing the long-term consequences of control decisions is a key part of optimal control theory, which was developed in the 1950s by extending nineteenth century state-function theories of classical mechanics (see, for example, Schultz and Melsa, 1967). In describing how a computer could be programmed to play chess, Shannon (1950) suggested using an evaluation function that took into account the long-term advantages and disadvantages of chess positions.

Watkins's (1989) Q-learning algorithm for estimating q_* (Chapter 6) made action-value functions an important part of reinforcement learning, and consequently these functions are often called “Q-functions.” But the idea of an action-value function is much older than this. Shannon (1950) suggested that a function $h(P, M)$ could be used by a chess-playing program to decide whether a move M in position P is worth exploring. Michie's (1961, 1963) MENACE system and Michie and Chambers's (1968) BOXES system can be understood as estimating action-value functions. In classical physics, Hamilton's principal function is an action-value function; Newtonian dynamics are greedy with respect to this function (e.g., Goldstein, 1957). Action-value functions also played a central role in Denardo's (1967) theoretical treatment of dynamic programming in terms of contraction mappings.

The Bellman optimality equation (for v_*) was popularized by Richard Bellman (1957a), who called it the “basic functional equation.” The counterpart of the Bellman optimality equation for continuous time and state problems is known as the Hamilton–Jacobi–Bellman equation (or often just the Hamilton–Jacobi equation), indicating its roots in classical physics (e.g., Schultz and Melsa, 1967).

The golf example was suggested by Chris Watkins.

Chapter 4

Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. DP provides an essential foundation for the understanding of the methods presented in the rest of this book. In fact, all of these methods can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

Starting with this chapter, we usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets, \mathcal{S} , \mathcal{A} , and \mathcal{R} , are finite, and that its dynamics are given by a set of probabilities $p(s', r | s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $r \in \mathcal{R}$, and $s' \in \mathcal{S}^+$ (\mathcal{S}^+ is \mathcal{S} plus a terminal state if the problem is episodic). Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods. The methods we explore in Part II are applicable to continuous problems and are a significant extension of that approach.

The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies. In this chapter we show how DP can be used to compute the value functions defined in Chapter 3. As discussed there, we can easily obtain optimal policies once we have found the optimal value functions, v_* or q_* , which satisfy the Bellman optimality equations:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v_*(s')], \end{aligned} \tag{4.1}$$

or

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s', r} p(s', r | s, a) [r + \gamma \max_{a'} q_*(s', a')], \end{aligned} \tag{4.2}$$

for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $s' \in \mathcal{S}^+$. As we shall see, DP algorithms are obtained by turning Bellman equations such as these into assignments, that is, into update rules for improving approximations of the desired value functions.

4.1 Policy Evaluation (Prediction)

First we consider how to compute the state-value function v_π for an arbitrary policy π . This is called *policy evaluation* in the DP literature. We also refer to it as the *prediction problem*. Recall from Chapter 3 that, for all $s \in \mathcal{S}$,

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \end{aligned} \tag{from (3.9)}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \tag{4.3}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')], \tag{4.4}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π , and the expectations are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π .

If the environment's dynamics are completely known, then (4.4) is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns (the $v_\pi(s)$, $s \in \mathcal{S}$). In principle, its solution is a straightforward, if tedious, computation. For our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions v_0, v_1, v_2, \dots , each mapping \mathcal{S}^+ to \mathbb{R} (the real numbers). The initial approximation, v_0 , is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π (4.4) as an update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')], \end{aligned} \tag{4.5}$$

for all $s \in \mathcal{S}$. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \rightarrow \infty$ under the same conditions that guarantee the existence of v_π . This algorithm is called *iterative policy evaluation*.

To produce each successive approximation, v_{k+1} from v_k , iterative policy evaluation applies the same operation to each state s : it replaces the old value of s with a new value obtained from the old values of the successor states of s , and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. We call this kind of operation an *expected update*. Each iteration of iterative policy evaluation updates the value of every state once to produce the new approximate value function

v_{k+1} . There are several different kinds of expected updates, depending on whether a state (as here) or a state-action pair is being updated, and depending on the precise way the estimated values of the successor states are combined. All the updates done in DP algorithms are called *expected* updates because they are based on an expectation over all possible next states rather than on a sample next state. The nature of an update can be expressed in an equation, as above, or in a backup diagram like those introduced in Chapter 3. For example, the backup diagram corresponding to the expected update used in iterative policy evaluation is shown on page 59.

To write a sequential computer program to implement iterative policy evaluation as given by (4.5) you would have to use two arrays, one for the old values, $v_k(s)$, and one for the new values, $v_{k+1}(s)$. With two arrays, the new values can be computed one by one from the old values without the old values being changed. Alternatively, you could use one array and update the values “in place,” that is, with each new value immediately overwriting the old one. Then, depending on the order in which the states are updated, sometimes new values are used instead of old ones on the right-hand side of (4.5). This in-place algorithm also converges to v_π ; in fact, it usually converges faster than the two-array version, as you might expect, because it uses new data as soon as they are available. We think of the updates as being done in a *sweep* through the state space. For the in-place algorithm, the order in which states have their values updated during the sweep has a significant influence on the rate of convergence. We usually have the in-place version in mind when we think of DP algorithms.

A complete in-place version of iterative policy evaluation is shown in pseudocode in the box below. Note how it handles termination. Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this. The pseudocode tests the quantity $\max_{s \in \mathcal{S}} |v_{k+1}(s) - v_k(s)|$ after each sweep and stops when it is sufficiently small.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π , the policy to be evaluated

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$ arbitrarily, for $s \in \mathcal{S}$, and $V(\text{terminal})$ to 0

Loop:

$$\Delta \leftarrow 0$$

Loop for each $s \in \mathcal{S}$:

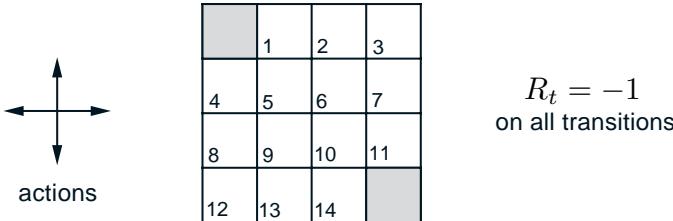
$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$

Example 4.1 Consider the 4×4 gridworld shown below.



The nonterminal states are $\mathcal{S} = \{1, 2, \dots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\text{up}, \text{down}, \text{right}, \text{left}\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. Thus, for instance, $p(6, -1|5, \text{right}) = 1$, $p(7, -1|7, \text{right}) = 1$, and $p(10, r|5, \text{right}) = 0$ for all $r \in \mathcal{R}$. This is an undiscounted, episodic task. The reward is -1 on all transitions until the terminal state is reached. The terminal state is shaded in the figure (although it is shown in two places, it is formally one state). The expected reward function is thus $r(s, a, s') = -1$ for all states s, s' and actions a . Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 4.1 shows the sequence of value functions $\{v_k\}$ computed by iterative policy evaluation. The final estimate is in fact v_π , which in this case gives for each state the negation of the expected number of steps from that state until termination. ■

Exercise 4.1 In Example 4.1, if π is the equiprobable random policy, what is $q_\pi(11, \text{down})$? What is $q_\pi(7, \text{down})$? □

Exercise 4.2 In Example 4.1, suppose a new state 15 is added to the gridworld just below state 13, and its actions, `left`, `up`, `right`, and `down`, take the agent to states 12, 13, 14, and 15, respectively. Assume that the transitions *from* the original states are unchanged. What, then, is $v_\pi(15)$ for the equiprobable random policy? Now suppose the dynamics of state 13 are also changed, such that action `down` from state 13 takes the agent to the new state 15. What is $v_\pi(15)$ for the equiprobable random policy in this case? □

Exercise 4.3 What are the equations analogous to (4.3), (4.4), and (4.5), but for *action-value* functions instead of state-value functions? □

4.2 Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. Suppose we have determined the value function v_π for an arbitrary deterministic policy π . For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. We know how good it is to follow the current policy from s —that is $v_\pi(s)$ —but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting a in s and thereafter

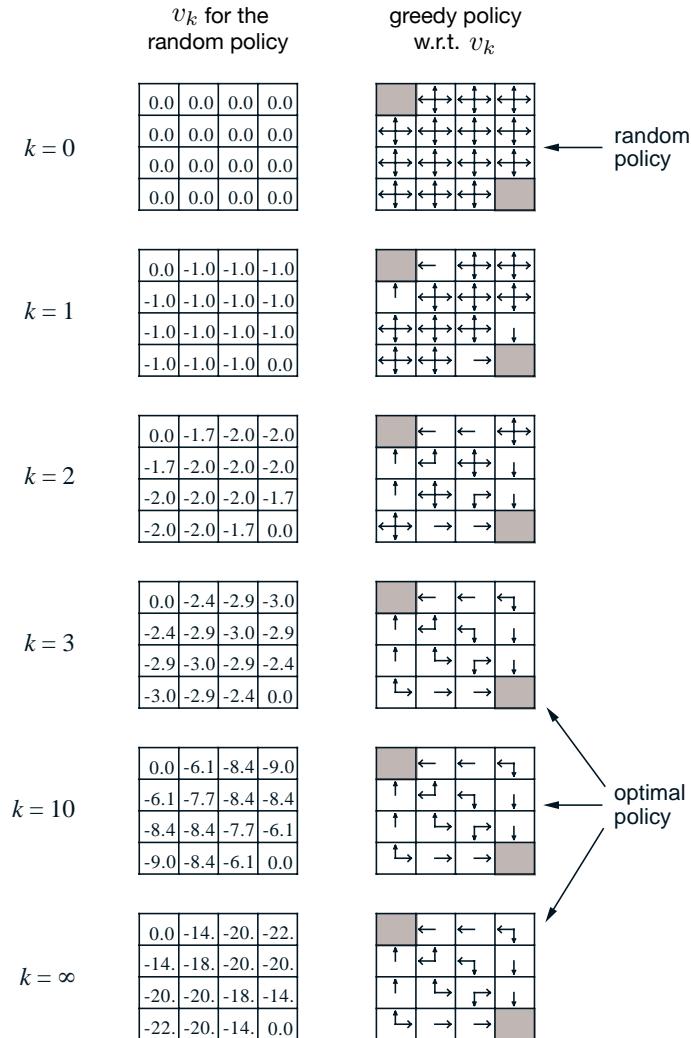


Figure 4.1: Convergence of iterative policy evaluation on a small gridworld. The left column is the sequence of approximations of the state-value function for the random policy (all actions equally likely). The right column is the sequence of greedy policies corresponding to the value function estimates (arrows are shown for all actions achieving the maximum, and the numbers shown are rounded to two significant digits). The last policy is guaranteed only to be an improvement over the random policy, but in this case it, and all policies after the third iteration, are optimal.

following the existing policy, π . The value of this way of behaving is

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_\pi(s')]. \end{aligned} \tag{4.6}$$

The key criterion is whether this is greater than or less than $v_\pi(s)$. If it is greater—that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time—then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.

That this is true is a special case of a general result called the *policy improvement theorem*. Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s). \tag{4.7}$$

Then the policy π' must be as good as, or better than, π . That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s). \tag{4.8}$$

Moreover, if there is strict inequality of (4.7) at any state, then there must be strict inequality of (4.8) at that state.

The policy improvement theorem applies to the two policies that we considered at the beginning of this section: an original deterministic policy, π , and a changed policy, π' , that is identical to π except that $\pi'(s) = a \neq \pi(s)$. For states other than s , (4.7) holds because the two sides are equal. Thus, if $q_\pi(s, a) > v_\pi(s)$, then the changed policy is indeed better than π .

The idea behind the proof of the policy improvement theorem is easy to understand. Starting from (4.7), we keep expanding the q_π side with (4.6) and reapplying (4.7) until we get $v_{\pi'}(s)$:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\ &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)] \quad (\text{by (4.6)}) \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] \quad (\text{by (4.7)}) \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_{t+1}, A_{t+1} = \pi'(S_{t+1})] \mid S_t = s] \\ &= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) \mid S_t = s] \\ &\vdots \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \mid S_t = s] \\ &= v_{\pi'}(s). \end{aligned}$$

So far we have seen how, given a policy and its value function, we can easily evaluate a change in the policy at a single state. It is a natural extension to consider changes at

all states, selecting at each state the action that appears best according to $q_\pi(s, a)$. In other words, to consider the new *greedy* policy, π' , given by

$$\begin{aligned}\pi'(s) &\doteq \arg \max_a q_\pi(s, a) \\ &= \arg \max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \arg \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_\pi(s')],\end{aligned}\tag{4.9}$$

where $\arg \max_a$ denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). The greedy policy takes the action that looks best in the short term—after one step of lookahead—according to v_π . By construction, the greedy policy meets the conditions of the policy improvement theorem (4.7), so we know that it is as good as, or better than, the original policy. The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called *policy improvement*.

Suppose the new greedy policy, π' , is as good as, but not better than, the old policy π . Then $v_\pi = v_{\pi'}$, and from (4.9) it follows that for all $s \in \mathcal{S}$:

$$\begin{aligned}v_{\pi'}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_{\pi'}(s')].\end{aligned}$$

But this is the same as the Bellman optimality equation (4.1), and therefore, $v_{\pi'}$ must be v_* , and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

So far in this section we have considered the special case of deterministic policies. In the general case, a stochastic policy π specifies probabilities, $\pi(a|s)$, for taking each action, a , in each state, s . We will not go through the details, but in fact all the ideas of this section extend easily to stochastic policies. In particular, the policy improvement theorem carries through as stated for the stochastic case. In addition, if there are ties in policy improvement steps such as (4.9)—that is, if there are several actions at which the maximum is achieved—then in the stochastic case we need not select a single action from among them. Instead, each maximizing action can be given a portion of the probability of being selected in the new greedy policy. Any apportioning scheme is allowed as long as all submaximal actions are given zero probability.

The last row of Figure 4.1 shows an example of policy improvement for stochastic policies. Here the original policy, π , is the equiprobable random policy, and the new policy, π' , is greedy with respect to v_π . The value function v_π is shown in the bottom-left diagram and the set of possible π' is shown in the bottom-right diagram. The states with multiple arrows in the π' diagram are those in which several actions achieve the maximum in (4.9); any apportionment of probability among these actions is permitted. For any such policy, its state values $v_{\pi'}(s)$ can be seen by inspection to be either -1 , -2 , or -3 , for all states $s \in \mathcal{S}$, whereas $v_\pi(s)$ is at most -14 . Thus, $v_{\pi'}(s) \geq v_\pi(s)$, for all

$s \in \mathcal{S}$, illustrating policy improvement. Although in this case the new policy π' happens to be optimal, in general only an improvement is guaranteed.

4.3 Policy Iteration

Once a policy, π , has been improved using v_π to yield a better policy, π' , we can then compute $v_{\pi'}$ and improve it again to yield an even better π'' . We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

where $\xrightarrow{\text{E}}$ denotes a policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a policy *improvement*. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of deterministic policies, this process must converge to an optimal policy and the optimal value function in a finite number of iterations.

This way of finding an optimal policy is called *policy iteration*. A complete algorithm is given in the box below. Note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy. This typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next).

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(\text{terminal}) \doteq 0$

2. Policy Evaluation

Loop:

$$\Delta \leftarrow 0$$

Loop for each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_{s',r} p(s', r | s, \pi(s)) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement

$\text{policy-stable} \leftarrow \text{true}$

For each $s \in \mathcal{S}$:

$$\text{old-action} \leftarrow \pi(s)$$

$$\pi(s) \leftarrow \arg \max_a \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

If $\text{old-action} \neq \pi(s)$, then $\text{policy-stable} \leftarrow \text{false}$

If policy-stable , then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Example 4.2: Jack’s Car Rental Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is $\frac{\lambda^n}{n!} e^{-\lambda}$, where λ is the expected number. Suppose λ is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight. Figure 4.2 shows the sequence of policies found by policy iteration starting from the policy that never moves any cars.

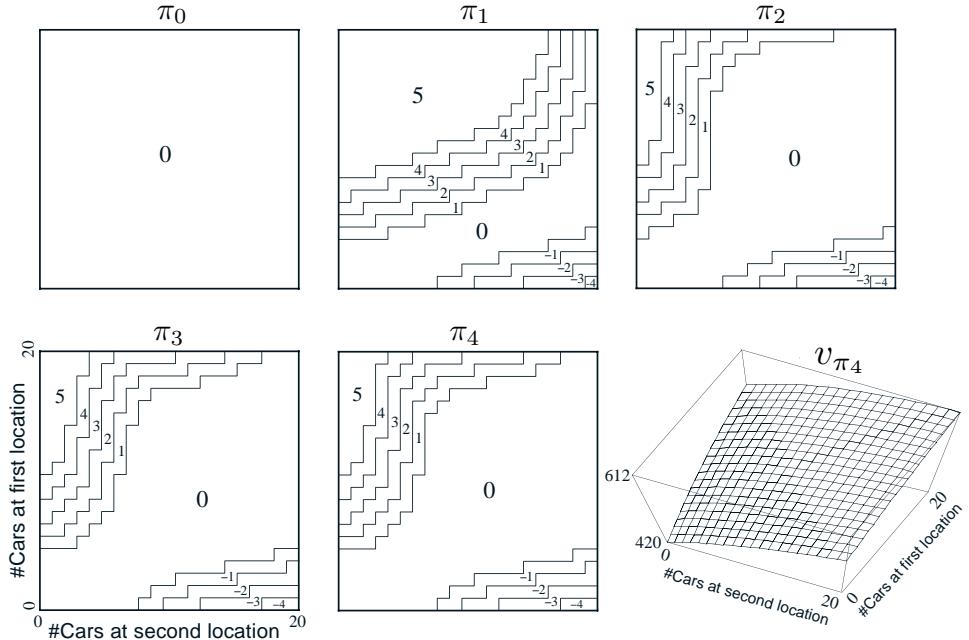


Figure 4.2: The sequence of policies found by policy iteration on Jack’s car rental problem, and the final state-value function. The first five diagrams show, for each number of cars at each location at the end of the day, the number of cars to be moved from the first location to the second (negative numbers indicate transfers from the second location to the first). Each successive policy is a strict improvement over the previous policy, and the last policy is optimal. ■

Policy iteration often converges in surprisingly few iterations, as illustrated in the example of Jack’s car rental and in the example in Figure 4.1. The bottom-left diagram of Figure 4.1 shows the value function for the equiprobable random policy, and the bottom-right diagram shows a greedy policy for this value function. The policy improvement theorem assures us that these policies are better than the original random policy. In this case, however, these policies are not just better, but optimal, proceeding to the terminal states in the minimum number of steps. In this example, policy iteration would find the optimal policy after just one iteration.

Exercise 4.4 The policy iteration algorithm on page 80 has a subtle bug in that it may never terminate if the policy continually switches between two or more policies that are equally good. This is okay for pedagogy, but not for actual use. Modify the pseudocode so that convergence is guaranteed. \square

Exercise 4.5 How would policy iteration be defined for action values? Give a complete algorithm for computing q_* , analogous to that on page 80 for computing v_* . Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book. \square

Exercise 4.6 Suppose you are restricted to considering only policies that are ε -soft, meaning that the probability of selecting each action in each state, s , is at least $\varepsilon/|\mathcal{A}(s)|$. Describe qualitatively the changes that would be required in each of the steps 3, 2, and 1, in that order, of the policy iteration algorithm for v_* on page 80. \square

Exercise 4.7 (programming) Write a program for policy iteration and re-solve Jack’s car rental problem with the following changes. One of Jack’s employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs \$2, as do all cars moved in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of \$4 must be incurred to use a second parking lot (independent of how many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming. To check your program, first replicate the results given for the original problem. \square

4.4 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set. If policy evaluation is done iteratively, then convergence exactly to v_π occurs only in the limit. Must we wait for exact convergence, or can we stop short of that? The example in Figure 4.1 certainly suggests that it may be possible to truncate policy evaluation. In that example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration. One important special

case is when policy evaluation is stopped after just one sweep (one update of each state). This algorithm is called *value iteration*. It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')], \end{aligned} \quad (4.10)$$

for all $s \in \mathcal{S}$. For arbitrary v_0 , the sequence $\{v_k\}$ can be shown to converge to v_* under the same conditions that guarantee the existence of v_* .

Another way of understanding value iteration is by reference to the Bellman optimality equation (4.1). Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule. Also note how the value iteration update is identical to the policy evaluation update (4.5) except that it requires the maximum to be taken over all actions. Another way of seeing this close relationship is to compare the backup diagrams for these algorithms on page 59 (policy evaluation) and on the left of Figure 3.4 (value iteration). These two are the natural backup operations for computing v_π and v_* .

Finally, let us consider how value iteration terminates. Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_* . In practice, we stop once the value function changes by only a small amount in a sweep. The box below shows a complete algorithm with this kind of termination condition.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

```

    Δ ← 0
    Loop for each  $s \in \mathcal{S}$ :
         $v \leftarrow V(s)$ 
         $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$ 
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
    until  $\Delta < \theta$ 

```

Output a deterministic policy, $\pi \approx \pi_*$, such that

$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates. Because the max operation in (4.10) is the only difference between

these updates, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs.

Example 4.3: Gambler's Problem A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake. The game ends when the gambler wins by reaching his goal of \$100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer numbers of dollars. This problem can be formulated as an undiscounted, episodic, finite MDP. The state is the gambler's capital, $s \in \{1, 2, \dots, 99\}$ and the actions are stakes, $a \in \{0, 1, \dots, \min(s, 100 - s)\}$. The reward is zero on all transitions except those on which the gambler reaches his goal, when it is +1. The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal. Let p_h denote the probability of the coin coming up heads. If p_h is known, then the entire problem is known and it can be solved, for instance, by value iteration. Figure 4.3 shows the change in the value function over successive sweeps of value iteration, and the final policy found, for the case of $p_h = 0.4$. This policy is optimal, but not unique. In fact, there is a whole family of optimal policies, all corresponding to ties for the argmax action selection with respect to the optimal value function. Can you guess what the entire family looks like?

Exercise 4.8 Why does the optimal policy for the gambler's problem have such a curious form? In particular, for capital of 50 it bets it all on one flip, but for capital of 51 it does not. Why is this a good policy? \square

Exercise 4.9 (programming) Implement value iteration for the gambler's problem and solve it for $p_h = 0.25$ and $p_h = 0.55$. In programming, you may find it convenient to introduce two dummy states corresponding to termination with capital of 0 and 100, giving them values of 0 and 1 respectively. Show your results graphically, as in Figure 4.3. Are your results stable as $\theta \rightarrow 0$? \square

Exercise 4.10 What is the analog of the value iteration update (4.10) for action values, $q_{k+1}(s, a)$? \square

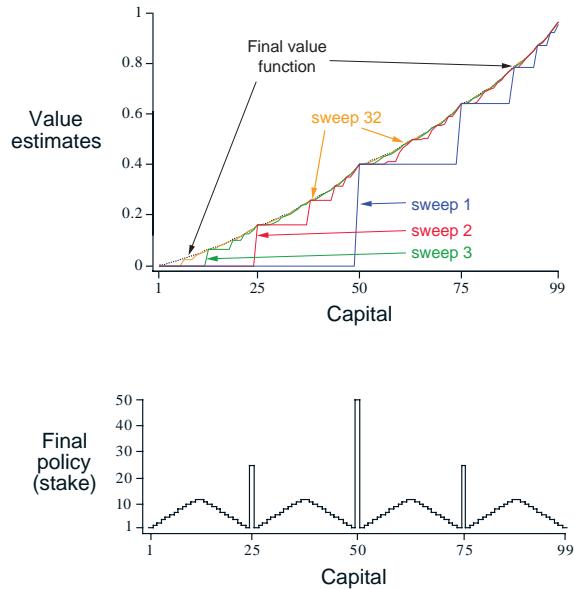


Figure 4.3: The solution to the gambler's problem for $p_h = 0.4$. The upper graph shows the value function found by successive sweeps of value iteration. The lower graph shows the final policy. \blacksquare

4.5 Asynchronous Dynamic Programming

A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive. For example, the game of backgammon has over 10^{20} states. Even if we could perform the value iteration update on a million states per second, it would take over a thousand years to complete a single sweep.

Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms update the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be updated several times before the values of others are updated once. To converge correctly, however, an asynchronous algorithm must continue to update the values of all the states: it can't ignore any state after some point in the computation. Asynchronous DP algorithms allow great flexibility in selecting states to update.

For example, one version of asynchronous value iteration updates the value, in place, of only one state, s_k , on each step, k , using the value iteration update (4.10). If $0 \leq \gamma < 1$, asymptotic convergence to v_* is guaranteed given only that all states occur in the sequence $\{s_k\}$ an infinite number of times (the sequence could even be random).¹ Similarly, it is possible to intermix policy evaluation and value iteration updates to produce a kind of asynchronous truncated policy iteration. Although the details of this and other more unusual DP algorithms are beyond the scope of this book, it is clear that a few different updates form building blocks that can be used flexibly in a wide variety of sweepless DP algorithms.

Of course, avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy. We can try to take advantage of this flexibility by selecting the states to which we apply updates so as to improve the algorithm's rate of progress. We can try to order the updates to let value information propagate from state to state in an efficient way. Some states may not need their values updated as often as others. We might even try to skip updating some states entirely if they are not relevant to optimal behavior. Some ideas for doing this are discussed in Chapter 8.

Asynchronous algorithms also make it easier to intermix computation with real-time interaction. To solve a given MDP, we can run an iterative DP algorithm *at the same time that an agent is actually experiencing the MDP*. The agent's experience can be used to determine the states to which the DP algorithm applies its updates. At the same time, the latest value and policy information from the DP algorithm can guide the agent's decision making. For example, we can apply updates to states as the agent visits them. This makes it possible to *focus* the DP algorithm's updates onto parts of the state set

¹In the undiscounted episodic case, it is possible that there are some orderings of updates that do not result in convergence, but it is relatively easy to avoid these.

that are most relevant to the agent. This kind of focusing is a repeated theme in reinforcement learning.

4.6 Generalized Policy Iteration

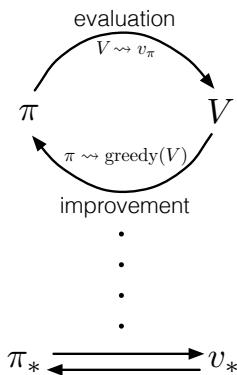
Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, these two processes alternate, each completing before the other begins, but this is not really necessary. In value iteration, for example, only a single iteration of policy evaluation is performed in between each policy improvement. In asynchronous DP methods, the evaluation and improvement processes are interleaved at an even finer grain. In some cases a single state is updated in one process before returning to the other. As long as both processes continue to update all states, the ultimate result is typically the same—convergence to the optimal value function and an optimal policy.

We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy, as suggested by the diagram to the right. If both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal. The value function stabilizes only when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function.

Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation (4.1) holds, and thus that the policy and the value function are optimal.

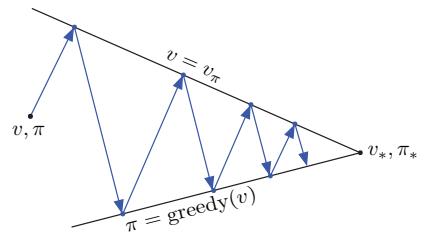
The evaluation and improvement processes in GPI can be viewed as both competing and cooperating. They compete in the sense that they pull in opposing directions. Making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes that policy no longer to be greedy. In the long run, however, these two processes interact to find a single joint solution: the optimal value function and an optimal policy.

One might also think of the interaction between the evaluation and improvement processes in GPI in terms of two constraints or goals—for example, as two lines in



a two-dimensional space as suggested by the diagram to the right. Although the real geometry is much more complicated than this, the diagram suggests what happens in the real case. Each process drives the value function or policy toward one of the lines representing a solution to one of the two goals. The goals interact because the two lines are not orthogonal. Driving directly toward one goal causes some movement away from the other goal.

Inevitably, however, the joint process is brought closer to the overall goal of optimality. The arrows in this diagram correspond to the behavior of policy iteration in that each takes the system all the way to achieving one of the two goals completely. In GPI one could also take smaller, incomplete steps toward each goal. In either case, the two processes together achieve the overall goal of optimality even though neither is attempting to achieve it directly.



4.7 Efficiency of Dynamic Programming

DP may not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient. If we ignore a few technical details, then, in the worst case, the time that DP methods take to find an optimal policy is polynomial in the number of states and actions. If n and k denote the number of states and actions, this means that a DP method takes a number of computational operations that is less than some polynomial function of n and k . A DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is k^n . In this sense, DP is exponentially faster than any direct search in policy space could be, because direct search would have to exhaustively examine each policy to provide the same guarantee. Linear programming methods can also be used to solve MDPs, and in some cases their worst-case convergence guarantees are better than those of DP methods. But linear programming methods become impractical at a much smaller number of states than do DP methods (by a factor of about 100). For the largest problems, only DP methods are feasible.

DP is sometimes thought to be of limited applicability because of the *curse of dimensionality*, the fact that the number of states often grows exponentially with the number of state variables. Large state sets do create difficulties, but these are inherent difficulties of the problem, not of DP as a solution method. In fact, DP is comparatively better suited to handling large state spaces than competing methods such as direct search and linear programming.

In practice, DP methods can be used with today's computers to solve MDPs with millions of states. Both policy iteration and value iteration are widely used, and it is not clear which, if either, is better in general. In practice, these methods usually converge much faster than their theoretical worst-case run times, particularly if they are started with good initial value functions or policies.

On problems with large state spaces, *asynchronous* DP methods are often preferred. To complete even one sweep of a synchronous method requires computation and memory for every state. For some problems, even this much memory and computation is impractical, yet the problem is still potentially solvable because relatively few states occur along optimal solution trajectories. Asynchronous methods and other variations of GPI can be applied in such cases and may find good or optimal policies much faster than synchronous methods can.

4.8 Summary

In this chapter we have become familiar with the basic ideas and algorithms of dynamic programming as they relate to solving finite MDPs. *Policy evaluation* refers to the (typically) iterative computation of the value function for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Putting these two computations together, we obtain *policy iteration* and *value iteration*, the two most popular DP methods. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing an *expected update* operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Expected updates are closely related to Bellman equations: they are little more than these equations turned into assignment statements. When the updates no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation. Just as there are four primary value functions (v_π , v_* , q_π , and q_*), there are four corresponding Bellman equations and four corresponding expected updates. An intuitive view of the operation of DP updates is given by their *backup diagrams*.

Insight into DP methods and, in fact, into almost all reinforcement learning methods, can be gained by viewing them as *generalized policy iteration* (GPI). GPI is the general idea of two interacting processes revolving around an approximate policy and an approximate value function. One process takes the policy as given and performs some form of policy evaluation, changing the value function to be more like the true value function for the policy. The other process takes the value function as given and performs some form of policy improvement, changing the policy to make it better, assuming that the value function is its value function. Although each process changes the basis for the other, overall they work together to find a joint solution: a policy and value function that are unchanged by either process and, consequently, are optimal. In some cases, GPI can be proved to converge, most notably for the classical DP methods that we have presented in this chapter. In other cases convergence has not been proved, but still the idea of GPI improves our understanding of the methods.

It is not necessary to perform DP methods in complete sweeps through the state set. *Asynchronous DP* methods are in-place iterative methods that update states in an arbitrary order, perhaps stochastically determined and using out-of-date information. Many of these methods can be viewed as fine-grained forms of GPI.

Finally, we note one last special property of DP methods. All of them update estimates of the values of states based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea *bootstrapping*. Many reinforcement learning methods perform bootstrapping, even those that do not require, as DP requires, a complete and accurate model of the environment. In the next chapter we explore reinforcement learning methods that do not require a model and do not bootstrap. In the chapter after that we explore methods that do not require a model but do bootstrap. These key features and properties are separable, yet can be mixed in interesting combinations.

Bibliographical and Historical Remarks

The term “dynamic programming” is due to Bellman (1957a), who showed how these methods could be applied to a wide range of problems. Extensive treatments of DP can be found in many texts, including Bertsekas (2005, 2012), Bertsekas and Tsitsiklis (1996), Dreyfus and Law (1977), Ross (1983), White (1969), and Whittle (1982, 1983). Our interest in DP is restricted to its use in solving MDPs, but DP also applies to other types of problems. Kumar and Kanal (1988) provide a more general look at DP.

To the best of our knowledge, the first connection between DP and reinforcement learning was made by Minsky (1961) in commenting on Samuel’s checkers player. In a footnote, Minsky mentioned that it is possible to apply DP to problems in which Samuel’s backing-up process can be handled in closed analytic form. This remark may have misled artificial intelligence researchers into believing that DP was restricted to analytically tractable problems and therefore largely irrelevant to artificial intelligence. Andreae (1969) mentioned DP in the context of reinforcement learning. Werbos (1977) suggested an approach to approximating DP called “heuristic dynamic programming” that emphasizes gradient-descent methods for continuous-state problems (Werbos, 1982, 1987, 1988, 1989, 1992). These methods are closely related to the reinforcement learning algorithms that we discuss in this book. Watkins (1989) was explicit in connecting reinforcement learning to DP, characterizing a class of reinforcement learning methods as “incremental dynamic programming.”

4.1–4 These sections describe well-established DP algorithms that are covered in any of the general DP references cited above. The policy improvement theorem and the policy iteration algorithm are due to Bellman (1957a) and Howard (1960). Our presentation was influenced by the local view of policy improvement taken by Watkins (1989). Our discussion of value iteration as a form of truncated policy iteration is based on the approach of Puterman and Shin (1978), who presented a class of algorithms called *modified policy iteration*, which includes policy iteration and value iteration as special cases. An analysis showing how value iteration can be made to find an optimal policy in finite time is given by Bertsekas (1987).

Iterative policy evaluation is an example of a classical successive approximation algorithm for solving a system of linear equations. The version of the algorithm

that uses two arrays, one holding the old values while the other is updated, is often called a *Jacobi-style* algorithm, after Jacobi's classical use of this method. It is also sometimes called a *synchronous* algorithm because the effect is as if all the values are updated at the same time. The second array is needed to simulate this parallel computation sequentially. The in-place version of the algorithm is often called a *Gauss–Seidel-style* algorithm after the classical Gauss–Seidel algorithm for solving systems of linear equations. In addition to iterative policy evaluation, other DP algorithms can be implemented in these different versions. Bertsekas and Tsitsiklis (1989) provide excellent coverage of these variations and their performance differences.

- 4.5** Asynchronous DP algorithms are due to Bertsekas (1982, 1983), who also called them distributed DP algorithms. The original motivation for asynchronous DP was its implementation on a multiprocessor system with communication delays between processors and no global synchronizing clock. These algorithms are extensively discussed by Bertsekas and Tsitsiklis (1989). Jacobi-style and Gauss–Seidel-style DP algorithms are special cases of the asynchronous version. Williams and Baird (1990) presented DP algorithms that are asynchronous at a finer grain than the ones we have discussed: the update operations themselves are broken into steps that can be performed asynchronously.
- 4.7** This section, written with the help of Michael Littman, is based on Littman, Dean, and Kaelbling (1995). The phrase “curse of dimensionality” is due to Bellman (1957a).
Foundational work on the linear programming approach to reinforcement learning was done by Daniela de Farias (de Farias, 2002; de Farias and Van Roy, 2003).

Chapter 5

Monte Carlo Methods

In this chapter we consider our first learning methods for estimating value functions and discovering optimal policies. Unlike the previous chapter, here we do not assume complete knowledge of the environment. Monte Carlo methods require only *experience*—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Learning from *actual* experience is striking because it requires no prior knowledge of the environment’s dynamics, yet can still attain optimal behavior. Learning from *simulated* experience is also powerful. Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming (DP). In surprisingly many cases it is easy to generate experience sampled according to the desired probability distributions, but infeasible to obtain the distributions in explicit form.

Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns are available, here we define Monte Carlo methods only for episodic tasks. That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. Only on the completion of an episode are value estimates and policies changed. Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense. The term “Monte Carlo” is often used more broadly for any estimation method whose operation involves a significant random component. Here we use it specifically for methods based on averaging complete returns (as opposed to methods that learn from partial returns, considered in the next chapter).

Monte Carlo methods sample and average *returns* for each state–action pair much like the bandit methods we explored in Chapter 2 sample and average *rewards* for each action. The main difference is that now there are multiple states, each acting like a different bandit problem (like an associative-search or contextual bandit) and the different bandit problems are interrelated. That is, the return after taking an action in one state depends on the actions taken in later states in the same episode. Because all the action selections are undergoing learning, the problem becomes nonstationary from the point of view of the earlier state.

To handle the nonstationarity, we adapt the idea of general policy iteration (GPI) developed in Chapter 4 for DP. Whereas there we *computed* value functions from knowledge of the MDP, here we *learn* value functions from sample returns with the MDP. The value functions and corresponding policies still interact to attain optimality in essentially the same way (GPI). As in the DP chapter, first we consider the prediction problem (the computation of v_π and q_π for a fixed arbitrary policy π) then policy improvement, and, finally, the control problem and its solution by GPI. Each of these ideas taken from DP is extended to the Monte Carlo case in which only sample experience is available.

5.1 Monte Carlo Prediction

We begin by considering Monte Carlo methods for learning the state-value function for a given policy. Recall that the value of a state is the expected return—expected cumulative future discounted reward—starting from that state. An obvious way to estimate it from experience, then, is simply to average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

In particular, suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π , given a set of episodes obtained by following π and passing through s . Each occurrence of state s in an episode is called a *visit* to s . Of course, s may be visited multiple times in the same episode; let us call the first time it is visited in an episode the *first visit* to s . The *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following first visits to s , whereas the *every-visit MC method* averages the returns following all visits to s . These two Monte Carlo (MC) methods are very similar but have slightly different theoretical properties. First-visit MC has been most widely studied, dating back to the 1940s, and is the one we focus on in this chapter. Every-visit MC extends more naturally to function approximation and eligibility traces, as discussed in Chapters 9 and 12. First-visit MC is shown in procedural form in the box. Every-visit MC would be the same except without the check for S_t having occurred earlier in the episode.

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in S_0, S_1, \dots, S_{t-1} :

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Both first-visit MC and every-visit MC converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to infinity. This is easy to see for the case of first-visit MC. In this case each return is an independent, identically distributed estimate of $v_\pi(s)$ with finite variance. By the law of large numbers the sequence of averages of these estimates converges to their expected value. Each average is itself an unbiased estimate, and the standard deviation of its error falls as $1/\sqrt{n}$, where n is the number of returns averaged. Every-visit MC is less straightforward, but its estimates also converge quadratically to $v_\pi(s)$ (Singh and Sutton, 1996).

The use of Monte Carlo methods is best illustrated through an example.

Example 5.1: Blackjack The object of the popular casino card game of *blackjack* is to obtain cards the sum of whose numerical values is as great as possible without exceeding 21. All face cards count as 10, and an ace can count either as 1 or as 11. We consider the version in which each player competes independently against the dealer. The game begins with two cards dealt to both dealer and player. One of the dealer's cards is face up and the other is face down. If the player has 21 immediately (an ace and a 10-card), it is called a *natural*. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, then he can request additional cards, one by one (*hits*), until he either stops (*sticks*) or exceeds 21 (*goes bust*). If he goes bust, he loses; if he sticks, then it becomes the dealer's turn. The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise, the outcome—win, lose, or draw—is determined by whose final sum is closer to 21.

Playing blackjack is naturally formulated as an episodic finite MDP. Each game of blackjack is an episode. Rewards of +1, -1, and 0 are given for winning, losing, and drawing, respectively. All rewards within a game are zero, and we do not discount ($\gamma = 1$); therefore these terminal rewards are also the returns. The player's actions are to hit or to stick. The states depend on the player's cards and the dealer's showing card. We assume that cards are dealt from an infinite deck (i.e., with replacement) so that there is no advantage to keeping track of the cards already dealt. If the player holds an ace that he could count as 11 without going bust, then the ace is said to be *usable*. In this case it is always counted as 11 because counting it as 1 would make the sum 11 or less, in which case there is no decision to be made because, obviously, the player should always hit. Thus, the player makes decisions on the basis of three variables: his current sum (12–21), the dealer's one showing card (ace–10), and whether or not he holds a usable ace. This makes for a total of 200 states.

Consider the policy that sticks if the player's sum is 20 or 21, and otherwise hits. To find the state-value function for this policy by a Monte Carlo approach, one simulates many blackjack games using the policy and averages the returns following each state. In this way, we obtained the estimates of the state-value function shown in Figure 5.1. The estimates for states with a usable ace are less certain and less regular because these states are less common. In any event, after 500,000 games the value function is very well approximated.

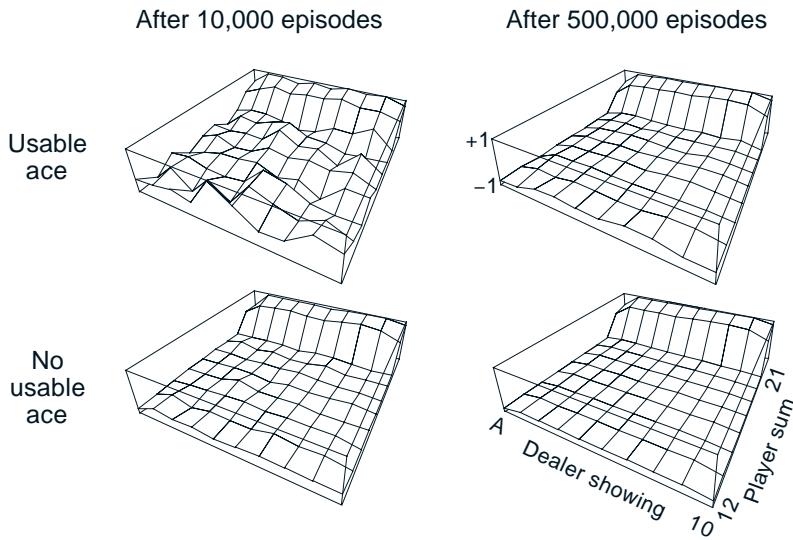


Figure 5.1: Approximate state-value functions for the blackjack policy that sticks only on 20 or 21, computed by Monte Carlo policy evaluation. ■

Exercise 5.1 Consider the diagrams on the right in Figure 5.1. Why does the estimated value function jump up for the last two rows in the rear? Why does it drop off for the whole last row on the left? Why are the frontmost values higher in the upper diagrams than in the lower? □

Exercise 5.2 Suppose every-visit MC was used instead of first-visit MC on the blackjack task. Would you expect the results to be very different? Why or why not? □

Although we have complete knowledge of the environment in the blackjack task, it would not be easy to apply DP methods to compute the value function. DP methods require the distribution of next events—in particular, they require the environments dynamics as given by the four-argument function p —and it is not easy to determine this for blackjack. For example, suppose the player’s sum is 14 and he chooses to stick. What is his probability of terminating with a reward of +1 as a function of the dealer’s showing card? All of the probabilities must be computed *before* DP can be applied, and such computations are often complex and error-prone. In contrast, generating the sample games required by Monte Carlo methods is easy. This is the case surprisingly often; the ability of Monte Carlo methods to work with sample episodes alone can be a significant advantage even when one has complete knowledge of the environment’s dynamics.

Can we generalize the idea of backup diagrams to Monte Carlo algorithms? The general idea of a backup diagram is to show at the top the root node to be updated and to show below all the transitions and leaf nodes whose rewards and estimated values contribute to the update. For Monte Carlo estimation of v_π , the root is a state node, and below it is the entire trajectory of transitions along a particular single episode, ending

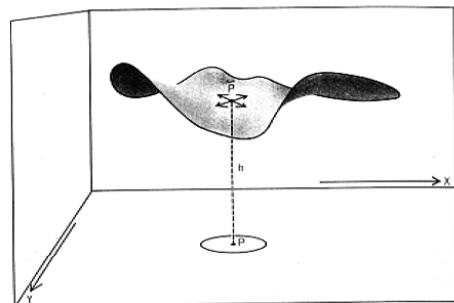
at the terminal state, as shown to the right. Whereas the DP diagram (page 59) shows all possible transitions, the Monte Carlo diagram shows only those sampled on the one episode. Whereas the DP diagram includes only one-step transitions, the Monte Carlo diagram goes all the way to the end of the episode. These differences in the diagrams accurately reflect the fundamental differences between the algorithms.

An important fact about Monte Carlo methods is that the estimates for each state are independent. The estimate for one state does not build upon the estimate of any other state, as is the case in DP. In other words, Monte Carlo methods do not *bootstrap* as we defined it in the previous chapter.

In particular, note that the computational expense of estimating the value of a single state is independent of the number of states. This can make Monte Carlo methods particularly attractive when one requires the value of only one or a subset of states. One can generate many sample episodes starting from the states of interest, averaging returns from only these states, ignoring all others. This is a third advantage Monte Carlo methods can have over DP methods (after the ability to learn from actual experience and from simulated experience).

Example 5.2: Soap Bubble Suppose a wire frame forming a closed loop is dunked in soapy water to form a soap surface or bubble conforming at its edges to the wire frame. If the geometry of the wire frame is irregular but known, how can you compute the shape of the surface? The shape has the property that the total force on each point exerted by neighboring points is zero (or else the shape would change). This means that the surface's height at any point is the average of its heights at points in a small circle around that point. In addition, the surface must meet at its boundaries with the wire frame. The usual approach to problems of this kind is to put a grid over the area covered by the surface and solve for its height at the grid points by an iterative computation. Grid points at the boundary are forced to the wire frame, and all others are adjusted toward the average of the heights of their four nearest neighbors. This process then iterates, much like DP's iterative policy evaluation, and ultimately converges to a close approximation to the desired surface.

This is similar to the kind of problem for which Monte Carlo methods were originally designed. Instead of the iterative computation described above, imagine standing on the surface and taking a random walk, stepping randomly from grid point to neighboring grid point, with equal probability, until you reach the boundary. It turns out that the expected value of the height at the boundary is a close approximation to the height of the desired surface at the starting point (in fact, it is exactly the value computed by the iterative method described above). Thus, one can closely approximate the height of the



A bubble on a wire loop.

From Hersh and Griego (1969). Reproduced with permission. ©1969 Scientific American, a division of Nature America, Inc. All rights reserved.

surface at a point by simply averaging the boundary heights of many walks started at the point. If one is interested in only the value at one point, or any fixed small set of points, then this Monte Carlo method can be far more efficient than the iterative method based on local consistency. ■

5.2 Monte Carlo Estimation of Action Values

If a model is not available, then it is particularly useful to estimate *action* values (the values of state–action pairs) rather than *state* values. With a model, state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to the best combination of reward and next state, as we did in the chapter on DP. Without a model, however, state values alone are not sufficient. One must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. Thus, one of our primary goals for Monte Carlo methods is to estimate q_* . To achieve this, we first consider the policy evaluation problem for action values.

The policy evaluation problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s , taking action a , and thereafter following policy π . The Monte Carlo methods for this are essentially the same as just presented for state values, except now we talk about visits to a state–action pair rather than to a state. A state–action pair s, a is said to be visited in an episode if ever the state s is visited and action a is taken in it. The every-visit MC method estimates the value of a state–action pair as the average of the returns that have followed all the visits to it. The first-visit MC method averages the returns following the first time in each episode that the state was visited and the action was selected. These methods converge quadratically, as before, to the true expected values as the number of visits to each state–action pair approaches infinity.

The only complication is that many state–action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience. This is a serious problem because the purpose of learning action values is to help in choosing among the actions available in each state. To compare alternatives we need to estimate the value of *all* the actions from each state, not just the one we currently favor.

This is the general problem of *maintaining exploration*, as discussed in the context of the k -armed bandit problem in Chapter 2. For policy evaluation to work for action values, we must assure continual exploration. One way to do this is by specifying that the episodes *start in a state–action pair*, and that every pair has a nonzero probability of being selected as the start. This guarantees that all state–action pairs will be visited an infinite number of times in the limit of an infinite number of episodes. We call this the assumption of *exploring starts*.

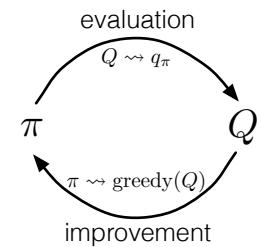
The assumption of exploring starts is sometimes useful, but of course it cannot be relied upon in general, particularly when learning directly from actual interaction with an environment. In that case the starting conditions are unlikely to be so helpful. The most common alternative approach to assuring that all state–action pairs are encountered is

to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state. We discuss two important variants of this approach in later sections. For now, we retain the assumption of exploring starts and complete the presentation of a full Monte Carlo control method.

Exercise 5.3 What is the backup diagram for Monte Carlo estimation of q_π ? \square

5.3 Monte Carlo Control

We are now ready to consider how Monte Carlo estimation can be used in control, that is, to approximate optimal policies. The overall idea is to proceed according to the same pattern as in the DP chapter, that is, according to the idea of generalized policy iteration (GPI). In GPI one maintains both an approximate policy and an approximate value function. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function, as suggested by the diagram to the right. These two kinds of changes work against each other to some extent, as each creates a moving target for the other, but together they cause both policy and value function to approach optimality.



To begin, let us consider a Monte Carlo version of classical policy iteration. In this method, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_*,$$

where $\xrightarrow{\text{E}}$ denotes a complete policy evaluation and $\xrightarrow{\text{I}}$ denotes a complete policy improvement. Policy evaluation is done exactly as described in the preceding section. Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. For the moment, let us assume that we do indeed observe an infinite number of episodes and that, in addition, the episodes are generated with exploring starts. Under these assumptions, the Monte Carlo methods will compute each q_{π_k} exactly, for arbitrary π_k .

Policy improvement is done by making the policy greedy with respect to the current value function. In this case we have an *action*-value function, and therefore no model is needed to construct the greedy policy. For any action-value function q , the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$, deterministically chooses an action with maximal action-value:

$$\pi(s) \doteq \arg \max_a q(s, a). \quad (5.1)$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k} . The policy improvement theorem (Section 4.2) then applies to π_k

and π_{k+1} because, for all $s \in \mathcal{S}$,

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg \max_a q_{\pi_k}(s, a)) \\ &= \max_a q_{\pi_k}(s, a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s). \end{aligned}$$

As we discussed in the previous chapter, the theorem assures us that each π_{k+1} is uniformly better than π_k , or just as good as π_k , in which case they are both optimal policies. This in turn assures us that the overall process converges to the optimal policy and optimal value function. In this way Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics.

We made two unlikely assumptions above in order to easily obtain this guarantee of convergence for the Monte Carlo method. One was that the episodes have exploring starts, and the other was that policy evaluation could be done with an infinite number of episodes. To obtain a practical algorithm we will have to remove both assumptions. We postpone consideration of the first assumption until later in this chapter.

For now we focus on the assumption that policy evaluation operates on an infinite number of episodes. This assumption is relatively easy to remove. In fact, the same issue arises even in classical DP methods such as iterative policy evaluation, which also converge only asymptotically to the true value function. In both DP and Monte Carlo cases there are two ways to solve the problem. One is to hold firm to the idea of approximating q_{π_k} in each policy evaluation. Measurements and assumptions are made to obtain bounds on the magnitude and probability of error in the estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficiently small. This approach can probably be made completely satisfactory in the sense of guaranteeing correct convergence up to some level of approximation. However, it is also likely to require far too many episodes to be useful in practice on any but the smallest problems.

There is a second approach to avoiding the infinite number of episodes nominally required for policy evaluation, in which we give up trying to complete policy evaluation before returning to policy improvement. On each evaluation step we move the value function *toward* q_{π_k} , but we do not expect to actually get close except over many steps. We used this idea when we first introduced the idea of GPI in Section 4.6. One extreme form of the idea is value iteration, in which only one iteration of iterative policy evaluation is performed between each step of policy improvement. The in-place version of value iteration is even more extreme; there we alternate between improvement and evaluation steps for single states.

For Monte Carlo policy iteration it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. A complete simple algorithm along these lines, which we call *Monte Carlo ES*, for Monte Carlo with Exploring Starts, is given in pseudocode in the box on the next page.

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:

$$\begin{aligned}\pi(s) &\in \mathcal{A}(s) \text{ (arbitrarily), for all } s \in \mathcal{S} \\ Q(s, a) &\in \mathbb{R} \text{ (arbitrarily), for all } s \in \mathcal{S}, a \in \mathcal{A}(s) \\ Returns(s, a) &\leftarrow \text{empty list, for all } s \in \mathcal{S}, a \in \mathcal{A}(s)\end{aligned}$$

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0

Generate an episode from S_0, A_0 , following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$$G \leftarrow 0$$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$$G \leftarrow \gamma G + R_{t+1}$$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$$

$$\pi(S_t) \leftarrow \arg \max_a Q(S_t, a)$$

Exercise 5.4 The pseudocode for Monte Carlo ES is inefficient because, for each state-action pair, it maintains a list of all returns and repeatedly calculates their mean. It would be more efficient to use techniques similar to those explained in Section 2.4 to maintain just the mean and a count (for each state-action pair) and update them incrementally. Describe how the pseudocode would be altered to achieve this. \square

In Monte Carlo ES, all the returns for each state-action pair are accumulated and averaged, irrespective of what policy was in force when they were observed. It is easy to see that Monte Carlo ES cannot converge to any suboptimal policy. If it did, then the value function would eventually converge to the value function for that policy, and that in turn would cause the policy to change. Stability is achieved only when both the policy and the value function are optimal. Convergence to this optimal fixed point seems inevitable as the changes to the action-value function decrease over time, but has not yet been formally proved. In our opinion, this is one of the most fundamental open theoretical questions in reinforcement learning (for a partial solution, see Tsitsiklis, 2002).

Example 5.3: Solving Blackjack It is straightforward to apply Monte Carlo ES to blackjack. Because the episodes are all simulated games, it is easy to arrange for exploring starts that include all possibilities. In this case one simply picks the dealer's cards, the player's sum, and whether or not the player has a usable ace, all at random with equal probability. As the initial policy we use the policy evaluated in the previous blackjack example, that which sticks only on 20 or 21. The initial action-value function can be zero for all state-action pairs. Figure 5.2 shows the optimal policy for blackjack found by Monte Carlo ES. This policy is the same as the "basic" strategy of Thorp (1966) with the sole exception of the leftmost notch in the policy for a usable ace, which is not present in Thorp's strategy. We are uncertain of the reason for this discrepancy, but confident that what is shown here is indeed the optimal policy for the version of blackjack we have described.

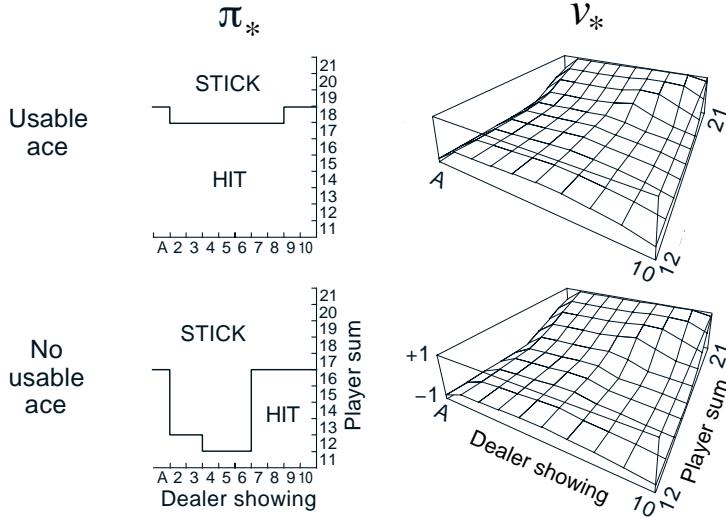


Figure 5.2: The optimal policy and state-value function for blackjack, found by Monte Carlo ES. The state-value function shown was computed from the action-value function found by Monte Carlo ES. ■

5.4 Monte Carlo Control without Exploring Starts

How can we avoid the unlikely assumption of exploring starts? The only general way to ensure that all actions are selected infinitely often is for the agent to continue to select them. There are two approaches to ensuring this, resulting in what we call *on-policy* methods and *off-policy* methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. The Monte Carlo ES method developed above is an example of an on-policy method. In this section we show how an on-policy Monte Carlo control method can be designed that does not use the unrealistic assumption of exploring starts. Off-policy methods are considered in the next section.

In on-policy control methods the policy is generally *soft*, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$, but gradually shifted closer and closer to a deterministic optimal policy. Many of the methods discussed in Chapter 2 provide mechanisms for this. The on-policy method we present in this section uses ε -*greedy* policies, meaning that most of the time they choose an action that has maximal estimated action value, but with probability ε they instead select an action at random. That is, all nongreedy actions are given the minimal probability of selection, $\frac{\varepsilon}{|\mathcal{A}(s)|}$, and the remaining bulk of the probability, $1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$, is given to the greedy action. The ε -greedy policies are examples of ε -*soft* policies, defined as policies for which $\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}(s)|}$ for all states and actions, for some $\varepsilon > 0$. Among ε -soft policies, ε -greedy policies are in some sense those that are closest to greedy.

The overall idea of on-policy Monte Carlo control is still that of GPI. As in Monte Carlo ES, we use first-visit MC methods to estimate the action-value function for the current policy. Without the assumption of exploring starts, however, we cannot simply improve the policy by making it greedy with respect to the current value function, because that would prevent further exploration of nongreedy actions. Fortunately, GPI does not require that the policy be taken all the way to a greedy policy, only that it be moved *toward* a greedy policy. In our on-policy method we will move it only to an ε -greedy policy. For any ε -soft policy, π , any ε -greedy policy with respect to q_π is guaranteed to be better than or equal to π . The complete algorithm is given in the box below.

On-policy first-visit MC control (for ε -soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε -soft policy

$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg \max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon / |\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon / |\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

That any ε -greedy policy with respect to q_π is an improvement over any ε -soft policy π is assured by the policy improvement theorem. Let π' be the ε -greedy policy. The conditions of the policy improvement theorem apply because for any $s \in \mathcal{S}$:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s) q_\pi(s, a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) + (1 - \varepsilon) \max_a q_\pi(s, a) \\ &\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) + (1 - \varepsilon) \sum_a \frac{\pi(a|s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1 - \varepsilon} q_\pi(s, a) \end{aligned} \tag{5.2}$$

(the sum is a weighted average with nonnegative weights summing to 1, and as such it must be less than or equal to the largest number averaged)

$$\begin{aligned} &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) - \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) + \sum_a \pi(a|s) q_\pi(s, a) \\ &= v_\pi(s). \end{aligned}$$

Thus, by the policy improvement theorem, $\pi' \geq \pi$ (i.e., $v_{\pi'}(s) \geq v_\pi(s)$, for all $s \in \mathcal{S}$). We now prove that equality can hold only when both π' and π are optimal among the ε -soft policies, that is, when they are better than or equal to all other ε -soft policies.

Consider a new environment that is just like the original environment, except with the requirement that policies be ε -soft “moved inside” the environment. The new environment has the same action and state set as the original and behaves as follows. If in state s and taking action a , then with probability $1 - \varepsilon$ the new environment behaves exactly like the old environment. With probability ε it repicks the action at random, with equal probabilities, and then behaves like the old environment with the new, random action. The best one can do in this new environment with general policies is the same as the best one could do in the original environment with ε -soft policies. Let \tilde{v}_* and \tilde{q}_* denote the optimal value functions for the new environment. Then a policy π is optimal among ε -soft policies if and only if $v_\pi = \tilde{v}_*$. We know that \tilde{v}_* is the unique solution to the Bellman optimality equation (3.19) with altered transition probabilities:

$$\begin{aligned} \tilde{v}_*(s) &= \max_a \sum_{s', r} \left[(1 - \varepsilon)p(s', r | s, a) + \sum_{a'} \frac{\varepsilon}{|\mathcal{A}(s)|} p(s', r | s, a') \right] \left[r + \gamma \tilde{v}_*(s') \right] \\ &= (1 - \varepsilon) \max_a \sum_{s', r} p(s', r | s, a) \left[r + \gamma \tilde{v}_*(s') \right] \\ &\quad + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s', r} p(s', r | s, a) \left[r + \gamma \tilde{v}_*(s') \right]. \end{aligned}$$

When equality holds and the ε -soft policy π is no longer improved, then we also know, from (5.2), that

$$\begin{aligned} v_\pi(s) &= (1 - \varepsilon) \max_a q_\pi(s, a) + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s, a) \\ &= (1 - \varepsilon) \max_a \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_\pi(s') \right] \\ &\quad + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_\pi(s') \right]. \end{aligned}$$

However, this equation is the same as the previous one, except for the substitution of v_π for \tilde{v}_* . Because \tilde{v}_* is the unique solution, it must be that $v_\pi = \tilde{v}_*$.

In essence, we have shown in the last few pages that policy iteration works for ε -soft policies. Using the natural notion of greedy policy for ε -soft policies, one is assured of improvement on every step, except when the best policy has been found among the ε -soft policies. This analysis is independent of how the action-value functions are determined

at each stage, but it does assume that they are computed exactly. This brings us to roughly the same point as in the previous section. Now we only achieve the best policy among the ε -soft policies, but on the other hand, we have eliminated the assumption of exploring starts.

5.5 Off-policy Prediction via Importance Sampling

All learning control methods face a dilemma: They seek to learn action values conditional on subsequent *optimal* behavior, but they need to behave non-optimally in order to explore all actions (to *find* the optimal actions). How can they learn about the optimal policy while behaving according to an exploratory policy? The on-policy approach in the preceding section is actually a compromise—it learns action values not for the optimal policy, but for a near-optimal policy that still explores. A more straightforward approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. The policy being learned about is called the *target policy*, and the policy used to generate behavior is called the *behavior policy*. In this case we say that learning is from data “off” the target policy, and the overall process is termed *off-policy learning*.

Throughout the rest of this book we consider both on-policy and off-policy methods. On-policy methods are generally simpler and are considered first. Off-policy methods require additional concepts and notation, and because the data is due to a different policy, off-policy methods are often of greater variance and are slower to converge. On the other hand, off-policy methods are more powerful and general. They include on-policy methods as the special case in which the target and behavior policies are the same. Off-policy methods also have a variety of additional uses in applications. For example, they can often be applied to learn from data generated by a conventional non-learning controller, or from a human expert. Off-policy learning is also seen by some as key to learning multi-step predictive models of the world’s dynamics (see Section 17.2; Sutton, 2009; Sutton et al., 2011).

In this section we begin the study of off-policy methods by considering the *prediction* problem, in which both target and behavior policies are fixed. That is, suppose we wish to estimate v_π or q_π , but all we have are episodes following another policy b , where $b \neq \pi$. In this case, π is the target policy, b is the behavior policy, and both policies are considered fixed and given.

In order to use episodes from b to estimate values for π , we require that every action taken under π is also taken, at least occasionally, under b . That is, we require that $\pi(a|s) > 0$ implies $b(a|s) > 0$. This is called the assumption of *coverage*. It follows from coverage that b must be stochastic in states where it is not identical to π . The target policy π , on the other hand, may be deterministic, and, in fact, this is a case of particular interest in control applications. In control, the target policy is typically the deterministic greedy policy with respect to the current estimate of the action-value function. This policy becomes a deterministic optimal policy while the behavior policy remains stochastic and more exploratory, for example, an ε -greedy policy. In this section, however, we consider the prediction problem, in which π is unchanging and given.

Almost all off-policy methods utilize *importance sampling*, a general technique for estimating expected values under one distribution given samples from another. We apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the *importance-sampling ratio*. Given a starting state S_t , the probability of the subsequent state-action trajectory, $A_t, S_{t+1}, A_{t+1}, \dots, S_T$, occurring under any policy π is

$$\begin{aligned} & \Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T \mid S_t, A_{t:T-1} \sim \pi\} \\ &= \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\ &= \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k), \end{aligned}$$

where p here is the state-transition probability function defined by (3.4). Thus, the relative probability of the trajectory under the target and behavior policies (the importance-sampling ratio) is

$$\rho_{t:T-1} \doteq \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \quad (5.3)$$

Although the trajectory probabilities depend on the MDP's transition probabilities, which are generally unknown, they appear identically in both the numerator and denominator, and thus cancel. The importance sampling ratio ends up depending only on the two policies and the sequence, not on the MDP.

Recall that we wish to estimate the expected returns (values) under the target policy, but all we have are returns G_t due to the behavior policy. These returns have the wrong expectation $\mathbb{E}[G_t|S_t=s] = v_b(s)$ and so cannot be averaged to obtain v_π . This is where importance sampling comes in. The ratio $\rho_{t:T-1}$ transforms the returns to have the right expected value:

$$\mathbb{E}[\rho_{t:T-1} G_t \mid S_t=s] = v_\pi(s). \quad (5.4)$$

Now we are ready to give a Monte Carlo algorithm that averages returns from a batch of observed episodes following policy b to estimate $v_\pi(s)$. It is convenient here to number time steps in a way that increases across episode boundaries. That is, if the first episode of the batch ends in a terminal state at time 100, then the next episode begins at time $t = 101$. This enables us to use time-step numbers to refer to particular steps in particular episodes. In particular, we can define the set of all time steps in which state s is visited, denoted $\mathcal{T}(s)$. This is for an every-visit method; for a first-visit method, $\mathcal{T}(s)$ would only include time steps that were first visits to s within their episodes. Also, let $T(t)$ denote the first time of termination following time t , and G_t denote the return after t up through $T(t)$. Then $\{G_t\}_{t \in \mathcal{T}(s)}$ are the returns that pertain to state s , and $\{\rho_{t:T(t)-1}\}_{t \in \mathcal{T}(s)}$ are the corresponding importance-sampling ratios. To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|}. \quad (5.5)$$

When importance sampling is done as a simple average in this way it is called *ordinary importance sampling*.

An important alternative is *weighted importance sampling*, which uses a *weighted* average, defined as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}, \quad (5.6)$$

or zero if the denominator is zero. To understand these two varieties of importance sampling, consider the estimates of their first-visit methods after observing a single return from state s . In the weighted-average estimate, the ratio $\rho_{t:T(t)-1}$ for the single return cancels in the numerator and denominator, so that the estimate is equal to the observed return independent of the ratio (assuming the ratio is nonzero). Given that this return was the only one observed, this is a reasonable estimate, but its expectation is $v_b(s)$ rather than $v_\pi(s)$, and in this statistical sense it is biased. In contrast, the first-visit version of the ordinary importance-sampling estimator (5.5) is always $v_\pi(s)$ in expectation (it is unbiased), but it can be extreme. Suppose the ratio were ten, indicating that the trajectory observed is ten times as likely under the target policy as under the behavior policy. In this case the ordinary importance-sampling estimate would be *ten times* the observed return. That is, it would be quite far from the observed return even though the episode's trajectory is considered very representative of the target policy.

Formally, the difference between the first-visit methods of the two kinds of importance sampling is expressed in their biases and variances. Ordinary importance sampling is unbiased whereas weighted importance sampling is biased (though the bias converges asymptotically to zero). On the other hand, the variance of ordinary importance sampling is in general unbounded because the variance of the ratios can be unbounded, whereas in the weighted estimator the largest weight on any single return is one. In fact, assuming bounded returns, the variance of the weighted importance-sampling estimator converges to zero even if the variance of the ratios themselves is infinite (Precup, Sutton, and Dasgupta 2001). In practice, the weighted estimator usually has dramatically lower variance and is strongly preferred. Nevertheless, we will not totally abandon ordinary importance sampling as it is easier to extend to the approximate methods using function approximation that we explore in the second part of this book.

The every-visit methods for ordinary and weighed importance sampling are both biased, though, again, the bias falls asymptotically to zero as the number of samples increases. In practice, every-visit methods are often preferred because they remove the need to keep track of which states have been visited and because they are much easier to extend to approximations. A complete every-visit MC algorithm for off-policy policy evaluation using weighted importance sampling is given in the next section on page 110.

Exercise 5.5 Consider an MDP with a single nonterminal state and a single action that transitions back to the nonterminal state with probability p and transitions to the terminal state with probability $1-p$. Let the reward be +1 on all transitions, and let $\gamma=1$. Suppose you observe one episode that lasts 10 steps, with a return of 10. What are the first-visit and every-visit estimators of the value of the nonterminal state? \square

Example 5.4: Off-policy Estimation of a Blackjack State Value We applied both ordinary and weighted importance-sampling methods to estimate the value of a single blackjack state (Example 5.1) from off-policy data. Recall that one of the advantages of Monte Carlo methods is that they can be used to evaluate a single state without forming estimates for any other states. In this example, we evaluated the state in which the dealer is showing a deuce, the sum of the player’s cards is 13, and the player has a usable ace (that is, the player holds an ace and a deuce, or equivalently three aces). The data was generated by starting in this state then choosing to hit or stick at random with equal probability (the behavior policy). The target policy was to stick only on a sum of 20 or 21, as in Example 5.1. The value of this state under the target policy is approximately -0.27726 (this was determined by separately generating one-hundred million episodes using the target policy and averaging their returns). Both off-policy methods closely approximated this value after 1000 off-policy episodes using the random policy. To make sure they did this reliably, we performed 100 independent runs, each starting from estimates of zero and learning for 10,000 episodes. Figure 5.3 shows the resultant learning curves—the squared error of the estimates of each method as a function of number of episodes, averaged over the 100 runs. The error approaches zero for both algorithms, but the weighted importance-sampling method has much lower error at the beginning, as is typical in practice.

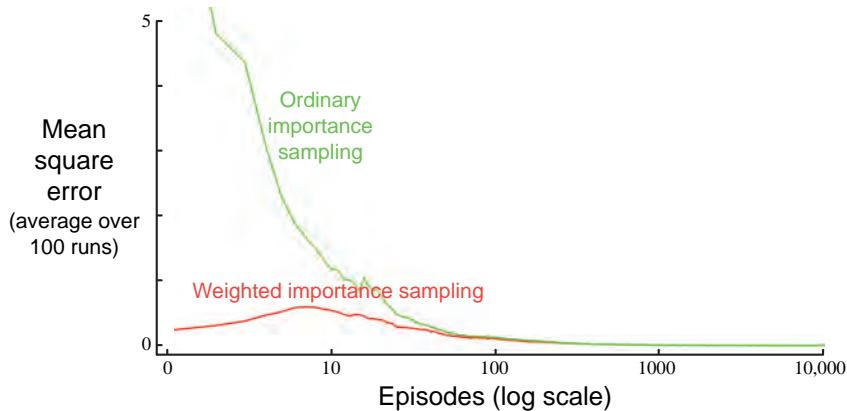


Figure 5.3: Weighted importance sampling produces lower error estimates of the value of a single blackjack state from off-policy episodes. ■

Example 5.5: Infinite Variance The estimates of ordinary importance sampling will typically have infinite variance, and thus unsatisfactory convergence properties, whenever the scaled returns have infinite variance—and this can easily happen in off-policy learning when trajectories contain loops. A simple example is shown inset in Figure 5.4. There is only one nonterminal state s and two actions, right and left. The right action causes a deterministic transition to termination, whereas the left action transitions, with probability 0.9, back to s or, with probability 0.1, on to termination. The rewards are +1 on the latter transition and otherwise zero. Consider the target policy that always selects left. All episodes under this policy consist of some number (possibly zero) of transitions back

to s followed by termination with a reward and return of +1. Thus the value of s under the target policy is 1 ($\gamma = 1$). Suppose we are estimating this value from off-policy data using the behavior policy that selects `right` and `left` with equal probability.

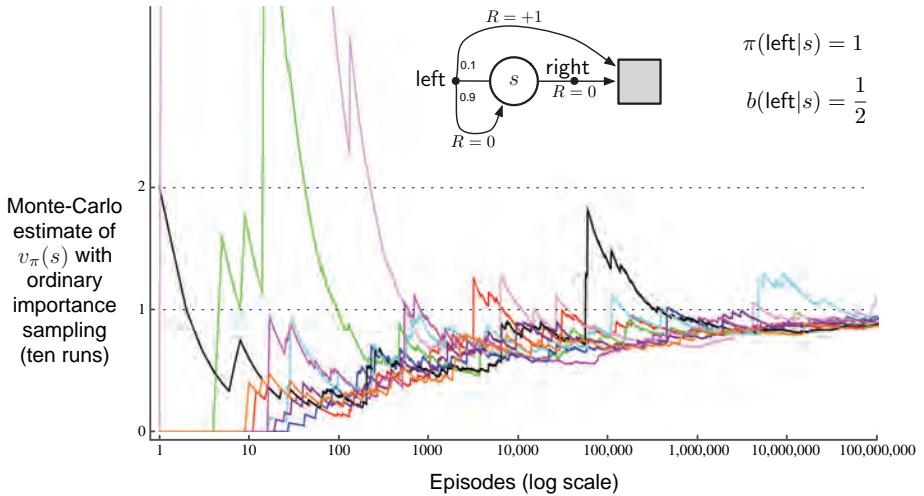


Figure 5.4: Ordinary importance sampling produces surprisingly unstable estimates on the one-state MDP shown inset (Example 5.5). The correct estimate here is 1 ($\gamma = 1$), and, even though this is the expected value of a sample return (after importance sampling), the variance of the samples is infinite, and the estimates do not converge to this value. These results are for off-policy first-visit MC.

The lower part of Figure 5.4 shows ten independent runs of the first-visit MC algorithm using ordinary importance sampling. Even after millions of episodes, the estimates fail to converge to the correct value of 1. In contrast, the weighted importance-sampling algorithm would give an estimate of exactly 1 forever after the first episode that ended with the `left` action. All returns not equal to 1 (that is, ending with the `right` action) would be inconsistent with the target policy and thus would have a $\rho_{t:T(t)-1}$ of zero and contribute neither to the numerator nor denominator of (5.6). The weighted importance-sampling algorithm produces a weighted average of only the returns consistent with the target policy, and all of these would be exactly 1.

We can verify that the variance of the importance-sampling-scaled returns is infinite in this example by a simple calculation. The variance of any random variable X is the expected value of the deviation from its mean \bar{X} , which can be written

$$\text{Var}[X] \doteq \mathbb{E}[(X - \bar{X})^2] = \mathbb{E}[X^2 - 2X\bar{X} + \bar{X}^2] = \mathbb{E}[X^2] - \bar{X}^2.$$

Thus, if the mean is finite, as it is in our case, the variance is infinite if and only if the expectation of the square of the random variable is infinite. Thus, we need only show

that the expected square of the importance-sampling-scaled return is infinite:

$$\mathbb{E} \left[\left(\prod_{t=0}^{T-1} \frac{\pi(A_t|S_t)}{b(A_t|S_t)} G_0 \right)^2 \right].$$

To compute this expectation, we break it down into cases based on episode length and termination. First note that, for any episode ending with the right action, the importance sampling ratio is zero, because the target policy would never take this action; these episodes thus contribute nothing to the expectation (the quantity in parenthesis will be zero) and can be ignored. We need only consider episodes that involve some number (possibly zero) of left actions that transition back to the nonterminal state, followed by a left action transitioning to termination. All of these episodes have a return of 1, so the G_0 factor can be ignored. To get the expected square we need only consider each length of episode, multiplying the probability of the episode's occurrence by the square of its importance-sampling ratio, and add these up:

$$\begin{aligned} &= \frac{1}{2} \cdot 0.1 \left(\frac{1}{0.5} \right)^2 && \text{(the length 1 episode)} \\ &+ \frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.1 \left(\frac{1}{0.5} \frac{1}{0.5} \right)^2 && \text{(the length 2 episode)} \\ &+ \frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.1 \left(\frac{1}{0.5} \frac{1}{0.5} \frac{1}{0.5} \right)^2 && \text{(the length 3 episode)} \\ &+ \dots \\ &= 0.1 \sum_{k=0}^{\infty} 0.9^k \cdot 2^k \cdot 2 = 0.2 \sum_{k=0}^{\infty} 1.8^k = \infty. \end{aligned}$$
■

Exercise 5.6 What is the equation analogous to (5.6) for *action* values $Q(s, a)$ instead of state values $V(s)$, again given returns generated using b ? □

Exercise 5.7 In learning curves such as those shown in Figure 5.3 error generally decreases with training, as indeed happened for the ordinary importance-sampling method. But for the weighted importance-sampling method error first increased and then decreased. Why do you think this happened? □

Exercise 5.8 The results with Example 5.5 and shown in Figure 5.4 used a first-visit MC method. Suppose that instead an every-visit MC method was used on the same problem. Would the variance of the estimator still be infinite? Why or why not? □

5.6 Incremental Implementation

Monte Carlo prediction methods can be implemented incrementally, on an episode-by-episode basis, using extensions of the techniques described in Chapter 2 (Section 2.4). Whereas in Chapter 2 we averaged *rewards*, in Monte Carlo methods we average *returns*. In all other respects exactly the same methods as used in Chapter 2 can be used for *on-policy* Monte Carlo methods. For *off-policy* Monte Carlo methods, we need to separately consider those that use *ordinary* importance sampling and those that use *weighted* importance sampling.

In ordinary importance sampling, the returns are scaled by the importance sampling ratio $\rho_{t:T(t)-1}$ (5.3), then simply averaged, as in (5.5). For these methods we can again use the incremental methods of Chapter 2, but using the scaled returns in place of the rewards of that chapter. This leaves the case of off-policy methods using *weighted* importance sampling. Here we have to form a weighted average of the returns, and a slightly different incremental algorithm is required.

Suppose we have a sequence of returns G_1, G_2, \dots, G_{n-1} , all starting in the same state and each with a corresponding random weight W_i (e.g., $W_i = \rho_{t_i:T(t_i)-1}$). We wish to form the estimate

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \quad n \geq 2, \tag{5.7}$$

and keep it up-to-date as we obtain a single additional return G_n . In addition to keeping track of V_n , we must maintain for each state the cumulative sum C_n of the weights given to the first n returns. The update rule for V_n is

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n} [G_n - V_n], \quad n \geq 1, \tag{5.8}$$

and

$$C_{n+1} \doteq C_n + W_{n+1},$$

where $C_0 \doteq 0$ (and V_1 is arbitrary and thus need not be specified). The box on the next page contains a complete episode-by-episode incremental algorithm for Monte Carlo policy evaluation. The algorithm is nominally for the off-policy case, using weighted importance sampling, but applies as well to the on-policy case just by choosing the target and behavior policies as the same (in which case ($\pi = b$), W is always 1). The approximation Q converges to q_π (for all encountered state-action pairs) while actions are selected according to a potentially different policy, b .

Exercise 5.9 Modify the algorithm for first-visit MC policy evaluation (Section 5.1) to use the incremental implementation for sample averages described in Section 2.4. \square

Exercise 5.10 Derive the weighted-average update rule (5.8) from (5.7). Follow the pattern of the derivation of the unweighted rule (2.3). \square

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$$Q(s, a) \in \mathbb{R} \text{ (arbitrarily)}$$

$$C(s, a) \leftarrow 0$$

Loop forever (for each episode):

$$b \leftarrow \text{any policy with coverage of } \pi$$

Generate an episode following b : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$$G \leftarrow 0$$

$$W \leftarrow 1$$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$, while $W \neq 0$:

$$G \leftarrow \gamma G + R_{t+1}$$

$$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$$

$$W \leftarrow W \frac{\pi(A_t | S_t)}{b(A_t | S_t)}$$

5.7 Off-policy Monte Carlo Control

We are now ready to present an example of the second class of learning control methods we consider in this book: off-policy methods. Recall that the distinguishing feature of on-policy methods is that they estimate the value of a policy while using it for control. In off-policy methods these two functions are separated. The policy used to generate behavior, called the *behavior* policy, may in fact be unrelated to the policy that is evaluated and improved, called the *target* policy. An advantage of this separation is that the target policy may be deterministic (e.g., greedy), while the behavior policy can continue to sample all possible actions.

Off-policy Monte Carlo control methods use one of the techniques presented in the preceding two sections. They follow the behavior policy while learning about and improving the target policy. These techniques require that the behavior policy has a nonzero probability of selecting all actions that might be selected by the target policy (coverage). To explore all possibilities, we require that the behavior policy be soft (i.e., that it select all actions in all states with nonzero probability).

The box on the next page shows an off-policy Monte Carlo control method, based on GPI and weighted importance sampling, for estimating π_* and q_* . The target policy $\pi \approx \pi_*$ is the greedy policy with respect to Q , which is an estimate of q_π . The behavior policy b can be anything, but in order to assure convergence of π to the optimal policy, an infinite number of returns must be obtained for each pair of state and action. This can be assured by choosing b to be ε -soft. The policy π converges to optimal at all encountered states even though actions are selected according to a different soft policy b , which may change between or even within episodes.

Off-policy MC control, for estimating $\pi \approx \pi_*$

```

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :
 $Q(s, a) \in \mathbb{R}$  (arbitrarily)
 $C(s, a) \leftarrow 0$ 
 $\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a)$  (with ties broken consistently)

Loop forever (for each episode):
 $b \leftarrow$  any soft policy
Generate an episode using  $b$ :  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ 
 $G \leftarrow 0$ 
 $W \leftarrow 1$ 
Loop for each step of episode,  $t = T-1, T-2, \dots, 0$ :
 $G \leftarrow \gamma G + R_{t+1}$ 
 $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$ 
 $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$ 
 $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$  (with ties broken consistently)
If  $A_t \neq \pi(S_t)$  then exit inner Loop (proceed to next episode)
 $W \leftarrow W \frac{1}{b(A_t|S_t)}$ 

```

A potential problem is that this method learns only from the tails of episodes, when all of the remaining actions in the episode are greedy. If nongreedy actions are common, then learning will be slow, particularly for states appearing in the early portions of long episodes. Potentially, this could greatly slow learning. There has been insufficient experience with off-policy Monte Carlo methods to assess how serious this problem is. If it is serious, the most important way to address it is probably by incorporating temporal-difference learning, the algorithmic idea developed in the next chapter. Alternatively, if γ is less than 1, then the idea developed in the next section may also help significantly.

Exercise 5.11 In the boxed algorithm for off-policy MC control, you may have been expecting the W update to have involved the importance-sampling ratio $\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$, but instead it involves $\frac{1}{b(A_t|S_t)}$. Why is this nevertheless correct? \square

Exercise 5.12: Racetrack (programming) Consider driving a race car around a turn like those shown in Figure 5.5. You want to go as fast as possible, but not so fast as to run off the track. In our simplified racetrack, the car is at one of a discrete set of grid positions, the cells in the diagram. The velocity is also discrete, a number of grid cells moved horizontally and vertically per time step. The actions are increments to the velocity components. Each may be changed by $+1$, -1 , or 0 in each step, for a total of nine (3×3) actions. Both velocity components are restricted to be nonnegative and less than 5, and they cannot both be zero except at the starting line. Each episode begins in one of the randomly selected start states with both velocity components zero and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random position on the starting line, both velocity components are reduced to zero, and the



Figure 5.5: A couple of right turns for the racetrack task.

episode continues. Before updating the car’s location at each time step, check to see if the projected path of the car intersects the track boundary. If it intersects the finish line, the episode ends; if it intersects anywhere else, the car is considered to have hit the track boundary and is sent back to the starting line. To make the task more challenging, with probability 0.1 at each time step the velocity increments are both zero, independently of the intended increments. Apply a Monte Carlo control method to this task to compute the optimal policy from each starting state. Exhibit several trajectories following the optimal policy (but turn the noise off for these trajectories). \square

5.8 *Discounting-aware Importance Sampling

The off-policy methods that we have considered so far are based on forming importance-sampling weights for returns considered as unitary wholes, without taking into account the returns’ internal structures as sums of discounted rewards. We now briefly consider cutting-edge research ideas for using this structure to significantly reduce the variance of off-policy estimators.

For example, consider the case where episodes are long and γ is significantly less than 1. For concreteness, say that episodes last 100 steps and that $\gamma = 0$. The return from time 0 will then be just $G_0 = R_1$, but its importance sampling ratio will be a product of 100 factors, $\frac{\pi(A_0|S_0)}{b(A_0|S_0)} \frac{\pi(A_1|S_1)}{b(A_1|S_1)} \dots \frac{\pi(A_{99}|S_{99})}{b(A_{99}|S_{99})}$. In ordinary importance sampling, the return will be scaled by the entire product, but it is really only necessary to scale by the first factor, by $\frac{\pi(A_0|S_0)}{b(A_0|S_0)}$. The other 99 factors $\frac{\pi(A_1|S_1)}{b(A_1|S_1)} \dots \frac{\pi(A_{99}|S_{99})}{b(A_{99}|S_{99})}$ are irrelevant because after the first reward the return has already been determined. These later factors are all independent of the return and of expected value 1; they do not change the expected update, but they add enormously to its variance. In some cases they could even make the variance infinite. Let us now consider an idea for avoiding this large extraneous variance.

The essence of the idea is to think of discounting as determining a probability of termination or, equivalently, a *degree* of partial termination. For any $\gamma \in [0, 1)$, we can think of the return G_0 as partly terminating in one step, to the degree $1 - \gamma$, producing a return of just the first reward, R_1 , and as partly terminating after two steps, to the degree $(1 - \gamma)\gamma$, producing a return of $R_1 + R_2$, and so on. The latter degree corresponds to terminating on the second step, $1 - \gamma$, and not having already terminated on the first step, γ . The degree of termination on the third step is thus $(1 - \gamma)\gamma^2$, with the γ^2 reflecting that termination did not occur on either of the first two steps. The partial returns here are called *flat partial returns*:

$$\bar{G}_{t:h} \doteq R_{t+1} + R_{t+2} + \cdots + R_h, \quad 0 \leq t < h \leq T,$$

where “flat” denotes the absence of discounting, and “partial” denotes that these returns do not extend all the way to termination but instead stop at h , called the *horizon* (and T is the time of termination of the episode). The conventional full return G_t can be viewed as a sum of flat partial returns as suggested above as follows:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \\ &= (1 - \gamma)R_{t+1} \\ &\quad + (1 - \gamma)\gamma(R_{t+1} + R_{t+2}) \\ &\quad + (1 - \gamma)\gamma^2(R_{t+1} + R_{t+2} + R_{t+3}) \\ &\quad \vdots \\ &\quad + (1 - \gamma)\gamma^{T-t-2}(R_{t+1} + R_{t+2} + \cdots + R_{T-1}) \\ &\quad + \gamma^{T-t-1}(R_{t+1} + R_{t+2} + \cdots + R_T) \\ &= (1 - \gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} + \gamma^{T-t-1} \bar{G}_{t:T}. \end{aligned}$$

Now we need to scale the flat partial returns by an importance sampling ratio that is similarly truncated. As $\bar{G}_{t:h}$ only involves rewards up to a horizon h , we only need the ratio of the probabilities up to $h - 1$. We define an ordinary importance-sampling estimator, analogous to (5.5), as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1 - \gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{|\mathcal{T}(s)|}, \quad (5.9)$$

and a weighted importance-sampling estimator, analogous to (5.6), as

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left((1 - \gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right)}{\sum_{t \in \mathcal{T}(s)} \left((1 - \gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \right)}. \quad (5.10)$$

We call these two estimators *discounting-aware* importance sampling estimators. They take into account the discount rate but have no effect (are the same as the off-policy estimators from Section 5.5) if $\gamma = 1$.

5.9 *Per-decision Importance Sampling

There is one more way in which the structure of the return as a sum of rewards can be taken into account in off-policy importance sampling, a way that may be able to reduce variance even in the absence of discounting (that is, even if $\gamma = 1$). In the off-policy estimators (5.5) and (5.6), each term of the sum in the numerator is itself a sum:

$$\begin{aligned}\rho_{t:T-1}G_t &= \rho_{t:T-1}(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1}R_T) \\ &= \rho_{t:T-1}R_{t+1} + \gamma\rho_{t:T-1}R_{t+2} + \cdots + \gamma^{T-t-1}\rho_{t:T-1}R_T.\end{aligned}\quad (5.11)$$

The off-policy estimators rely on the expected values of these terms, which can be written in a simpler way. Note that each sub-term of (5.11) is a product of a random reward and a random importance-sampling ratio. For example, the first sub-term can be written, using (5.3), as

$$\rho_{t:T-1}R_{t+1} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})} \frac{\pi(A_{t+2}|S_{t+2})}{b(A_{t+2}|S_{t+2})} \cdots \frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})} R_{t+1}. \quad (5.12)$$

Of all these factors, one might suspect that only the first and the last (the reward) are related; all the others are for events that occurred after the reward. Moreover, the expected value of all these other factors is one:

$$\mathbb{E}\left[\frac{\pi(A_k|S_k)}{b(A_k|S_k)}\right] \doteq \sum_a b(a|S_k) \frac{\pi(a|S_k)}{b(a|S_k)} = \sum_a \pi(a|S_k) = 1. \quad (5.13)$$

With a few more steps, one can show that, as suspected, all of these other factors have no effect in expectation, in other words, that

$$\mathbb{E}[\rho_{t:T-1}R_{t+1}] = \mathbb{E}[\rho_{t:t}R_{t+1}]. \quad (5.14)$$

If we repeat this process for the k th sub-term of (5.11), we get

$$\mathbb{E}[\rho_{t:T-1}R_{t+k}] = \mathbb{E}[\rho_{t:t+k-1}R_{t+k}].$$

It follows then that the expectation of our original term (5.11) can be written

$$\mathbb{E}[\rho_{t:T-1}G_t] = \mathbb{E}\left[\tilde{G}_t\right],$$

where

$$\tilde{G}_t = \rho_{t:t}R_{t+1} + \gamma\rho_{t:t+1}R_{t+2} + \gamma^2\rho_{t:t+2}R_{t+3} + \cdots + \gamma^{T-t-1}\rho_{t:T-1}R_T.$$

We call this idea *per-decision* importance sampling.

It follows immediately that there is an alternate importance-sampling estimator, with the same unbiased expectation (in the first-visit case) as the ordinary-importance-sampling estimator (5.5), using \tilde{G}_t :

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \tilde{G}_t}{|\mathcal{T}(s)|}, \quad (5.15)$$

which we might expect to sometimes be of lower variance.

Is there a per-decision version of *weighted* importance sampling? This is less clear. So far, all the estimators that have been proposed for this that we know of are not consistent (that is, they do not converge to the true value with infinite data).

*Exercise 5.13 Show the steps to derive (5.14) from (5.12). \square

*Exercise 5.14 Modify the algorithm for off-policy Monte Carlo control (page 111) to use the idea of the truncated weighted-average estimator (5.10). Note that you will first need to convert this equation to action values. \square

5.10 Summary

The Monte Carlo methods presented in this chapter learn value functions and optimal policies from experience in the form of *sample episodes*. This gives them at least three kinds of advantages over DP methods. First, they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics. Second, they can be used with simulation or *sample models*. For surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods. Third, it is easy and efficient to *focus* Monte Carlo methods on a small subset of the states. A region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set (we explore this further in Chapter 8).

A fourth advantage of Monte Carlo methods, which we discuss later in the book, is that they may be less harmed by violations of the Markov property. This is because they do not update their value estimates on the basis of the value estimates of successor states. In other words, it is because they do not bootstrap.

In designing Monte Carlo control methods we have followed the overall schema of *generalized policy iteration* (GPI) introduced in Chapter 4. GPI involves interacting processes of policy evaluation and policy improvement. Monte Carlo methods provide an alternative policy evaluation process. Rather than use a model to compute the value of each state, they simply average many returns that start in the state. Because a state's value is the expected return, this average can become a good approximation to the value. In control methods we are particularly interested in approximating action-value functions, because these can be used to improve the policy without requiring a model of the environment's transition dynamics. Monte Carlo methods intermix policy evaluation and policy improvement steps on an episode-by-episode basis, and can be incrementally implemented on an episode-by-episode basis.

Maintaining *sufficient exploration* is an issue in Monte Carlo control methods. It is not enough just to select the actions currently estimated to be best, because then no returns will be obtained for alternative actions, and it may never be learned that they are actually better. One approach is to ignore this problem by assuming that episodes begin with state-action pairs randomly selected to cover all possibilities. Such *exploring starts* can sometimes be arranged in applications with simulated episodes, but are unlikely in learning from real experience. In *on-policy* methods, the agent commits to always exploring and tries to find the best policy that still explores. In *off-policy* methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.

Off-policy prediction refers to learning the value function of a *target policy* from data generated by a different *behavior policy*. Such learning methods are based on some form of *importance sampling*, that is, on weighting returns by the ratio of the probabilities of taking the observed actions under the two policies, thereby transforming their expectations from the behavior policy to the target policy. *Ordinary importance sampling* uses a simple average of the weighted returns, whereas *weighted importance sampling* uses a weighted average. Ordinary importance sampling produces unbiased estimates, but has larger, possibly infinite, variance, whereas weighted importance sampling always has finite variance and is preferred in practice. Despite their conceptual simplicity, off-policy Monte Carlo methods for both prediction and control remain unsettled and are a subject of ongoing research.

The Monte Carlo methods treated in this chapter differ from the DP methods treated in the previous chapter in two major ways. First, they operate on sample experience, and thus can be used for direct learning without a model. Second, they do not bootstrap. That is, they do not update their value estimates on the basis of other value estimates. These two differences are not tightly linked, and can be separated. In the next chapter we consider methods that learn from experience, like Monte Carlo methods, but also bootstrap, like DP methods.

Bibliographical and Historical Remarks

The term “Monte Carlo” dates from the 1940s, when physicists at Los Alamos devised games of chance that they could study to help understand complex physical phenomena relating to the atom bomb. Coverage of Monte Carlo methods in this sense can be found in several textbooks (e.g., Kalos and Whitlock, 1986; Rubinstein, 1981).

5.1–2 Singh and Sutton (1996) distinguished between every-visit and first-visit MC methods and proved results relating these methods to reinforcement learning algorithms. The blackjack example is based on an example used by Widrow, Gupta, and Maitra (1973). The soap bubble example is a classical Dirichlet problem whose Monte Carlo solution was first proposed by Kakutani (1945; see Hersh and Griego, 1969; Doyle and Snell, 1984).

Barto and Duff (1994) discussed policy evaluation in the context of classical Monte Carlo algorithms for solving systems of linear equations. They used the

analysis of Curtiss (1954) to point out the computational advantages of Monte Carlo policy evaluation for large problems.

- 5.3–4** Monte Carlo ES was introduced in the 1998 edition of this book. That may have been the first explicit connection between Monte Carlo estimation and control methods based on policy iteration. An early use of Monte Carlo methods to estimate action values in a reinforcement learning context was by Michie and Chambers (1968). In pole balancing (page 56), they used averages of episode durations to assess the worth (expected balancing “life”) of each possible action in each state, and then used these assessments to control action selections. Their method is similar in spirit to Monte Carlo ES with every-visit MC estimates. Narendra and Wheeler (1986) studied a Monte Carlo method for ergodic finite Markov chains that used the return accumulated between successive visits to the same state as a reward for adjusting a learning automaton’s action probabilities.
- 5.5** Efficient off-policy learning has become recognized as an important challenge that arises in several fields. For example, it is closely related to the idea of “interventions” and “counterfactuals” in probabilistic graphical (Bayesian) models (e.g., Pearl, 1995; Balke and Pearl, 1994). Off-policy methods using importance sampling have a long history and yet still are not well understood. Weighted importance sampling, which is also sometimes called normalized importance sampling (e.g., Koller and Friedman, 2009), is discussed by Rubinstein (1981), Hesterberg (1988), Shelton (2001), and Liu (2001) among others. The target policy in off-policy learning is sometimes referred to in the literature as the “estimation” policy, as it was in the first edition of this book.
- 5.7** The racetrack exercise is adapted from Barto, Bradtke, and Singh (1995), and from Gardner (1973).
- 5.8** Our treatment of the idea of discounting-aware importance sampling is based on the analysis of Sutton, Mahmood, Precup, and van Hasselt (2014). It has been worked out most fully to date by Mahmood (2017; Mahmood, van Hasselt, and Sutton, 2014).
- 5.9** Per-decision importance sampling was introduced by Precup, Sutton, and Singh (2000). They also combined off-policy learning with temporal-difference learning, eligibility traces, and approximation methods, introducing subtle issues that we consider in later chapters.

Exercise 5.15 Make new equations analogous to the importance-sampling Monte Carlo estimates (5.5) and (5.6), but for action value estimates $Q(s, a)$. You will need new notation $\mathcal{T}(s, a)$ for the time steps on which the state-action pair s, a is visited on the episode. Do these estimates involve more or less importance-sampling correction?

Chapter 6

Temporal-Difference Learning

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be *temporal-difference* (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). The relationship between TD, DP, and Monte Carlo methods is a recurring theme in the theory of reinforcement learning; this chapter is the beginning of our exploration of it. Before we are done, we will see that these ideas and methods blend into each other and can be combined in many ways. In particular, in Chapter 7 we introduce n -step algorithms, which provide a bridge from TD to Monte Carlo methods, and in Chapter 12 we introduce the $\text{TD}(\lambda)$ algorithm, which seamlessly unifies them.

As usual, we start by focusing on the policy evaluation or *prediction* problem, the problem of estimating the value function v_π for a given policy π . For the *control* problem (finding an optimal policy), DP, TD, and Monte Carlo methods all use some variation of generalized policy iteration (GPI). The differences in the methods are primarily differences in their approaches to the prediction problem.

6.1 TD Prediction

Both TD and Monte Carlo methods use experience to solve the prediction problem. Given some experience following a policy π , both methods update their estimate V of v_π for the nonterminal states S_t occurring in that experience. Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit Monte Carlo method suitable for nonstationary environments is

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)], \quad (6.1)$$

where G_t is the actual return following time t , and α is a constant step-size parameter (c.f., Equation 2.4). Let us call this method *constant- α MC*. Whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known), TD methods need to wait only until the next time step. At time $t+1$ they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad (6.2)$$

immediately on transition to S_{t+1} and receiving R_{t+1} . In effect, the target for the Monte Carlo update is G_t , whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This TD method is called *TD(0)*, or *one-step TD*, because it is a special case of the $\text{TD}(\lambda)$ and n -step TD methods developed in Chapter 12 and Chapter 7. The box below specifies TD(0) completely in procedural form.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Algorithm parameter: step size $\alpha \in (0, 1]$

Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

$A \leftarrow$ action given by π for S

 Take action A , observe R, S'

$V(S) \leftarrow V(S) + \alpha [R + \gamma V(S') - V(S)]$

$S \leftarrow S'$

 until S is terminal

Because TD(0) bases its update in part on an existing estimate, we say that it is a *bootstrapping* method, like DP. We know from Chapter 3 that

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] \quad (6.3)$$

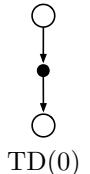
$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \quad (\text{from (3.9)})$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]. \quad (6.4)$$

Roughly speaking, Monte Carlo methods use an estimate of (6.3) as a target, whereas DP methods use an estimate of (6.4) as a target. The Monte Carlo target is an estimate because the expected value in (6.3) is not known; a sample return is used in place of the real expected return. The DP target is an estimate not because of the expected values, which are assumed to be completely provided by a model of the environment, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. The TD target is an estimate for both reasons: it samples the expected values in (6.4) *and* it uses the current estimate V instead of the true v_π . Thus, TD methods combine the sampling of

Monte Carlo with the bootstrapping of DP. As we shall see, with care and imagination this can take us a long way toward obtaining the advantages of both Monte Carlo and DP methods.

Shown to the right is the backup diagram for tabular TD(0). The value estimate for the state node at the top of the backup diagram is updated on the basis of the one sample transition from it to the immediately following state. We refer to TD and Monte Carlo updates as *sample updates* because they involve looking ahead to a sample successor state (or state-action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state-action pair) accordingly. *Sample* updates differ from the *expected* updates of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors.



Finally, note that the quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. This quantity, called the *TD error*, arises in various forms throughout reinforcement learning:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \quad (6.5)$$

Notice that the TD error at each time is the error in the estimate *made at that time*. Because the TD error depends on the next state and next reward, it is not actually available until one time step later. That is, δ_t is the error in $V(S_t)$, available at time $t+1$. Also note that if the array V does not change during the episode (as it does not in Monte Carlo methods), then the Monte Carlo error can be written as a sum of TD errors:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) && \text{(from (3.9))} \\ &= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - V(S_{t+2})) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(G_T - V(S_T)) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0 - 0) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. \end{aligned} \quad (6.6)$$

This identity is not exact if V is updated during the episode (as it is in TD(0)), but if the step size is small then it may still hold approximately. Generalizations of this identity play an important role in the theory and algorithms of temporal-difference learning.

Exercise 6.1 If V changes during the episode, then (6.6) only holds approximately; what would the difference be between the two sides? Let V_t denote the array of state values used at time t in the TD error (6.5) and in the TD update (6.2). Redo the derivation above to determine the additional amount that must be added to the sum of TD errors in order to equal the Monte Carlo error. \square

Example 6.1: Driving Home Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant. Say on this Friday you are leaving at exactly 6 o'clock, and you estimate that it will take 30 minutes to get home. As you reach your car it is 6:05, and you notice it is starting to rain. Traffic is often slower in the rain, so you reestimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn onto the side street where you live at 6:40. Three minutes later you are home. The sequence of states, times, and predictions is thus as follows:

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

The rewards in this example are the elapsed times on each leg of the journey.¹ We are not discounting ($\gamma = 1$), and thus the return for each state is the actual time to go from that state. The value of each state is the *expected* time to go. The second column of numbers gives the current estimated value for each state encountered.

A simple way to view the operation of Monte Carlo methods is to plot the predicted total time (the last column) over the sequence, as in Figure 6.1 (left). The red arrows show the changes in predictions recommended by the constant- α MC method (6.1), for $\alpha = 1$. These are exactly the errors between the estimated value (predicted time to go) in each state and the actual return (actual time to go). For example, when you exited the highway you thought it would take only 15 minutes more to get home, but in fact it took 23 minutes. Equation 6.1 applies at this point and determines an increment in the estimate of time to go after exiting the highway. The error, $G_t - V(S_t)$, at this time is eight minutes. Suppose the step-size parameter, α , is 1/2. Then the predicted time to go after exiting the highway would be revised upward by four minutes as a result of this experience. This is probably too large a change in this case; the truck was probably just an unlucky break. In any event, the change can only be made off-line, that is, after you have reached home. Only at this point do you know any of the actual returns.

Is it necessary to wait until the final outcome is known before learning can begin? Suppose on another day you again estimate when leaving your office that it will take 30 minutes to drive home, but then you become stuck in a massive traffic jam. Twenty-five minutes after leaving the office you are still bumper-to-bumper on the highway. You now

¹If this were a control problem with the objective of minimizing travel time, then we would of course make the rewards the *negative* of the elapsed time. But because we are concerned here only with prediction (policy evaluation), we can keep things simple by using positive numbers.

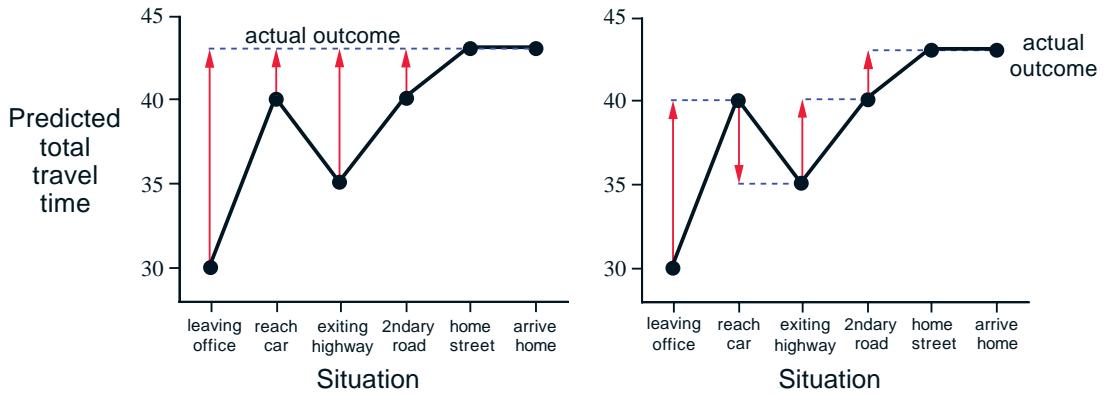


Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

estimate that it will take another 25 minutes to get home, for a total of 50 minutes. As you wait in traffic, you already know that your initial estimate of 30 minutes was too optimistic. Must you wait until you get home before increasing your estimate for the initial state? According to the Monte Carlo approach you must, because you don't yet know the true return.

According to a TD approach, on the other hand, you would learn immediately, shifting your initial estimate from 30 minutes toward 50. In fact, each estimate would be shifted toward the estimate that immediately follows it. Returning to our first day of driving, Figure 6.1 (right) shows the changes in the predictions recommended by the TD rule (6.2) (these are the changes made by the rule if $\alpha = 1$). Each error is proportional to the change over time of the prediction, that is, to the *temporal differences* in predictions.

Besides giving you something to do while waiting in traffic, there are several computational reasons why it is advantageous to learn based on your current predictions rather than waiting until termination when you know the actual return. We briefly discuss some of these in the next section. ■

Exercise 6.2 This is an exercise to help develop your intuition about why TD methods are often more efficient than Monte Carlo methods. Consider the driving home example and how it is addressed by TD and Monte Carlo methods. Can you imagine a scenario in which a TD update would be better on average than a Monte Carlo update? Give an example scenario—a description of past experience and a current state—in which you would expect the TD update to be better. Here's a hint: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original scenario? □

6.2 Advantages of TD Prediction Methods

TD methods update their estimates based in part on other estimates. They learn a guess from a guess—they *bootstrap*. Is this a good thing to do? What advantages do TD methods have over Monte Carlo and DP methods? Developing and answering such questions will take the rest of this book and more. In this section we briefly anticipate some of the answers.

Obviously, TD methods have an advantage over DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

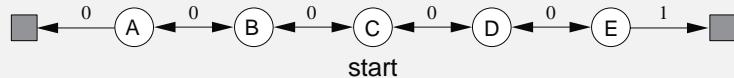
The next most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an online, fully incremental fashion. With Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step. Surprisingly often this turns out to be a critical consideration. Some applications have very long episodes, so that delaying all learning until the end of the episode is too slow. Other applications are continuing tasks and have no episodes at all. Finally, as we noted in the previous chapter, some Monte Carlo methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken.

But are TD methods sound? Certainly it is convenient to learn one guess from the next, without waiting for an actual outcome, but can we still guarantee convergence to the correct answer? Happily, the answer is yes. For any fixed policy π , TD(0) has been proved to converge to v_π , in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions (2.7). Most convergence proofs apply only to the table-based case of the algorithm presented above (6.2), but some also apply to the case of general linear function approximation. These results are discussed in a more general setting in Section 9.4.

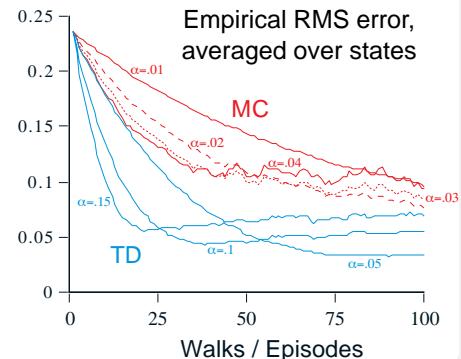
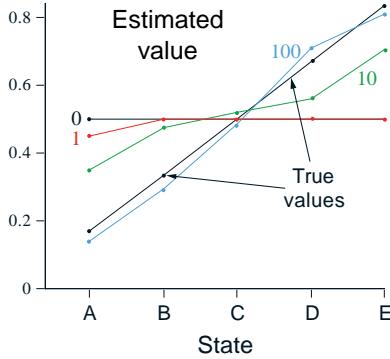
If both TD and Monte Carlo methods converge asymptotically to the correct predictions, then a natural next question is “Which gets there first?” In other words, which method learns faster? Which makes the more efficient use of limited data? At the current time this is an open question in the sense that no one has been able to prove mathematically that one method converges faster than the other. In fact, it is not even clear what is the most appropriate formal way to phrase this question! In practice, however, TD methods have usually been found to converge faster than constant- α MC methods on stochastic tasks, as illustrated in Example 6.2.

Example 6.2 Random Walk

In this example we empirically compare the prediction abilities of TD(0) and constant- α MC when applied to the following Markov reward process:



A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which there is no need to distinguish the dynamics due to the environment from those due to the agent. In this MRP, all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero. For example, a typical episode might consist of the following state-and-reward sequence: C, 0, B, 0, C, 0, D, 0, E, 1. Because this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $v_\pi(C) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$.



The left graph above shows the values learned after various numbers of episodes on a single run of TD(0). The estimates after 100 episodes are about as close as they ever come to the true values—with a constant step-size parameter ($\alpha = 0.1$ in this example), the values fluctuate indefinitely in response to the outcomes of the most recent episodes. The right graph shows learning curves for the two methods for various values of α . The performance measure shown is the root mean square (RMS) error between the value function learned and the true value function, averaged over the five states, then averaged over 100 runs. In all cases the approximate value function was initialized to the intermediate value $V(s) = 0.5$, for all s . The TD method was consistently better than the MC method on this task.

Exercise 6.3 From the results shown in the left graph of the random walk example it appears that the first episode results in a change in only $V(A)$. What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed? \square

Exercise 6.4 The specific results shown in the right graph of the random walk example are dependent on the value of the step-size parameter, α . Do you think the conclusions about which algorithm is better would be affected if a wider range of α values were used? Is there a different, fixed value of α at which either algorithm would have performed significantly better than shown? Why or why not? \square

**Exercise 6.5* In the right graph of the random walk example, the RMS error of the TD method seems to go down and then up again, particularly at high α 's. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized? \square

Exercise 6.6 In Example 6.2 we stated that the true values for the random walk example are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$, for states A through E. Describe at least two different ways that these could have been computed. Which would you guess we actually used? Why? \square

6.3 Optimality of TD(0)

Suppose there is available only a finite amount of experience, say 10 episodes or 100 time steps. In this case, a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer. Given an approximate value function, V , the increments specified by (6.1) or (6.2) are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments. Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges. We call this *batch updating* because updates are made only after processing each complete *batch* of training data.

Under batch updating, TD(0) converges deterministically to a single answer independent of the step-size parameter, α , as long as α is chosen to be sufficiently small. The constant- α MC method also converges deterministically under the same conditions, but to a different answer. Understanding these two answers will help us understand the difference between the two methods. Under normal updating the methods do not move all the way to their respective batch answers, but in some sense they take steps in these directions. Before trying to understand the two answers in general, for all possible tasks, we first look at a few examples.

Example 6.3: Random walk under batch updating Batch-updating versions of TD(0) and constant- α MC were applied as follows to the random walk prediction example (Example 6.2). After each new episode, all episodes seen so far were treated as a batch. They were repeatedly presented to the algorithm, either TD(0) or constant- α MC, with α sufficiently small that the value function converged. The resulting value function was then compared with v_π , and the average root mean square error across the five states (and across 100 independent repetitions of the whole experiment) was plotted to obtain

the learning curves shown in Figure 6.2. Note that the batch TD method was consistently better than the batch Monte Carlo method.

Under batch training, constant- α MC converges to values, $V(s)$, that are sample averages of the actual returns experienced after visiting each state s . These are optimal estimates in the sense that they minimize the mean square error from the actual returns in the training set. In this sense it is surprising that the batch TD method was able to perform better according to the root mean square error measure shown in the figure to the right. How is it that batch TD was able to perform better than this optimal method? The answer is that the Monte Carlo method is optimal only in a limited way, and that TD is optimal in a way that is more relevant to predicting returns. ■

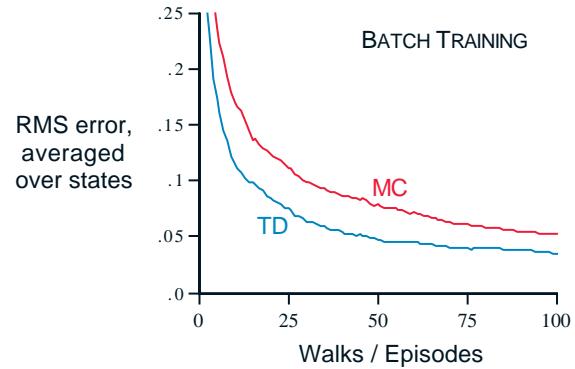


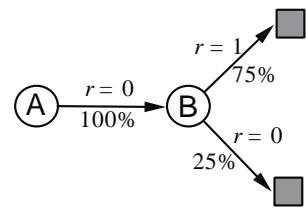
Figure 6.2: Performance of TD(0) and constant- α MC under batch training on the random walk task.

Example 6.4: You are the Predictor Place yourself now in the role of the predictor of returns for an unknown Markov reward process. Suppose you observe the following eight episodes:

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

This means that the first episode started in state A, transitioned to B with a reward of 0, and then terminated from B with a reward of 0. The other seven episodes were even shorter, starting from B and terminating immediately. Given this batch of data, what would you say are the optimal predictions, the best values for the estimates $V(A)$ and $V(B)$? Everyone would probably agree that the optimal value for $V(B)$ is $\frac{3}{4}$, because six out of the eight times in state B the process terminated immediately with a return of 1, and the other two times in B the process terminated immediately with a return of 0.

But what is the optimal value for the estimate $V(A)$ given this data? Here there are two reasonable answers. One is to observe that 100% of the times the process was in state A it traversed immediately to B (with a reward of 0); and because we have already decided that B has value $\frac{3}{4}$, therefore A must have value $\frac{3}{4}$ as well. One way of viewing this answer is that it is based on first modeling the Markov process, in this case as shown to the right, and then computing the correct estimates given the model, which indeed in this case gives $V(A) = \frac{3}{4}$. This is also the answer that batch TD(0) gives.



The other reasonable answer is simply to observe that we have seen A once and the return that followed it was 0; we therefore estimate $V(A)$ as 0. This is the answer that batch Monte Carlo methods give. Notice that it is also the answer that gives minimum squared error on the training data. In fact, it gives zero error on the data. But still we expect the first answer to be better. If the process is Markov, we expect that the first answer will produce lower error on *future* data, even though the Monte Carlo answer is better on the existing data. ■

Example 6.4 illustrates a general difference between the estimates found by batch TD(0) and batch Monte Carlo methods. Batch Monte Carlo methods always find the estimates that minimize mean square error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest. In this case, the maximum-likelihood estimate is the model of the Markov process formed in the obvious way from the observed episodes: the estimated transition probability from i to j is the fraction of observed transitions from i that went to j , and the associated expected reward is the average of the rewards observed on those transitions. Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct. This is called the *certainty-equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. In general, batch TD(0) converges to the certainty-equivalence estimate.

This helps explain why TD methods converge more quickly than Monte Carlo methods. In batch form, TD(0) is faster than Monte Carlo methods because it computes the true certainty-equivalence estimate. This explains the advantage of TD(0) shown in the batch results on the random walk task (Figure 6.2). The relationship to the certainty-equivalence estimate may also explain in part the speed advantage of nonbatch TD(0) (e.g., Example 6.2, page 125, right graph). Although the nonbatch methods do not achieve either the certainty-equivalence or the minimum squared error estimates, they can be understood as moving roughly in these directions. Nonbatch TD(0) may be faster than constant- α MC because it is moving toward a better estimate, even though it is not getting all the way there. At the current time nothing more definite can be said about the relative efficiency of online TD and Monte Carlo methods.

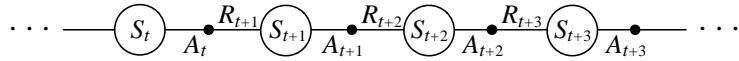
Finally, it is worth noting that although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly. If $n = |\mathcal{S}|$ is the number of states, then just forming the maximum-likelihood estimate of the process may require on the order of n^2 memory, and computing the corresponding value function requires on the order of n^3 computational steps if done conventionally. In these terms it is indeed striking that TD methods can approximate the same solution using memory no more than order n and repeated computations over the training set. On tasks with large state spaces, TD methods may be the only feasible way of approximating the certainty-equivalence solution.

*Exercise 6.7 Design an off-policy version of the TD(0) update that can be used with arbitrary target policy π and covering behavior policy b , using at each step t the importance sampling ratio $\rho_{t:t}$ (5.3). □

6.4 Sarsa: On-policy TD Control

We turn now to the use of TD prediction methods for the control problem. As usual, we follow the pattern of generalized policy iteration (GPI), only this time using TD methods for the evaluation or prediction part. As with Monte Carlo methods, we face the need to trade off exploration and exploitation, and again approaches fall into two main classes: on-policy and off-policy. In this section we present an on-policy TD control method.

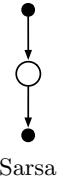
The first step is to learn an action-value function rather than a state-value function. In particular, for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a . This can be done using essentially the same TD method described above for learning v_π . Recall that an episode consists of an alternating sequence of states and state-action pairs:



In the previous section we considered transitions from state to state and learned the values of states. Now we consider transitions from state-action pair to state-action pair, and learn the values of state-action pairs. Formally these cases are identical: they are both Markov chains with a reward process. The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \quad (6.7)$$

This update is done after every transition from a nonterminal state S_t . If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state-action pair to the next. This quintuple gives rise to the name *Sarsa* for the algorithm. The backup diagram for Sarsa is as shown to the right.



It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method. As in all on-policy methods, we continually estimate q_π for the behavior policy π , and at the same time change π toward greediness with respect to q_π . The general form of the Sarsa control algorithm is given in the box on the next page.

The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q . For example, one could use ε -greedy or ε -soft policies. Sarsa converges with probability 1 to an optimal policy and action-value function, under the usual conditions on the step sizes (2.7), as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy (which can be arranged, for example, with ε -greedy policies by setting $\varepsilon = 1/t$).

Exercise 6.8 Show that an action-value version of (6.6) holds for the action-value form of the TD error $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$, again assuming that the values don't change from step to step. \square

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Loop for each step of episode:

 Take action A , observe R, S'

 Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$

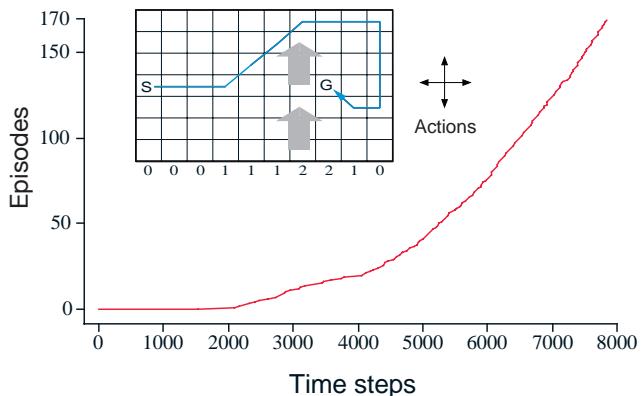
$S \leftarrow S'; A \leftarrow A';$

 until S is terminal

Example 6.5: Windy Gridworld Shown inset below is a standard gridworld, with start and goal states, but with one difference: there is a crosswind running upward through the middle of the grid. The actions are the standard four—`up`, `down`, `right`, and `left`—but in the middle region the resultant next states are shifted upward by a “wind,” the strength of which varies from column to column. The strength of the wind is given below each column, in number of cells shifted upward. For example, if you are one cell to the right of the goal, then the action `left` takes you to the cell just above the goal. This is an undiscounted episodic task, with constant rewards of -1 until the goal state is reached.

The graph to the right shows the results of applying ε -greedy Sarsa to this task, with $\varepsilon = 0.1$, $\alpha = 0.5$, and the initial values $Q(s, a) = 0$ for all s, a . The increasing slope of the graph shows that the goal was reached more quickly over time. By

8000 time steps, the greedy policy was long since optimal (a trajectory from it is shown inset); continued ε -greedy exploration kept the average episode length at about 17 steps, two more than the minimum of 15. Note that Monte Carlo methods cannot easily be used here because termination is not guaranteed for all policies. If a policy was ever found that caused the agent to stay in the same state, then the next episode would never end. Online learning methods such as Sarsa do not have this problem because they quickly learn *during the episode* that such policies are poor, and switch to something else. ■



Exercise 6.9: Windy Gridworld with King's Moves (programming) Re-solve the windy gridworld assuming eight possible actions, including the diagonal moves, rather than four. How much better can you do with the extra actions? Can you do even better by including a ninth action that causes no movement at all other than that caused by the wind? \square

Exercise 6.10: Stochastic Wind (programming) Re-solve the windy gridworld task with King's moves, assuming that the effect of the wind, if there is any, is stochastic, sometimes varying by 1 from the mean values given for each column. That is, a third of the time you move exactly according to these values, as in the previous exercise, but also a third of the time you move one cell above that, and another third of the time you move one cell below that. For example, if you are one cell to the right of the goal and you move `left`, then one-third of the time you move one cell above the goal, one-third of the time you move two cells above the goal, and one-third of the time you move to the goal. \square

6.5 Q-learning: Off-policy TD Control

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as *Q-learning* (Watkins, 1989), defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \quad (6.8)$$

In this case, the learned action-value function, Q , directly approximates q_* , the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we observed in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_* . The Q-learning algorithm is shown below in procedural form.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Take action A , observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

 until S is terminal

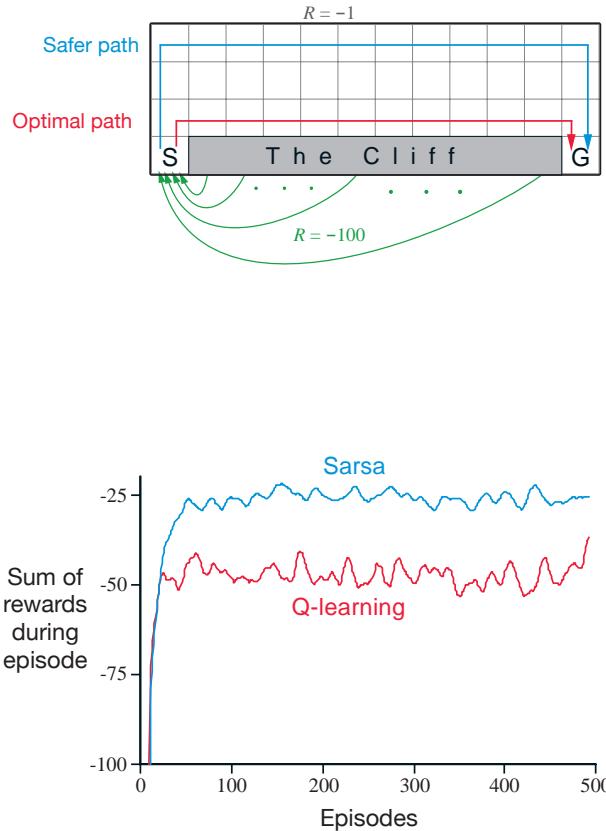
What is the backup diagram for Q-learning? The rule (6.8) updates a state-action pair, so the top node, the root of the update, must be a small, filled action node. The update is also *from* action nodes, maximizing over all those actions possible in the next state. Thus the bottom nodes of the backup diagram should be all these action nodes. Finally, remember that we indicate taking the maximum of these “next action” nodes with an arc across them (Figure 3.4-right). Can you guess now what the diagram is? If so, please do make a guess before turning to the answer in Figure 6.4 on page 134.

Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown to the right. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the region marked “The Cliff.” Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

The graph to the right shows the performance of the Sarsa and Q-learning methods with ε -greedy action selection, $\varepsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε -greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its online performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy. ■

Exercise 6.11 Why is Q-learning considered an *off-policy* control method? □

Exercise 6.12 Suppose action selection is greedy. Is Q-learning then exactly the same algorithm as Sarsa? Will they make exactly the same action selections and weight updates? □



6.6 Expected Sarsa

Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state-action pairs it uses the expected value, taking into account how likely each action is under the current policy. That is, consider the algorithm with the update rule

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi [Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t) \right] \\ &= Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right], \end{aligned} \quad (6.9)$$

but that otherwise follows the schema of Q-learning. Given the next state, S_{t+1} , this algorithm moves *deterministically* in the same direction as Sarsa moves *in expectation*, and accordingly it is called *Expected Sarsa*. Its backup diagram is shown on the right in Figure 6.4.

Expected Sarsa is more complex computationally than Sarsa but, in return, it eliminates the variance due to the random selection of A_{t+1} . Given the same amount of experience we might expect it to perform slightly better than Sarsa, and indeed it generally does. Figure 6.3 shows summary results on the cliff-walking task with Expected Sarsa compared to Sarsa and Q-learning. Expected Sarsa retains the significant advantage of Sarsa over Q-learning on this problem. In addition, Expected Sarsa shows a significant improvement

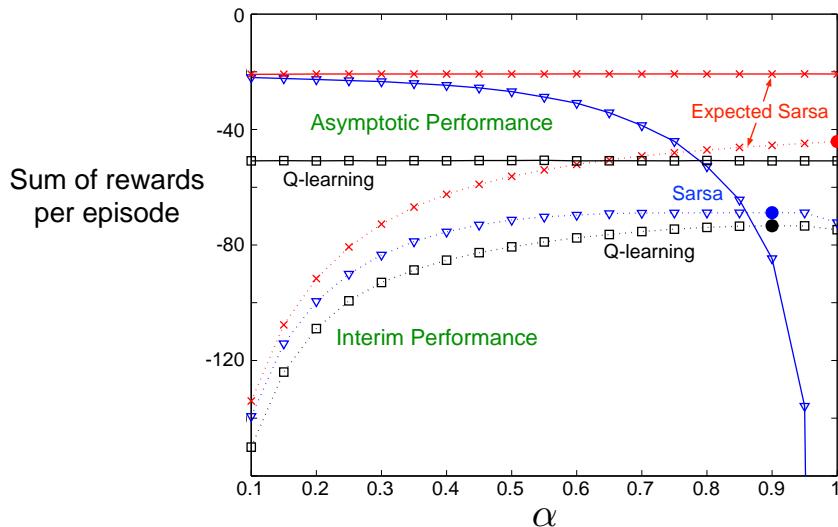


Figure 6.3: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α . All algorithms used an ε -greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method. Adapted from van Seijen et al. (2009).



Figure 6.4: The backup diagrams for Q-learning and Expected Sarsa.

over Sarsa over a wide range of values for the step-size parameter α . In cliff walking the state transitions are all deterministic and all randomness comes from the policy. In such cases, Expected Sarsa can safely set $\alpha = 1$ without suffering any degradation of asymptotic performance, whereas Sarsa can only perform well in the long run at a small value of α , at which short-term performance is poor. In this and other examples there is a consistent empirical advantage of Expected Sarsa over Sarsa.

In these cliff walking results Expected Sarsa was used on-policy, but in general it might use a policy different from the target policy π to generate behavior, in which case it becomes an off-policy algorithm. For example, suppose π is the greedy policy while behavior is more exploratory; then Expected Sarsa is exactly Q-learning. In this sense Expected Sarsa subsumes and generalizes Sarsa while reliably improving over Sarsa. Except for the small additional computational cost, Expected Sarsa may completely dominate both of the other more-well-known TD control algorithms.

6.7 Maximization Bias and Double Learning

All the control algorithms that we have discussed so far involve maximization in the construction of their target policies. For example, in Q-learning the target policy is the greedy policy given the current action values, which is defined with a max, and in Sarsa the policy is often ε -greedy, which also involves a maximization operation. In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant positive bias. To see why, consider a single state s where there are many actions a whose true values, $q(s, a)$, are all zero but whose estimated values, $Q(s, a)$, are uncertain and thus distributed some above and some below zero. The maximum of the true values is zero, but the maximum of the estimates is positive, a positive bias. We call this *maximization bias*.

Example 6.7: Maximization Bias Example The small MDP shown inset in Figure 6.5 provides a simple example of how maximization bias can harm the performance of TD control algorithms. The MDP has two non-terminal states A and B. Episodes always start in A with a choice between two actions, left and right. The right action transitions immediately to the terminal state with a reward and return of zero. The left action transitions to B, also with a reward of zero, from which there are many possible actions all of which cause immediate termination with a reward drawn from a normal distribution with mean -0.1 and variance 1.0. Thus, the expected return for any trajectory starting with left is -0.1 , and thus taking left in state A is always a

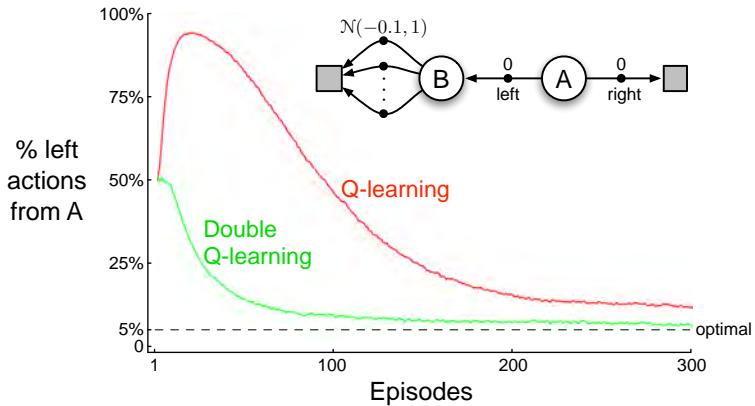


Figure 6.5: Comparison of Q-learning and Double Q-learning on a simple episodic MDP (shown inset). Q-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by ε -greedy action selection with $\varepsilon = 0.1$. In contrast, Double Q-learning is essentially unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. Any ties in ε -greedy action selection were broken randomly.

mistake. Nevertheless, our control methods may favor left because of maximization bias making B appear to have a positive value. Figure 6.5 shows that Q-learning with ε -greedy action selection initially learns to strongly favor the left action on this example. Even at asymptote, Q-learning takes the left action about 5% more often than is optimal at our parameter settings ($\varepsilon = 0.1$, $\alpha = 0.1$, and $\gamma = 1$). ■

Are there algorithms that avoid maximization bias? To start, consider a bandit case in which we have noisy estimates of the value of each of many actions, obtained as sample averages of the rewards received on all the plays with each action. As we discussed above, there will be a positive maximization bias if we use the maximum of the estimates as an estimate of the maximum of the true values. One way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divided the plays in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$, for all $a \in \mathcal{A}$. We could then use one estimate, say Q_1 , to determine the maximizing action $A^* = \operatorname{argmax}_a Q_1(a)$, and the other, Q_2 , to provide the estimate of its value, $Q_2(A^*) = Q_2(\operatorname{argmax}_a Q_1(a))$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(\operatorname{argmax}_a Q_2(a))$. This is the idea of *double learning*. Note that although we learn two estimates, only one estimate is updated on each play; double learning doubles the memory requirements, but does not increase the amount of computation per step.

The idea of double learning extends naturally to algorithms for full MDPs. For example, the double learning algorithm analogous to Q-learning, called Double Q-learning, divides the time steps in two, perhaps by flipping a coin on each step. If the coin comes up heads, the update is

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2(S_{t+1}, \arg \max_a Q_1(S_{t+1}, a)) - Q_1(S_t, A_t) \right]. \quad (6.10)$$

If the coin comes up tails, then the same update is done with Q_1 and Q_2 switched, so that Q_2 is updated. The two approximate value functions are treated completely symmetrically. The behavior policy can use both action-value estimates. For example, an ε -greedy policy for Double Q-learning could be based on the average (or sum) of the two action-value estimates. A complete algorithm for Double Q-learning is given in the box below. This is the algorithm used to produce the results in Figure 6.5. In that example, double learning seems to eliminate the harm caused by maximization bias. Of course there are also double versions of Sarsa and Expected Sarsa.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose A from S using the policy ε -greedy in $Q_1 + Q_2$

 Take action A , observe R, S'

 With 0.5 probability:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha \left(R + \gamma Q_2(S', \arg \max_a Q_1(S', a)) - Q_1(S, A) \right)$$

 else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha \left(R + \gamma Q_1(S', \arg \max_a Q_2(S', a)) - Q_2(S, A) \right)$$

$S \leftarrow S'$

 until S is terminal

*Exercise 6.13 What are the update equations for Double Expected Sarsa with an ε -greedy target policy? \square

6.8 Games, Afterstates, and Other Special Cases

In this book we try to present a uniform approach to a wide class of tasks, but of course there are always exceptional tasks that are better treated in a specialized way. For example, our general approach involves learning an *action*-value function, but in Chapter 1 we presented a TD method for learning to play tic-tac-toe that learned something much more like a *state*-value function. If we look closely at that example, it becomes apparent that the function learned there is neither an action-value function nor a state-value function in the usual sense. A conventional state-value function evaluates states in which the agent has the option of selecting an action, but the state-value function used in

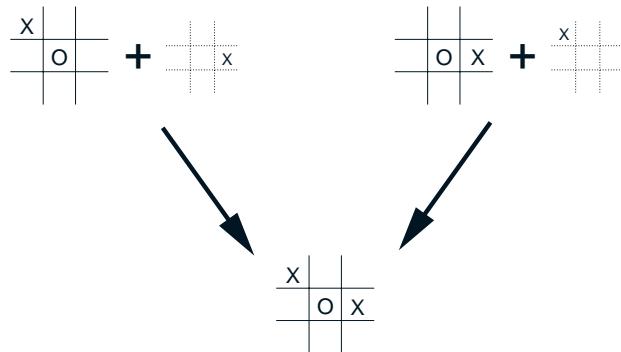
tic-tac-toe evaluates board positions *after* the agent has made its move. Let us call these *afterstates*, and value functions over these, *afterstate value functions*. Afterstates are useful when we have knowledge of an initial part of the environment's dynamics but not necessarily of the full dynamics. For example, in games we typically know the immediate effects of our moves. We know for each possible chess move what the resulting position will be, but not how our opponent will reply. Afterstate value functions are a natural way to take advantage of this kind of knowledge and thereby produce a more efficient learning method.

The reason it is more efficient to design algorithms in terms of afterstates is apparent from the tic-tac-toe example. A conventional action-value function would map from positions *and* moves to an estimate of the value. But many position–move pairs produce the same resulting position, as in the example below: In such cases the position–move pairs are different but produce the same “afterposition,” and thus must have the same value. A conventional action-value function would have to separately assess both pairs, whereas an afterstate value function would immediately assess both equally. Any learning about the position–move pair on the left would immediately transfer to the pair on the right.

Afterstates arise in many tasks, not just games. For example, in queuing tasks there are actions such as assigning customers to servers, rejecting customers, or discarding information. In such cases the actions are in fact defined in terms of their immediate effects, which are completely known.

It is impossible to describe all the possible kinds of specialized problems and corresponding specialized learning algorithms. However, the principles developed in this book should apply widely. For example, afterstate methods are still aptly described in terms of generalized policy iteration, with a policy and (afterstate) value function interacting in essentially the same way. In many cases one will still face the choice between on-policy and off-policy methods for managing the need for persistent exploration.

Exercise 6.14 Describe how the task of Jack’s Car Rental (Example 4.2) could be reformulated in terms of afterstates. Why, in terms of this specific task, would such a reformulation be likely to speed convergence? □



6.9 Summary

In this chapter we introduced a new kind of learning method, temporal-difference (TD) learning, and showed how it can be applied to the reinforcement learning problem. As usual, we divided the overall problem into a prediction problem and a control problem. TD methods are alternatives to Monte Carlo methods for solving the prediction problem. In both cases, the extension to the control problem is via the idea of generalized policy iteration (GPI) that we abstracted from dynamic programming. This is the idea that approximate policy and value functions should interact in such a way that they both move toward their optimal values.

One of the two processes making up GPI drives the value function to accurately predict returns for the current policy; this is the prediction problem. The other process drives the policy to improve locally (e.g., to be ε -greedy) with respect to the current value function. When the first process is based on experience, a complication arises concerning maintaining sufficient exploration. We can classify TD control methods according to whether they deal with this complication by using an on-policy or off-policy approach. Sarsa is an on-policy method, and Q-learning is an off-policy method. Expected Sarsa is also an off-policy method as we present it here. There is a third way in which TD methods can be extended to control which we did not include in this chapter, called actor-critic methods. These methods are covered in full in Chapter 13.

The methods presented in this chapter are today the most widely used reinforcement learning methods. This is probably due to their great simplicity: they can be applied online, with a minimal amount of computation, to experience generated from interaction with an environment; they can be expressed nearly completely by single equations that can be implemented with small computer programs. In the next few chapters we extend these algorithms, making them slightly more complicated and significantly more powerful. All the new algorithms will retain the essence of those introduced here: they will be able to process experience online, with relatively little computation, and they will be driven by TD errors. The special cases of TD methods introduced in the present chapter should rightly be called *one-step, tabular, model-free* TD methods. In the next two chapters we extend them to n -step forms (a link to Monte Carlo methods) and forms that include a model of the environment (a link to planning and dynamic programming). Then, in the second part of the book we extend them to various forms of function approximation rather than tables (a link to deep learning and artificial neural networks).

Finally, in this chapter we have discussed TD methods entirely within the context of reinforcement learning problems, but TD methods are actually more general than this. They are general methods for learning to make long-term predictions about dynamical systems. For example, TD methods may be relevant to predicting financial data, life spans, election outcomes, weather patterns, animal behavior, demands on power stations, or customer purchases. It was only when TD methods were analyzed as pure prediction methods, independent of their use in reinforcement learning, that their theoretical properties first came to be well understood. Even so, these other potential applications of TD learning methods have not yet been extensively explored.

Bibliographical and Historical Remarks

As we outlined in Chapter 1, the idea of TD learning has its early roots in animal learning psychology and artificial intelligence, most notably the work of Samuel (1959) and Klopff (1972). Samuel’s work is described as a case study in Section 16.2. Also related to TD learning are Holland’s (1975, 1976) early ideas about consistency among value predictions. These influenced one of the authors (Barto), who was a graduate student from 1970 to 1975 at the University of Michigan, where Holland was teaching. Holland’s ideas led to a number of TD-related systems, including the work of Booker (1982) and the bucket brigade of Holland (1986), which is related to Sarsa as discussed below.

- 6.1–2** Most of the specific material from these sections is from Sutton (1988), including the TD(0) algorithm, the random walk example, and the term “temporal-difference learning.” The characterization of the relationship to dynamic programming and Monte Carlo methods was influenced by Watkins (1989), Werbos (1987), and others. The use of backup diagrams was new to the first edition of this book.

Tabular TD(0) was proved to converge in the mean by Sutton (1988) and with probability 1 by Dayan (1992), based on the work of Watkins and Dayan (1992). These results were extended and strengthened by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994) by using extensions of the powerful existing theory of stochastic approximation. Other extensions and generalizations are covered in later chapters.

- 6.3** The optimality of the TD algorithm under batch training was established by Sutton (1988). Illuminating this result is Barnard’s (1993) derivation of the TD algorithm as a combination of one step of an incremental method for learning a model of the Markov chain and one step of a method for computing predictions from the model. The term *certainty equivalence* is from the adaptive control literature (e.g., Goodwin and Sin, 1984).

- 6.4** The Sarsa algorithm was introduced by Rummery and Niranjan (1994). They explored it in conjunction with artificial neural networks and called it “Modified Connectionist Q-learning”. The name “Sarsa” was introduced by Sutton (1996). The convergence of one-step tabular Sarsa (the form treated in this chapter) has been proved by Singh, Jaakkola, Littman, and Szepesvári (2000). The “windy gridworld” example was suggested by Tom Kalt.

Holland’s (1986) bucket brigade idea evolved into an algorithm closely related to Sarsa. The original idea of the bucket brigade involved chains of rules triggering each other; it focused on passing credit back from the current rule to the rules that triggered it. Over time, the bucket brigade came to be more like TD learning in passing credit back to any temporally preceding rule, not just to the ones that triggered the current rule. The modern form of the bucket brigade, when simplified in various natural ways, is nearly identical to one-step Sarsa, as detailed by Wilson (1994).

- 6.5** Q-learning was introduced by Watkins (1989), whose outline of a convergence proof was made rigorous by Watkins and Dayan (1992). More general convergence results were proved by Jaakkola, Jordan, and Singh (1994) and Tsitsiklis (1994).
- 6.6** The Expected Sarsa algorithm was introduced by George John (1994), who called it “ \bar{Q} -learning” and stressed its advantages over Q-learning as an off-policy algorithm. John’s work was not known to us when we presented Expected Sarsa in the first edition of this book as an exercise, or to van Seijen, van Hasselt, Whiteson, and Weiring (2009) when they established Expected Sarsa’s convergence properties and conditions under which it will outperform regular Sarsa and Q-learning. Our Figure 6.3 is adapted from their results. Van Seijen et al. defined “Expected Sarsa” to be an on-policy method exclusively (as we did in the first edition), whereas now we use this name for the general algorithm in which the target and behavior policies may differ. The general off-policy view of Expected Sarsa was noted by van Hasselt (2011), who called it “General Q-learning.”
- 6.7** Maximization bias and double learning were introduced and extensively investigated by van Hasselt (2010, 2011). The example MDP in Figure 6.5 was adapted from that in his Figure 4.1 (van Hasselt, 2011).
- 6.8** The notion of an afterstate is the same as that of a “post-decision state” (Van Roy, Bertsekas, Lee, and Tsitsiklis, 1997; Powell, 2011).

Chapter 7

n-step Bootstrapping

In this chapter we unify the Monte Carlo (MC) methods and the one-step temporal-difference (TD) methods presented in the previous two chapters. Neither MC methods nor one-step TD methods are always the best. In this chapter we present *n-step TD methods* that generalize both methods so that one can shift from one to the other smoothly as needed to meet the demands of a particular task. *n*-step methods span a spectrum with MC methods at one end and one-step TD methods at the other. The best methods are often intermediate between the two extremes.

Another way of looking at the benefits of *n*-step methods is that they free you from the tyranny of the time step. With one-step TD methods the same time step determines how often the action can be changed and the time interval over which bootstrapping is done. In many applications one wants to be able to update the action very fast to take into account anything that has changed, but bootstrapping works best if it is over a length of time in which a significant and recognizable state change has occurred. With one-step TD methods, these time intervals are the same, and so a compromise must be made. *n*-step methods enable bootstrapping to occur over multiple steps, freeing us from the tyranny of the single time step.

The idea of *n*-step methods is usually used as an introduction to the algorithmic idea of *eligibility traces* (Chapter 12), which enable bootstrapping over multiple time intervals simultaneously. Here we instead consider the *n*-step bootstrapping idea on its own, postponing the treatment of eligibility-trace mechanisms until later. This allows us to separate the issues better, dealing with as many of them as possible in the simpler *n*-step setting.

As usual, we first consider the prediction problem and then the control problem. That is, we first consider how *n*-step methods can help in predicting returns as a function of state for a fixed policy (i.e., in estimating v_π). Then we extend the ideas to action values and control methods.

7.1 n -step TD Prediction

What is the space of methods lying between Monte Carlo and TD methods? Consider estimating v_π from sample episodes generated using π . Monte Carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode. The update of one-step TD methods, on the other hand, is based on just the one next reward, bootstrapping from the value of the state one step later as a proxy for the remaining rewards. One kind of intermediate method, then, would perform an update based on an intermediate number of rewards: more than one, but less than all of them until termination. For example, a two-step update would be based on the first two rewards and the estimated value of the state two steps later. Similarly, we could have three-step updates, four-step updates, and so on. Figure 7.1 shows the backup diagrams of the spectrum of n -step updates for v_π , with the one-step TD update on the left and the up-until-termination Monte Carlo update on the right.

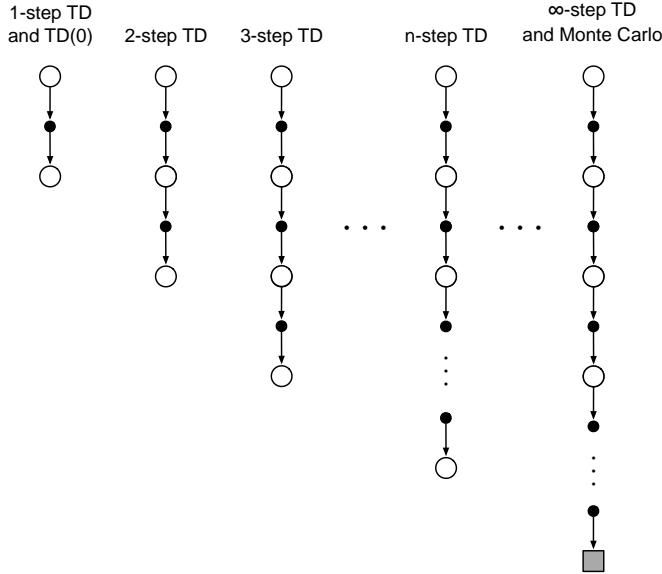


Figure 7.1: The backup diagrams of n -step methods. These methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

The methods that use n -step updates are still TD methods because they still change an earlier estimate based on how it differs from a later estimate. Now the later estimate is not one step later, but n steps later. Methods in which the temporal difference extends over n steps are called *n -step TD methods*. The TD methods introduced in the previous chapter all used one-step updates, which is why we called them one-step TD methods.

More formally, consider the update of the estimated value of state S_t as a result of the state-reward sequence, $S_t, R_{t+1}, S_{t+1}, R_{t+2}, \dots, R_T, S_T$ (omitting the actions). We know that in Monte Carlo updates the estimate of $v_\pi(S_t)$ is updated in the direction of the

complete return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T,$$

where T is the last time step of the episode. Let us call this quantity the *target* of the update. Whereas in Monte Carlo updates the target is the return, in one-step updates the target is the first reward plus the discounted estimated value of the next state, which we call the *one-step return*:

$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1}),$$

where $V_t : \mathcal{S} \rightarrow \mathbb{R}$ here is the estimate at time t of v_π . The subscripts on $G_{t:t+1}$ indicate that it is a truncated return for time t using rewards up until time $t+1$, with the discounted estimate $\gamma V_t(S_{t+1})$ taking the place of the other terms $\gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$ of the full return, as discussed in the previous chapter. Our point now is that this idea makes just as much sense after two steps as it does after one. The target for a two-step update is the *two-step return*:

$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2}),$$

where now $\gamma^2 V_{t+1}(S_{t+2})$ corrects for the absence of the terms $\gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots + \gamma^{T-t-1} R_T$. Similarly, the target for an arbitrary n -step update is the *n-step return*:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \quad (7.1)$$

for all n, t such that $n \geq 1$ and $0 \leq t < T - n$. All n -step returns can be considered approximations to the full return, truncated after n steps and then corrected for the remaining missing terms by $V_{t+n-1}(S_{t+n})$. If $t + n \geq T$ (if the n -step return extends to or beyond termination), then all the missing terms are taken as zero, and the n -step return defined to be equal to the ordinary full return ($G_{t:t+n} \doteq G_t$ if $t + n \geq T$).

Note that n -step returns for $n > 1$ involve future rewards and states that are not available at the time of transition from t to $t + 1$. No real algorithm can use the n -step return until after it has seen R_{t+n} and computed V_{t+n-1} . The first time these are available is $t + n$. The natural state-value learning algorithm for using n -step returns is thus

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha [G_{t:t+n} - V_{t+n-1}(S_t)], \quad 0 \leq t < T, \quad (7.2)$$

while the values of all other states remain unchanged: $V_{t+n}(s) = V_{t+n-1}(s)$, for all $s \neq S_t$. We call this algorithm *n-step TD*. Note that no changes at all are made during the first $n - 1$ steps of each episode. To make up for that, an equal number of additional updates are made at the end of the episode, after termination and before starting the next episode. Complete pseudocode is given in the box on the next page.

Exercise 7.1 In Chapter 6 we noted that the Monte Carlo error can be written as the sum of TD errors (6.6) if the value estimates don't change from step to step. Show that the n -step error used in (7.2) can also be written as a sum of TD errors (again if the value estimates don't change) generalizing the earlier result. \square

Exercise 7.2 (programming) With an n -step method, the value estimates *do* change from step to step, so an algorithm that used the sum of TD errors (see previous exercise) in

***n*-step TD for estimating $V \approx v_\pi$**

Input: a policy π

Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n

Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$

All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:

 Initialize and store $S_0 \neq$ terminal

$T \leftarrow \infty$

 Loop for $t = 0, 1, 2, \dots$:

 If $t < T$, then:

 Take an action according to $\pi(\cdot | S_t)$

 Observe and store the next reward as R_{t+1} and the next state as S_{t+1}

 If S_{t+1} is terminal, then $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)

 If $\tau \geq 0$:

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$$

 If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$

$$V(S_\tau) \leftarrow V(S_\tau) + \alpha [G - V(S_\tau)]$$

 Until $\tau = T - 1$

place of the error in (7.2) would actually be a slightly different algorithm. Would it be a better algorithm or a worse one? Devise and program a small experiment to answer this question empirically. \square

The n -step return uses the value function V_{t+n-1} to correct for the missing rewards beyond R_{t+n} . An important property of n -step returns is that their expectation is guaranteed to be a better estimate of v_π than V_{t+n-1} is, in a worst-state sense. That is, the worst error of the expected n -step return is guaranteed to be less than or equal to γ^n times the worst error under V_{t+n-1} :

$$\max_s |\mathbb{E}_\pi[G_{t:t+n} | S_t = s] - v_\pi(s)| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)|, \quad (7.3)$$

for all $n \geq 1$. This is called the *error reduction property* of n -step returns. Because of the error reduction property, one can show formally that all n -step TD methods converge to the correct predictions under appropriate technical conditions. The n -step TD methods thus form a family of sound methods, with one-step TD methods and Monte Carlo methods as extreme members.

Example 7.1: *n*-step TD Methods on the Random Walk Consider using n -step TD methods on the 5-state random walk task described in Example 6.2 (page 125). Suppose the first episode progressed directly from the center state, C, to the right, through D and E, and then terminated on the right with a return of 1. Recall that the estimated values of all the states started at an intermediate value, $V(s) = 0.5$. As a result of this experience, a one-step method would change only the estimate for the last state,

$V(E)$, which would be incremented toward 1, the observed return. A two-step method, on the other hand, would increment the values of the two states preceding termination: $V(D)$ and $V(E)$ both would be incremented toward 1. A three-step method, or any n -step method for $n > 2$, would increment the values of all three of the visited states toward 1, all by the same amount.

Which value of n is better? Figure 7.2 shows the results of a simple empirical test for a larger random walk process, with 19 states instead of 5 (and with a -1 outcome on the left, all values initialized to 0), which we use as a running example in this chapter. Results are shown for n -step TD methods with a range of values for n and α . The performance measure for each parameter setting, shown on the vertical axis, is the square-root of the average squared error between the predictions at the end of the episode for the 19 states and their true values, then averaged over the first 10 episodes and 100 repetitions of the whole experiment (the same sets of walks were used for all parameter settings). Note that methods with an intermediate value of n worked best. This illustrates how the generalization of TD and Monte Carlo methods to n -step methods can potentially perform better than either of the two extreme methods.

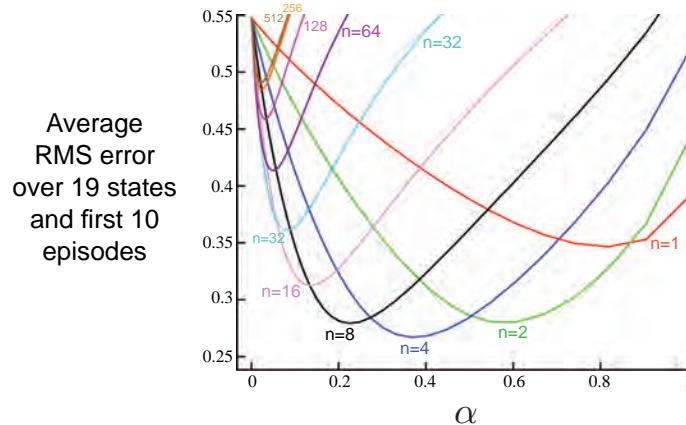


Figure 7.2: Performance of n -step TD methods as a function of α , for various values of n , on a 19-state random walk task (Example 7.1). ■

Exercise 7.3 Why do you think a larger random walk task (19 states instead of 5) was used in the examples of this chapter? Would a smaller walk have shifted the advantage to a different value of n ? How about the change in left-side outcome from 0 to -1 made in the larger walk? Do you think that made any difference in the best value of n ? □

7.2 *n*-step Sarsa

How can n -step methods be used not just for prediction, but for control? In this section we show how n -step methods can be combined with Sarsa in a straightforward way to

produce an on-policy TD control method. The n -step version of Sarsa we call n -step Sarsa, and the original version presented in the previous chapter we henceforth call *one-step Sarsa*, or *Sarsa(0)*.

The main idea is to simply switch states for actions (state-action pairs) and then use an ε -greedy policy. The backup diagrams for n -step Sarsa (shown in Figure 7.3), like those of n -step TD (Figure 7.1), are strings of alternating states and actions, except that the Sarsa ones all start and end with an action rather than a state. We redefine n -step returns (update targets) in terms of estimated action values:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), \quad n \geq 1, 0 \leq t < T-n, \quad (7.4)$$

with $G_{t:t+n} \doteq G_t$ if $t+n \geq T$. The natural algorithm is then

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad 0 \leq t < T, \quad (7.5)$$

while the values of all other states remain unchanged: $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$. This is the algorithm we call n -step Sarsa. Pseudocode is shown in the box on the next page, and an example of why it can speed up learning compared to one-step methods is given in Figure 7.4.

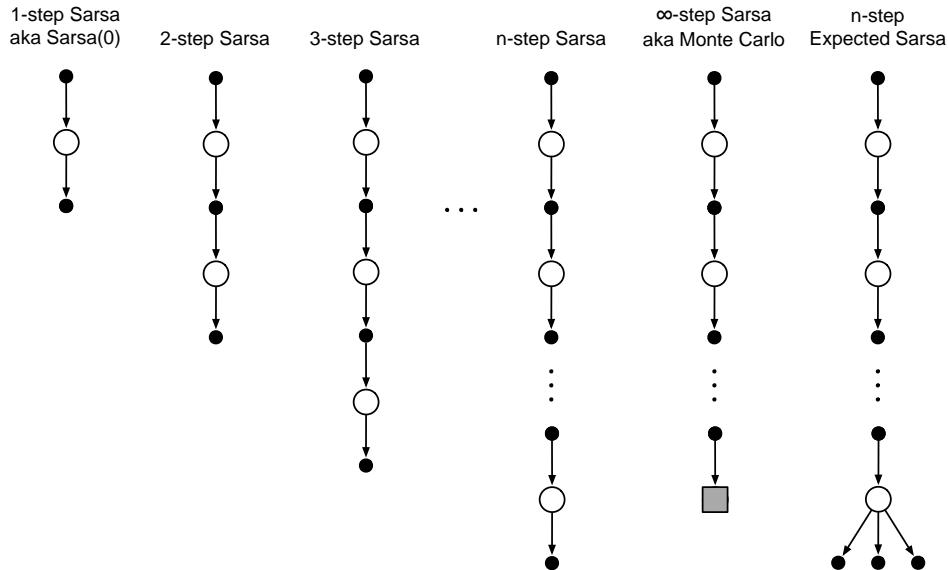


Figure 7.3: The backup diagrams for the spectrum of n -step methods for state-action values. They range from the one-step update of Sarsa(0) to the up-until-termination update of the Monte Carlo method. In between are the n -step updates, based on n steps of real rewards and the estimated value of the n th next state-action pair, all appropriately discounted. On the far right is the backup diagram for n -step Expected Sarsa.

***n*-step Sarsa for estimating $Q \approx q_*$ or q_π**

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Initialize π to be ε -greedy with respect to Q , or to a fixed given policy

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$, a positive integer n

All store and access operations (for S_t , A_t , and R_t) can take their index mod $n + 1$

Loop for each episode:

Initialize and store $S_0 \neq \text{terminal}$

Select and store an action $A_0 \sim \pi(\cdot | S_0)$

$$T \leftarrow \infty$$

Loop for $t = 0, 1, 2, \dots$:

If $t < T$, then:

Take action A_t

Observe and store the ne

S_{t+1} is ter

T

Select and store an action $A_{t+1} \sim \pi(\cdot | S_{t+1})$

$$\tau \leftarrow t -$$

$$\tau \geq 0:$$

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+1, n_1)} \gamma^{i-\tau-1} R_i$$

If $\tau + 1 \leq T - 1$, $G \leftarrow G + \frac{\eta}{\gamma} Q(S_{\tau+1}, A_{\tau+1})$ (G-1)

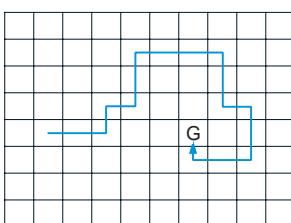
$$Q(S_{\tau+n}, A_{\tau+n}) \leftarrow Q(S_{\tau+n}, A_{\tau+n}) + \alpha [G_n - Q(S_{\tau+n}, A_{\tau+n})]$$

$$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$$

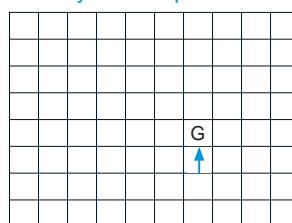
If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε -greedy wrt Q_{T-1}

Until $\tau = T - 1$

Path taken



Action values increased by one-step Sarsa



Action values increased by 10-step Sarsa

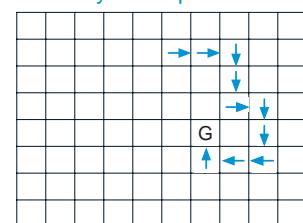


Figure 7.4: Gridworld example of the speedup of policy learning due to the use of n -step methods. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the G. In this example the values were all initially 0, and all rewards were zero except for a positive reward at G. The arrows in the other two panels show which action values were strengthened as a result of this path by one-step and n -step Sarsa methods. The one-step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the n -step method strengthens the last n actions of the sequence, so that much more is learned from the one episode.

Exercise 7.4 Prove that the *n*-step return of Sarsa (7.4) can be written exactly in terms of a novel TD error, as

$$G_{t:t+n} = Q_{t-1}(S_t, A_t) + \sum_{k=t}^{\min(t+n, T)-1} \gamma^{k-t} [R_{k+1} + \gamma Q_k(S_{k+1}, A_{k+1}) - Q_{k-1}(S_k, A_k)]. \quad (7.6)$$

□

What about Expected Sarsa? The backup diagram for the *n*-step version of Expected Sarsa is shown on the far right in Figure 7.3. It consists of a linear string of sample actions and states, just as in *n*-step Sarsa, except that its last element is a branch over all action possibilities weighted, as always, by their probability under π . This algorithm can be described by the same equation as *n*-step Sarsa (above) except with the *n*-step return redefined as

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}), \quad t+n < T, \quad (7.7)$$

(with $G_{t:t+n} \doteq G_t$ for $t+n \geq T$) where $\bar{V}_t(s)$ is the *expected approximate value* of state s , using the estimated action values at time t , under the target policy:

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) Q_t(s, a), \quad \text{for all } s \in \mathcal{S}. \quad (7.8)$$

Expected approximate values are used in developing many of the action-value methods in the rest of this book. If s is terminal, then its expected approximate value is defined to be 0.

7.3 *n*-step Off-policy Learning

Recall that off-policy learning is learning the value function for one policy, π , while following another policy, b . Often, π is the greedy policy for the current action-value-function estimate, and b is a more exploratory policy, perhaps ε -greedy. In order to use the data from b we must take into account the difference between the two policies, using their relative probability of taking the actions that were taken (see Section 5.5). In *n*-step methods, returns are constructed over *n* steps, so we are interested in the relative probability of just those *n* actions. For example, to make a simple off-policy version of *n*-step TD, the update for time t (actually made at time $t+n$) can simply be weighted by $\rho_{t:t+n-1}$:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1} [G_{t:t+n} - V_{t+n-1}(S_t)], \quad 0 \leq t < T, \quad (7.9)$$

where $\rho_{t:t+n-1}$, called the *importance sampling ratio*, is the relative probability under the two policies of taking the *n* actions from A_t to A_{t+n-1} (cf. Eq. 5.3):

$$\rho_{t:h} \doteq \prod_{k=t}^{\min(h, T)-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \quad (7.10)$$

For example, if any one of the actions would never be taken by π (i.e., $\pi(A_k|S_k) = 0$) then the n -step return should be given zero weight and be totally ignored. On the other hand, if by chance an action is taken that π would take with much greater probability than b does, then this will increase the weight that would otherwise be given to the return. This makes sense because that action is characteristic of π (and therefore we want to learn about it) but is selected only rarely by b and thus rarely appears in the data. To make up for this we have to over-weight it when it does occur. Note that if the two policies are actually the same (the on-policy case) then the importance sampling ratio is always 1. Thus our new update (7.9) generalizes and can completely replace our earlier n -step TD update. Similarly, our previous n -step Sarsa update can be completely replaced by a simple off-policy form:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n} [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], \quad (7.11)$$

for $0 \leq t < T$. Note that the importance sampling ratio here starts and ends one step later than for n -step TD (7.9). This is because here we are updating a state-action pair. We do not have to care how likely we were to select the action; now that we have selected it we want to learn fully from what happens, with importance sampling only for subsequent actions. Pseudocode for the full algorithm is shown in the box below.

Off-policy n -step Sarsa for estimating $Q \approx q_*$ or q_π

```

Input: an arbitrary behavior policy  $b$  such that  $b(a|s) > 0$ , for all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
Initialize  $Q(s, a)$  arbitrarily, for all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
Initialize  $\pi$  to be greedy with respect to  $Q$ , or as a fixed given policy
Algorithm parameters: step size  $\alpha \in (0, 1]$ , a positive integer  $n$ 
All store and access operations (for  $S_t$ ,  $A_t$ , and  $R_t$ ) can take their index mod  $n + 1$ 

Loop for each episode:
  Initialize and store  $S_0 \neq$  terminal
  Select and store an action  $A_0 \sim b(\cdot|S_0)$ 
   $T \leftarrow \infty$ 
  Loop for  $t = 0, 1, 2, \dots$  :
    If  $t < T$ , then:
      Take action  $A_t$ 
      Observe and store the next reward as  $R_{t+1}$  and the next state as  $S_{t+1}$ 
      If  $S_{t+1}$  is terminal, then:
         $T \leftarrow t + 1$ 
      else:
        Select and store an action  $A_{t+1} \sim b(\cdot|S_{t+1})$ 
       $\tau \leftarrow t - n + 1$    ( $\tau$  is the time whose estimate is being updated)
      If  $\tau \geq 0$ :
         $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n, T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$                                 $(\rho_{\tau+1:t+n})$ 
         $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$ 
        If  $\tau + n < T$ , then:  $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$                           $(G_{\tau:\tau+n})$ 
         $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \rho [G - Q(S_\tau, A_\tau)]$ 
        If  $\pi$  is being learned, then ensure that  $\pi(\cdot|S_\tau)$  is greedy wrt  $Q$ 
    Until  $\tau = T - 1$ 

```

The off-policy version of *n*-step Expected Sarsa would use the same update as above for *n*-step Sarsa except that the importance sampling ratio would have one less factor in it. That is, the above equation would use $\rho_{t+1:t+n-1}$ instead of $\rho_{t+1:t+n}$, and of course it would use the Expected Sarsa version of the *n*-step return (7.7). This is because in Expected Sarsa all possible actions are taken into account in the last state; the one actually taken has no effect and does not have to be corrected for.

7.4 *Per-decision Methods with Control Variates

The multi-step off-policy methods presented in the previous section are simple and conceptually clear, but are probably not the most efficient. A more sophisticated approach would use per-decision importance sampling ideas such as were introduced in Section 5.9. To understand this approach, first note that the ordinary *n*-step return (7.1), like all returns, can be written recursively. For the *n* steps ending at horizon *h*, the *n*-step return can be written

$$G_{t:h} = R_{t+1} + \gamma G_{t+1:h}, \quad t < h < T, \quad (7.12)$$

where $G_{h:h} \doteq V_{h-1}(S_h)$. (Recall that this return is used at time *h*, previously denoted *t* + *n*.) Now consider the effect of following a behavior policy *b* that is not the same as the target policy π . All of the resulting experience, including the first reward R_{t+1} and the next state S_{t+1} , must be weighted by the importance sampling ratio for time *t*, $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$. One might be tempted to simply weight the righthand side of the above equation, but one can do better. Suppose the action at time *t* would never be selected by π , so that ρ_t is zero. Then a simple weighting would result in the *n*-step return being zero, which could result in high variance when it was used as a target. Instead, in this more sophisticated approach, one uses an alternate, *off-policy* definition of the *n*-step return ending at horizon *h*, as

$$G_{t:h} \doteq \rho_t (R_{t+1} + \gamma G_{t+1:h}) + (1 - \rho_t) V_{h-1}(S_t), \quad t < h < T, \quad (7.13)$$

where again $G_{h:h} \doteq V_{h-1}(S_h)$. In this approach, if ρ_t is zero, then instead of the target being zero and causing the estimate to shrink, the target is the same as the estimate and causes no change. The importance sampling ratio being zero means we should ignore the sample, so leaving the estimate unchanged seems appropriate. The second, additional term in (7.13) is called a *control variate* (for obscure reasons). Notice that the control variate does not change the expected update; the importance sampling ratio has expected value one (Section 5.9) and is uncorrelated with the estimate, so the expected value of the control variate is zero. Also note that the off-policy definition (7.13) is a strict generalization of the earlier on-policy definition of the *n*-step return (7.1), as the two are identical in the on-policy case, in which ρ_t is always 1.

For a conventional *n*-step method, the learning rule to use in conjunction with (7.13) is the *n*-step TD update (7.2), which has no explicit importance sampling ratios other than those embedded in the return.

Exercise 7.5 Write the pseudocode for the off-policy state-value prediction algorithm described above. \square

For action values, the off-policy definition of the n -step return is a little different because the first action does not play a role in the importance sampling. That first action is the one being learned; it does not matter if it was unlikely or even impossible under the target policy—it has been taken and now full unit weight must be given to the reward and state that follows it. Importance sampling will apply only to the actions that follow it.

First note that for action values the n -step *on-policy* return ending at horizon h , expectation form (7.7), can be written recursively just as in (7.12), except that for action values the recursion ends with $G_{h:h} \doteq \bar{V}_{h-1}(S_h)$ as in (7.8). An off-policy form with control variates is

$$\begin{aligned} G_{t:h} &\doteq R_{t+1} + \gamma \left(\rho_{t+1} G_{t+1:h} + \bar{V}_{h-1}(S_{t+1}) - \rho_{t+1} Q_{h-1}(S_{t+1}, A_{t+1}) \right), \\ &= R_{t+1} + \gamma \rho_{t+1} \left(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1}) \right) + \gamma \bar{V}_{h-1}(S_{t+1}), \quad t < h \leq T. \end{aligned} \tag{7.14}$$

If $h < T$, then the recursion ends with $G_{h:h} \doteq Q_{h-1}(S_h, A_h)$, whereas, if $h \geq T$, the recursion ends with $G_{T-1:h} \doteq R_T$. The resultant prediction algorithm (after combining with (7.5)) is analogous to Expected Sarsa.

Exercise 7.6 Prove that the control variate in the above equations does not change the expected value of the return. \square

**Exercise 7.7* Write the pseudocode for the off-policy action-value prediction algorithm described immediately above. Pay particular attention to the termination conditions for the recursion upon hitting the horizon or the end of episode. \square

Exercise 7.8 Show that the general (off-policy) version of the n -step return (7.13) can still be written exactly and compactly as the sum of state-based TD errors (6.5) if the approximate state value function does not change. \square

Exercise 7.9 Repeat the above exercise for the action version of the off-policy n -step return (7.14) and the Expected Sarsa TD error (the quantity in brackets in Equation 6.9). \square

Exercise 7.10 (programming) Devise a small off-policy prediction problem and use it to show that the off-policy learning algorithm using (7.13) and (7.2) is more data efficient than the simpler algorithm using (7.1) and (7.9). \square

The importance sampling that we have used in this section, the previous section, and in Chapter 5, enables sound off-policy learning, but also results in high variance updates, forcing the use of a small step-size parameter and thereby causing learning to be slow. It is probably inevitable that off-policy training is slower than on-policy training—after all, the data is less relevant to what is being learned. However, it is probably also true that these methods can be improved on. The control variates are one way of reducing the variance. Another is to rapidly adapt the step sizes to the observed variance, as in the Autostep method (Mahmood, Sutton, Degris and Pilarski, 2012). Yet another promising approach is the invariant updates of Karampatziakis and Langford (2010) as extended to TD by Tian (in preparation). The usage technique of Mahmood (2017; Mahmood

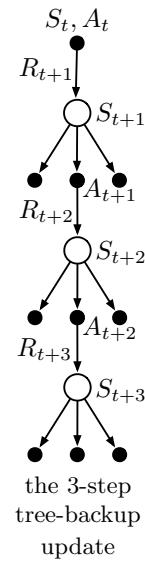
and Sutton, 2015) may also be part of the solution. In the next section we consider an off-policy learning method that does not use importance sampling.

7.5 Off-policy Learning Without Importance Sampling: The *n*-step Tree Backup Algorithm

Is off-policy learning possible without importance sampling? Q-learning and Expected Sarsa from Chapter 6 do this for the one-step case, but is there a corresponding multi-step algorithm? In this section we present just such an *n*-step method, called the *tree-backup algorithm*.

The idea of the algorithm is suggested by the 3-step tree-backup backup diagram shown to the right. Down the central spine and labeled in the diagram are three sample states and rewards, and two sample actions. These are the random variables representing the events occurring after the initial state-action pair S_t, A_t . Hanging off to the sides of each state are the actions that were *not* selected. (For the last state, all the actions are considered to have not (yet) been selected.) Because we have no sample data for the unselected actions, we bootstrap and use the estimates of their values in forming the target for the update. This slightly extends the idea of a backup diagram. So far we have always updated the estimated value of the node at the top of the diagram toward a target combining the rewards along the way (appropriately discounted) and the estimated values of the nodes at the bottom. In the tree-backup update, the target includes all these things *plus* the estimated values of the dangling action nodes hanging off the sides, at all levels. This is why it is called a *tree-backup* update; it is an update from the entire tree of estimated action values.

More precisely, the update is from the estimated action values of the *leaf nodes* of the tree. The action nodes in the interior, corresponding to the actual actions taken, do not participate. Each leaf node contributes to the target with a weight proportional to its probability of occurring under the target policy π . Thus each first-level action a contributes with a weight of $\pi(a|S_{t+1})$, except that the action actually taken, A_{t+1} , does not contribute at all. Its probability, $\pi(A_{t+1}|S_{t+1})$, is used to weight all the second-level action values. Thus, each non-selected second-level action a' contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(a'|S_{t+2})$. Each third-level action contributes with weight $\pi(A_{t+1}|S_{t+1})\pi(A_{t+2}|S_{t+2})\pi(a''|S_{t+3})$, and so on. It is as if each arrow to an action node in the diagram is weighted by the action's probability of being selected under the target policy and, if there is a tree below the action, then that weight applies to all the leaf nodes in the tree.



We can think of the 3-step tree-backup update as consisting of 6 half-steps, alternating between sample half-steps from an action to a subsequent state, and expected half-steps considering from that state all possible actions with their probabilities of occurring under the policy.

Now let us develop the detailed equations for the n -step tree-backup algorithm. The one-step return (target) is the same as that of Expected Sarsa,

$$G_{t:t+1} \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a), \quad (7.15)$$

for $t < T - 1$, and the two-step tree-backup return is

$$\begin{aligned} G_{t:t+2} &\doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) \\ &\quad + \gamma \pi(A_{t+1}|S_{t+1}) \left(R_{t+2} + \gamma \sum_a \pi(a|S_{t+2}) Q_{t+1}(S_{t+2}, a) \right) \\ &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+2}, \end{aligned}$$

for $t < T - 2$. The latter form suggests the general recursive definition of the tree-backup n -step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n}, \quad (7.16)$$

for $t < T - 1, n \geq 2$, with the $n = 1$ case handled by (7.15) except for $G_{T-1:T+n} \doteq R_T$. This target is then used with the usual action-value update rule from n -step Sarsa:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)],$$

for $0 \leq t < T$, while the values of all other state-action pairs remain unchanged: $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$, for all s, a such that $s \neq S_t$ or $a \neq A_t$. Pseudocode for this algorithm is shown in the box on the next page.

Exercise 7.11 Show that if the approximate action values are unchanging, then the tree-backup return (7.16) can be written as a sum of expectation-based TD errors:

$$G_{t:t+n} = Q(S_t, A_t) + \sum_{k=t}^{\min(t+n-1, T-1)} \delta_k \prod_{i=t+1}^k \gamma \pi(A_i|S_i),$$

where $\delta_t \doteq R_{t+1} + \gamma \bar{V}_t(S_{t+1}) - Q(S_t, A_t)$ and \bar{V}_t is given by (7.8). \square

***n*-step Tree Backup for estimating $Q \approx q_*$ or q_π**

```

Initialize  $Q(s, a)$  arbitrarily, for all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
Initialize  $\pi$  to be greedy with respect to  $Q$ , or as a fixed given policy
Algorithm parameters: step size  $\alpha \in (0, 1]$ , a positive integer  $n$ 
All store and access operations can take their index mod  $n + 1$ 

Loop for each episode:
  Initialize and store  $S_0 \neq$  terminal
  Choose an action  $A_0$  arbitrarily as a function of  $S_0$ ; Store  $A_0$ 
   $T \leftarrow \infty$ 
  Loop for  $t = 0, 1, 2, \dots$  :
    If  $t < T$ :
      Take action  $A_t$ ; observe and store the next reward and state as  $R_{t+1}, S_{t+1}$ 
      If  $S_{t+1}$  is terminal:
         $T \leftarrow t + 1$ 
      else:
        Choose an action  $A_{t+1}$  arbitrarily as a function of  $S_{t+1}$ ; Store  $A_{t+1}$ 
         $\tau \leftarrow t + 1 - n$  ( $\tau$  is the time whose estimate is being updated)
        If  $\tau \geq 0$ :
          If  $t + 1 \geq T$ :
             $G \leftarrow R_T$ 
          else
             $G \leftarrow R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a)$ 
        Loop for  $k = \min(t, T - 1)$  down through  $\tau + 1$ :
           $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k)Q(S_k, a) + \gamma \pi(A_k|S_k)G$ 
           $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$ 
          If  $\pi$  is being learned, then ensure that  $\pi(\cdot|S_\tau)$  is greedy wrt  $Q$ 
    Until  $\tau = T - 1$ 

```

7.6 *A Unifying Algorithm: *n*-step $Q(\sigma)$

So far in this chapter we have considered three different kinds of action-value algorithms, corresponding to the first three backup diagrams shown in Figure 7.5. *n*-step Sarsa has all sample transitions, the tree-backup algorithm has all state-to-action transitions fully branched without sampling, and *n*-step Expected Sarsa has all sample transitions except for the last state-to-action one, which is fully branched with an expected value. To what extent can these algorithms be unified?

One idea for unification is suggested by the fourth backup diagram in Figure 7.5. This is the idea that one might decide on a step-by-step basis whether one wanted to take the action as a sample, as in Sarsa, or consider the expectation over all actions instead, as in the tree-backup update. Then, if one chose always to sample, one would obtain Sarsa, whereas if one chose never to sample, one would get the tree-backup algorithm. Expected Sarsa would be the case where one chose to sample for all steps except for the last one.

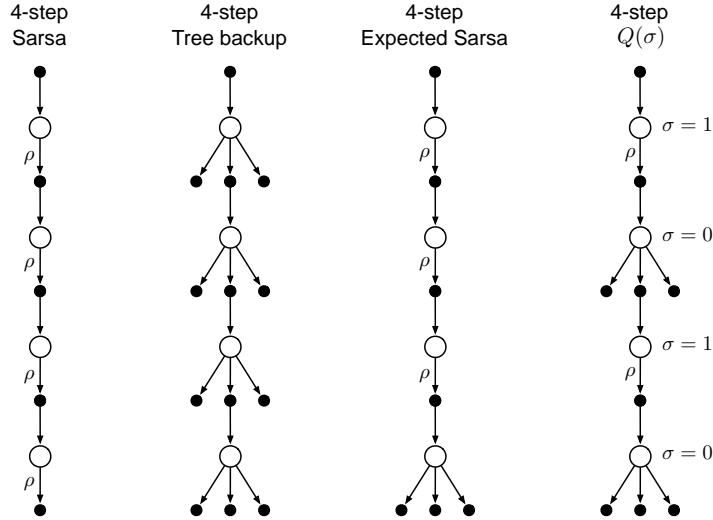


Figure 7.5: The backup diagrams of the three kinds of n -step action-value updates considered so far in this chapter (4-step case) plus the backup diagram of a fourth kind of update that unifies them all. The label ‘ ρ ’ indicates half transitions on which importance sampling is required in the off-policy case. The fourth kind of update unifies all the others by choosing on a state-by-state basis whether to sample ($\sigma_t = 1$) or not ($\sigma_t = 0$).

And of course there would be many other possibilities, as suggested by the last diagram in the figure. To increase the possibilities even further we can consider a continuous variation between sampling and expectation. Let $\sigma_t \in [0, 1]$ denote the degree of sampling on step t , with $\sigma = 1$ denoting full sampling and $\sigma = 0$ denoting a pure expectation with no sampling. The random variable σ_t might be set as a function of the state, action, or state-action pair at time t . We call this proposed new algorithm n -step $Q(\sigma)$.

Now let us develop the equations of n -step $Q(\sigma)$. First we write the tree-backup n -step return (7.16) in terms of the horizon $h = t + n$ and then in terms of the expected approximate value \bar{V} (7.8):

$$\begin{aligned} G_{t:h} &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{h-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:h} \\ &= R_{t+1} + \gamma \bar{V}_{h-1}(S_{t+1}) - \gamma \pi(A_{t+1}|S_{t+1}) Q_{h-1}(S_{t+1}, A_{t+1}) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:h} \\ &= R_{t+1} + \gamma \pi(A_{t+1}|S_{t+1}) (G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})) + \gamma \bar{V}_{h-1}(S_{t+1}), \end{aligned}$$

after which it is exactly like the n -step return for Sarsa with control variates (7.14) except with the action probability $\pi(A_{t+1}|S_{t+1})$ substituted for the importance-sampling ratio ρ_{t+1} . For $Q(\sigma)$, we slide linearly between these two cases:

$$\begin{aligned} G_{t:h} &\doteq R_{t+1} + \gamma (\sigma_{t+1} \rho_{t+1} + (1 - \sigma_{t+1}) \pi(A_{t+1}|S_{t+1})) (G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})) \\ &\quad + \gamma \bar{V}_{h-1}(S_{t+1}), \end{aligned} \tag{7.17}$$

for $t < h \leq T$. The recursion ends with $G_{h:h} \doteq Q_{h-1}(S_h, A_h)$ if $h < T$, or with $G_{T-1:T} \doteq R_T$ if $h = T$. Then we use the earlier update for n-step Sarsa without importance-sampling ratios (7.5) instead of (7.11), because now the ratios are incorporated in the n-step return. A complete algorithm is given in the box.

Off-policy n-step $Q(\sigma)$ for estimating $Q \approx q_*$ or q_π

Input: an arbitrary behavior policy b such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Initialize π to be greedy with respect to Q , or else it is a fixed given policy

Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n

All store and access operations can take their index mod $n + 1$

Loop for each episode:

 Initialize and store $S_0 \neq$ terminal

 Choose and store an action $A_0 \sim b(\cdot|S_0)$

$T \leftarrow \infty$

 Loop for $t = 0, 1, 2, \dots$:

 If $t < T$:

 Take action A_t ; observe and store the next reward and state as R_{t+1}, S_{t+1}

 If S_{t+1} is terminal:

$T \leftarrow t + 1$

 else:

 Choose and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$

 Select and store σ_{t+1}

 Store $\frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})}$ as ρ_{t+1}

$\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)

 If $\tau \geq 0$:

 If $t + 1 < T$:

$G \leftarrow Q(S_{t+1}, A_{t+1})$

 Loop for $k = \min(t + 1, T)$ down through $\tau + 1$:

 if $k = T$:

$G \leftarrow R_T$

 else:

$\bar{V} \leftarrow \sum_a \pi(a|S_k)Q(S_k, a)$

$G \leftarrow R_k + \gamma(\sigma_k \rho_k + (1 - \sigma_k)\pi(A_k|S_k))(G - Q(S_k, A_k)) + \gamma\bar{V}$

$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$

 If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q

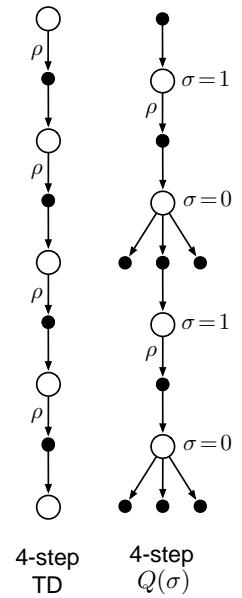
 Until $\tau = T - 1$

7.7 Summary

In this chapter we have developed a range of temporal-difference learning methods that lie in between the one-step TD methods of the previous chapter and the Monte Carlo methods of the chapter before. Methods that involve an intermediate amount of bootstrapping are important because they will typically perform better than either extreme.

Our focus in this chapter has been on n -step methods, which look ahead to the next n rewards, states, and actions. The two 4-step backup diagrams to the right together summarize most of the methods introduced. The state-value update shown is for n -step TD with importance sampling, and the action-value update is for n -step $Q(\sigma)$, which generalizes Expected Sarsa and Q-learning. All n -step methods involve a delay of n time steps before updating, as only then are all the required future events known. A further drawback is that they involve more computation per time step than previous methods. Compared to one-step methods, n -step methods also require more memory to record the states, actions, rewards, and sometimes other variables over the last n time steps. Eventually, in Chapter 12, we will see how multi-step TD methods can be implemented with minimal memory and computational complexity using eligibility traces, but there will always be some additional computation beyond one-step methods. Such costs can be well worth paying to escape the tyranny of the single time step.

Although n -step methods are more complex than those using eligibility traces, they have the great benefit of being conceptually clear. We have sought to take advantage of this by developing two approaches to off-policy learning in the n -step case. One, based on importance sampling is conceptually simple but can be of high variance. If the target and behavior policies are very different it probably needs some new algorithmic ideas before it can be efficient and practical. The other, based on tree-backup updates, is the natural extension of Q-learning to the multi-step case with stochastic target policies. It involves no importance sampling but, again if the target and behavior policies are substantially different, the bootstrapping may span only a few steps even if n is large.



Bibliographical and Historical Remarks

The notion of *n*-step returns is due to Watkins (1989), who also first discussed their error reduction property. *n*-step algorithms were explored in the first edition of this book, in which they were treated as of conceptual interest, but not feasible in practice. The work of Cichosz (1995) and particularly van Seijen (2016) showed that they are actually completely practical algorithms. Given this, and their conceptual clarity and simplicity, we have chosen to highlight them here in the second edition. In particular, we now postpone all discussion of the backward view and of eligibility traces until Chapter 12.

7.1–2 The results in the random walk examples were made for this text based on work of Sutton (1988) and Singh and Sutton (1996). The use of backup diagrams to describe these and other algorithms in this chapter is new.

7.3–5 The developments in these sections are based on the work of Precup, Sutton, and Singh (2000), Precup, Sutton, and Dasgupta (2001), and Sutton, Mahmood, Precup, and van Hasselt (2014).

The tree-backup algorithm is due to Precup, Sutton, and Singh (2000), but the presentation of it here is new.

7.6 The $Q(\sigma)$ algorithm is new to this text, but closely related algorithms have been explored further by De Asis, Hernandez-Garcia, Holland, and Sutton (2017).

Chapter 8

Planning and Learning with Tabular Methods

In this chapter we develop a unified view of reinforcement learning methods that require a model of the environment, such as dynamic programming and heuristic search, and methods that can be used without a model, such as Monte Carlo and temporal-difference methods. These are respectively called *model-based* and *model-free* reinforcement learning methods. Model-based methods rely on *planning* as their primary component, while model-free methods primarily rely on *learning*. Although there are real differences between these two kinds of methods, there are also great similarities. In particular, the heart of both kinds of methods is the computation of value functions. Moreover, all the methods are based on looking ahead to future events, computing a backed-up value, and then using it as an update target for an approximate value function. Earlier in this book we presented Monte Carlo and temporal-difference methods as distinct alternatives, then showed how they can be unified by n -step methods. Our goal in this chapter is a similar integration of model-based and model-free methods. Having established these as distinct in earlier chapters, we now explore the extent to which they can be intermixed.

8.1 Models and Planning

By a *model* of the environment we mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call *distribution models*. Other models produce just one of the possibilities, sampled according to the probabilities; these we call *sample models*. For example, consider modeling the sum of a dozen dice. A distribution model would produce all possible sums and their probabilities of occurring, whereas a sample model would produce an individual

sum drawn according to this probability distribution. The kind of model assumed in dynamic programming—estimates of the MDP’s dynamics, $p(s', r | s, a)$ —is a distribution model. The kind of model used in the blackjack example in Chapter 5 is a sample model. Distribution models are stronger than sample models in that they can always be used to produce samples. However, in many applications it is much easier to obtain sample models than distribution models. The dozen dice are a simple example of this. It would be easy to write a computer program to simulate the dice rolls and return the sum, but harder and more error-prone to figure out all the possible sums and their probabilities.

Models can be used to mimic or simulate experience. Given a starting state and action, a sample model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, we say the model is used to *simulate* the environment and produce *simulated experience*.

The word *planning* is used in several different ways in different fields. We use the term to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment:

$$\text{model} \xrightarrow{\text{planning}} \text{policy}$$

In artificial intelligence, there are two distinct approaches to planning according to our definition. *State-space planning*, which includes the approach we take in this book, is viewed primarily as a search through the state space for an optimal policy or an optimal path to a goal. Actions cause transitions from state to state, and value functions are computed over states. In what we call *plan-space planning*, planning is instead a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. Plan-space planning includes evolutionary methods and “partial-order planning,” a common kind of planning in artificial intelligence in which the ordering of steps is not completely determined at all stages of planning. Plan-space methods are difficult to apply efficiently to the stochastic sequential decision problems that are the focus in reinforcement learning, and we do not consider them further (but see, e.g., Russell and Norvig, 2010).

The unified view we present in this chapter is that all state-space planning methods share a common structure, a structure that is also present in the learning methods presented in this book. It takes the rest of the chapter to develop this view, but there are two basic ideas: (1) all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy, and (2) they compute value functions by updates or backup operations applied to simulated experience. This common structure can be diagrammed as follows:

$$\text{model} \longrightarrow \text{simulated experience} \xrightarrow{\text{backups}} \text{values} \longrightarrow \text{policy}$$

Dynamic programming methods clearly fit this structure: they make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value (update target) and update the

state's estimated value. In this chapter we argue that various other state-space planning methods also fit this structure, with individual methods differing only in the kinds of updates they do, the order in which they do them, and in how long the backed-up information is retained.

Viewing planning methods in this way emphasizes their relationship to the learning methods that we have described in this book. The heart of both learning and planning methods is the estimation of value functions by backing-up update operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment. Of course this difference leads to a number of other differences, for example, in how performance is assessed and in how flexibly experience can be generated. But the common structure means that many ideas and algorithms can be transferred between planning and learning. In particular, in many cases a learning algorithm can be substituted for the key update step of a planning method. Learning methods require only experience as input, and in many cases they can be applied to simulated experience just as well as to real experience. The box below shows a simple example of a planning method based on one-step tabular Q-learning and on random samples from a sample model. This method, which we call *random-sample one-step tabular Q-planning*, converges to the optimal policy for the model under the same conditions that one-step tabular Q-learning converges to the optimal policy for the real environment (each state-action pair must be selected an infinite number of times in Step 1, and α must decrease appropriately over time).

Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain
a sample next reward, R , and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S' :

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

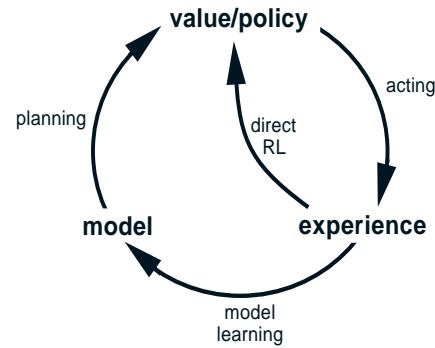
In addition to the unified view of planning and learning methods, a second theme in this chapter is the benefits of planning in small, incremental steps. This enables planning to be interrupted or redirected at any time with little wasted computation, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. Planning in very small steps may be the most efficient approach even on pure planning problems if the problem is too large to be solved exactly.

8.2 Dyna: Integrated Planning, Acting, and Learning

When planning is done online, while interacting with the environment, a number of interesting issues arise. New information gained from the interaction may change the model and thereby interact with planning. It may be desirable to customize the planning process in some way to the states or decisions currently under consideration, or expected

in the near future. If decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them. To begin exploring these issues, in this section we present Dyna-Q, a simple architecture integrating the major functions needed in an online planning agent. Each function appears in Dyna-Q in a simple, almost trivial, form. In subsequent sections we elaborate some of the alternate ways of achieving each function and the trade-offs between them. For now, we seek merely to illustrate the ideas and stimulate your intuition.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (to make it more accurately match the real environment) and it can be used to directly improve the value function and policy using the kinds of reinforcement learning methods we have discussed in previous chapters. The former we call *model-learning*, and the latter we call *direct reinforcement learning* (direct RL). The possible relationships between experience, model, values, and policy are summarized in the diagram to the right. Each arrow shows a relationship of influence and presumed improvement. Note how experience can improve value functions and policies either directly or indirectly via the model. It is the latter, which is sometimes called *indirect reinforcement learning*, that is involved in planning.



Both direct and indirect methods have advantages and disadvantages. Indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, direct methods are much simpler and are not affected by biases in the design of the model. Some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning. Related debates in psychology and artificial intelligence concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision making (see Chapter 14 for discussion of some of these issues from the perspective of psychology). Our view is that the contrast between the alternatives in all these debates has been exaggerated, that more insight can be gained by recognizing the similarities between these two sides than by opposing them. For example, in this book we have emphasized the deep similarities between dynamic programming and temporal-difference methods, even though one was designed for planning and the other for model-free learning.

Dyna-Q includes all of the processes shown in the diagram above—planning, acting, model-learning, and direct RL—all occurring continually. The planning method is the random-sample one-step tabular Q-planning method on page 161. The direct RL method is one-step tabular Q-learning. The model-learning method is also table-based and assumes the environment is deterministic. After each transition $S_t, A_t \rightarrow R_{t+1}, S_{t+1}$, the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow. Thus, if the model is queried with a state-action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction.

During planning, the Q-planning algorithm randomly samples only from state-action pairs that have previously been experienced (in Step 1), so the model is never queried with a pair about which it has no information.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 8.1. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term *search control* to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in Dyna-Q, the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience. The reinforcement learning method is thus the “final common path” for both learning and planning. Learning and planning are deeply integrated in the sense that they share almost all the same machinery, differing only in the source of their experience.

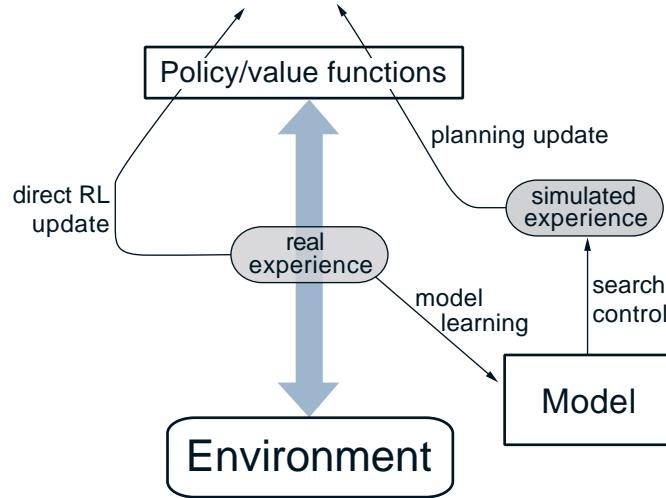


Figure 8.1: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

Conceptually, planning, acting, model-learning, and direct RL occur simultaneously and in parallel in Dyna agents. For concreteness and implementation on a serial computer, however, we fully specify the order in which they occur within a time step. In Dyna-Q, the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time. The remaining time in each step can be devoted to the planning process, which is inherently computation-intensive. Let us assume that there is time in each step, after acting, model-learning, and direct RL, to complete

n iterations (Steps 1–3) of the Q-planning algorithm. In the pseudocode algorithm for Dyna-Q in the box below, $Model(s, a)$ denotes the contents of the model (predicted next state and reward) for state-action pair (s, a) . Direct reinforcement learning, model-learning, and planning are implemented by steps (d), (e), and (f), respectively. If (e) and (f) were omitted, the remaining algorithm would be one-step tabular Q-learning.

Tabular Dyna-Q

```

Initialize  $Q(s, a)$  and  $Model(s, a)$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ 
Loop forever:
  (a)  $S \leftarrow$  current (nonterminal) state
  (b)  $A \leftarrow \varepsilon\text{-greedy}(S, Q)$ 
  (c) Take action  $A$ ; observe resultant reward,  $R$ , and state,  $S'$ 
  (d)  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
  (e)  $Model(S, A) \leftarrow R, S'$  (assuming deterministic environment)
  (f) Loop repeat  $n$  times:
     $S \leftarrow$  random previously observed state
     $A \leftarrow$  random action previously taken in  $S$ 
     $R, S' \leftarrow Model(S, A)$ 
     $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
```

Example 8.1: Dyna Maze Consider the simple maze shown inset in Figure 8.2. In each of the 47 states there are four actions, **up**, **down**, **right**, and **left**, which take the agent deterministically to the corresponding neighboring states, except when movement is blocked by an obstacle or the edge of the maze, in which case the agent remains where it is. Reward is zero on all transitions, except those into the goal state, on which it is +1. After reaching the goal state (G), the agent returns to the start state (S) to begin a new episode. This is a discounted, episodic task with $\gamma = 0.95$.

The main part of Figure 8.2 shows average learning curves from an experiment in which Dyna-Q agents were applied to the maze task. The initial action values were zero, the step-size parameter was $\alpha = 0.1$, and the exploration parameter was $\varepsilon = 0.1$. When selecting greedily among actions, ties were broken randomly. The agents varied in the number of planning steps, n , they performed per real step. For each n , the curves show the number of steps taken by the agent to reach the goal in each episode, averaged over 30 repetitions of the experiment. In each repetition, the initial seed for the random number generator was held constant across algorithms. Because of this, the first episode was exactly the same (about 1700 steps) for all values of n , and its data are not shown in the figure. After the first episode, performance improved for all values of n , but much more rapidly for larger values. Recall that the $n = 0$ agent is a nonplanning agent, using only direct reinforcement learning (one-step tabular Q-learning). This was by far the slowest agent on this problem, despite the fact that the parameter values (α and ε) were optimized for it. The nonplanning agent took about 25 episodes to reach (ε -)optimal performance, whereas the $n = 5$ agent took about five episodes, and the $n = 50$ agent took only three episodes.

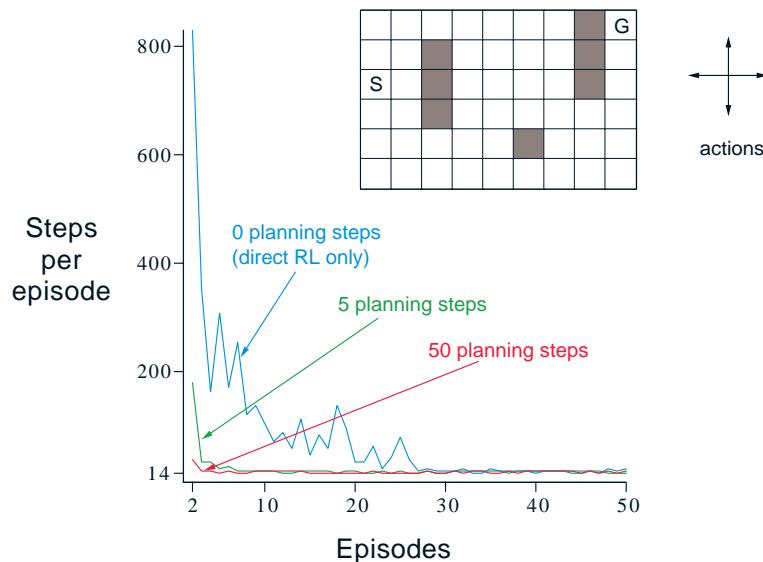


Figure 8.2: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from S to G as quickly as possible.

Figure 8.3 shows why the planning agents found the solution so much faster than the nonplanning agent. Shown are the policies found by the $n = 0$ and $n = 50$ agents halfway through the second episode. Without planning ($n = 0$), each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far. With planning, again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the end of the episode will reach almost back to the start state. This policy is built by the planning process while the agent is still wandering near the start state. By the end of the third episode a complete optimal policy will have been found and perfect performance attained.

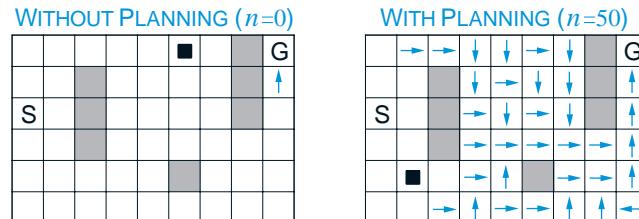


Figure 8.3: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent. ■

In Dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning. Because planning proceeds incrementally, it is trivial to intermix planning and acting. Both proceed as fast as they can. The agent is always reactive and always deliberative, responding instantly to the latest sensory information and yet always planning in the background. Also ongoing in the background is the model-learning process. As new information is gained, the model is updated to better match reality. As the model changes, the ongoing planning process will gradually compute a different way of behaving to match the new model.

Exercise 8.1 The nonplanning method looks particularly poor in Figure 8.3 because it is a one-step method; a method using multi-step bootstrapping would do better. Do you think one of the multi-step bootstrapping methods from Chapter 7 could do as well as the Dyna method? Explain why or why not. \square

8.3 When the Model Is Wrong

In the maze example presented in the previous section, the changes in the model were relatively modest. The model started out empty, and was then filled only with exactly correct information. In general, we cannot expect to be so fortunate. Models may be incorrect because the environment is stochastic and only a limited number of samples have been observed, or because the model was learned using function approximation that has generalized imperfectly, or simply because the environment has changed and its new behavior has not yet been observed. When the model is incorrect, the planning process is likely to compute a suboptimal policy.

In some cases, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error. This tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible. The planned policy attempts to exploit these opportunities and in doing so discovers that they do not exist.

Example 8.2: Blocking Maze A maze example illustrating this relatively minor kind of modeling error and recovery from it is shown in Figure 8.4. Initially, there is a short path from start to goal, to the right of the barrier, as shown in the upper left of the figure. After 1000 time steps, the short path is “blocked,” and a longer path is opened up along the left-hand side of the barrier, as shown in upper right of the figure. The graph shows average cumulative reward for a Dyna-Q agent and an enhanced Dyna-Q+ agent to be described shortly. The first part of the graph shows that both Dyna agents found the short path within 1000 steps. When the environment changed, the graphs become flat, indicating a period during which the agents obtained no reward because they were wandering around behind the barrier. After a while, however, they were able to find the new opening and the new optimal behavior.

Greater difficulties arise when the environment changes to become *better* than it was before, and yet the formerly correct policy does not reveal the improvement. In these cases the modeling error may not be detected for a long time, if ever.

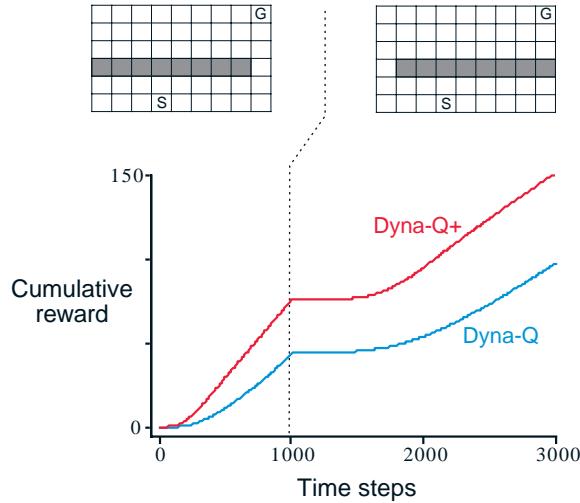


Figure 8.4: Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration. ■

Example 8.3: Shortcut Maze

The problem caused by this kind of environmental change is illustrated by the maze example shown in Figure 8.5. Initially, the optimal path is to go around the left side of the barrier (upper left). After 3000 steps, however, a shorter path is opened up along the right side, without disturbing the longer path (upper right). The graph shows that the regular Dyna-Q agent never switched to the shortcut. In fact, it never realized that it existed. Its model said that there was no shortcut, so the more it planned, the less likely it was to step to the right and discover it. Even with an ϵ -greedy policy, it is very unlikely that an agent will take so many exploratory actions as to discover the shortcut.

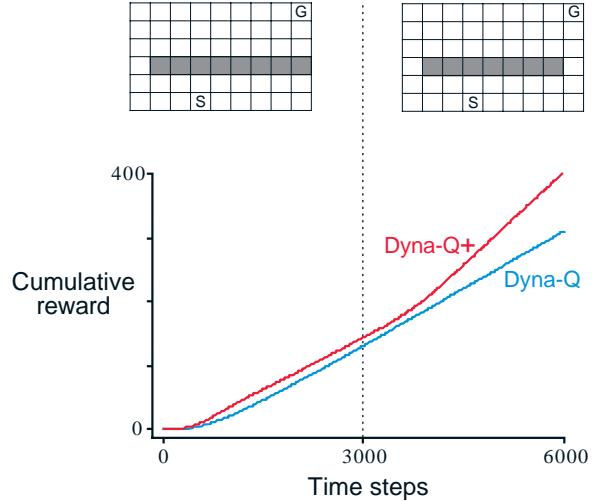


Figure 8.5: Average performance of Dyna agents on a shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest. ■

The general problem here is another version of the conflict between exploration and exploitation. In a planning context, exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model.

We want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded. As in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective.

The Dyna-Q+ agent that did solve the shortcut maze uses one such heuristic. This agent keeps track for each state-action pair of how many time steps have elapsed since the pair was last tried in a real interaction with the environment. The more time that has elapsed, the greater (we might presume) the chance that the dynamics of this pair has changed and that the model of it is incorrect. To encourage behavior that tests long-untried actions, a special “bonus reward” is given on simulated experiences involving these actions. In particular, if the modeled reward for a transition is r , and the transition has not been tried in τ time steps, then planning updates are done as if that transition produced a reward of $r + \kappa\sqrt{\tau}$, for some small κ . This encourages the agent to keep testing all accessible state transitions and even to find long sequences of actions in order to carry out such tests.¹ Of course all this testing has its cost, but in many cases, as in the shortcut maze, this kind of computational curiosity is well worth the extra exploration.

Exercise 8.2 Why did the Dyna agent with exploration bonus, Dyna-Q+, perform better in the first phase as well as in the second phase of the blocking and shortcut experiments? \square

Exercise 8.3 Careful inspection of Figure 8.5 reveals that the difference between Dyna-Q+ and Dyna-Q narrowed slightly over the first part of the experiment. What is the reason for this? \square

Exercise 8.4 (programming) The exploration bonus described above actually changes the estimated values of states and actions. Is this necessary? Suppose the bonus $\kappa\sqrt{\tau}$ was used not in updates, but solely in action selection. That is, suppose the action selected was always that for which $Q(S_t, a) + \kappa\sqrt{\tau(S_t, a)}$ was maximal. Carry out a gridworld experiment that tests and illustrates the strengths and weaknesses of this alternate approach. \square

Exercise 8.5 How might the tabular Dyna-Q algorithm shown on page 164 be modified to handle stochastic environments? How might this modification perform poorly on changing environments such as considered in this section? How could the algorithm be modified to handle stochastic environments *and* changing environments? \square

8.4 Prioritized Sweeping

In the Dyna agents presented in the preceding sections, simulated transitions are started in state-action pairs selected uniformly at random from all previously experienced pairs. But a uniform selection is usually not the best; planning can be much more efficient if simulated transitions and updates are focused on particular state-action pairs. For example, consider

¹The Dyna-Q+ agent was changed in two other ways as well. First, actions that had never been tried before from a state were allowed to be considered in the planning step (f) of the Tabular Dyna-Q algorithm in the box above. Second, the initial model for such actions was that they would lead back to the same state with a reward of zero.

what happens during the second episode of the first maze task (Figure 8.3). At the beginning of the second episode, only the state-action pair leading directly into the goal has a positive value; the values of all other pairs are still zero. This means that it is pointless to perform updates along almost all transitions, because they take the agent from one zero-valued state to another, and thus the updates would have no effect. Only an update along a transition into the state just prior to the goal, or from it, will change any values. If simulated transitions are generated uniformly, then many wasteful updates will be made before stumbling onto one of these useful ones. As planning progresses, the region of useful updates grows, but planning is still far less efficient than it would be if focused where it would do the most good. In the much larger problems that are our real objective, the number of states is so large that an unfocused search would be extremely inefficient.

This example suggests that search might be usefully focused by working *backward* from goal states. Of course, we do not really want to use any methods specific to the idea of “goal state.” We want methods that work for general reward functions. Goal states are just a special case, convenient for stimulating intuition. In general, we want to work back not just from goal states but from any state whose value has changed. Suppose that the values are initially correct given the model, as they were in the maze example prior to discovering the goal. Suppose now that the agent discovers a change in the environment and changes its estimated value of one state, either up or down. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step updates are those of actions that lead directly into the one state whose value has been changed. If the values of these actions are updated, then the values of the predecessor states may change in turn. If so, then actions leading into them need to be updated, and then *their* predecessor states may have changed. In this way one can work backward from arbitrary states that have changed in value, either performing useful updates or terminating the propagation. This general idea might be termed *backward focusing* of planning computations.

As the frontier of useful updates propagates backward, it often grows rapidly, producing many state-action pairs that could usefully be updated. But not all of these will be equally useful. The values of some states may have changed a lot, whereas others may have changed little. The predecessor pairs of those that have changed a lot are more likely to also change a lot. In a stochastic environment, variations in estimated transition probabilities also contribute to variations in the sizes of changes and in the urgency with which pairs need to be updated. It is natural to prioritize the updates according to a measure of their urgency, and perform them in order of priority. This is the idea behind *prioritized sweeping*. A queue is maintained of every state-action pair whose estimated value would change nontrivially if updated, prioritized by the size of the change. When the top pair in the queue is updated, the effect on each of its predecessor pairs is computed. If the effect is greater than some small threshold, then the pair is inserted in the queue with the new priority (if there is a previous entry of the pair in the queue, then insertion results in only the higher priority entry remaining in the queue). In this way the effects of changes are efficiently propagated backward until quiescence. The full algorithm for the case of deterministic environments is given in the box on the next page.

Prioritized sweeping for a deterministic environment

Initialize $Q(s, a)$, $Model(s, a)$, for all s, a , and $PQueue$ to empty

Loop forever:

- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow policy(S, Q)$
- (c) Take action A ; observe resultant reward, R , and state, S'
- (d) $Model(S, A) \leftarrow R, S'$
- (e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|.$
- (f) if $P > \theta$, then insert S, A into $PQueue$ with priority P

(g) Loop repeat n times, while $PQueue$ is not empty:

$$S, A \leftarrow first(PQueue)$$

$$R, S' \leftarrow Model(S, A)$$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

Loop for all \bar{S}, \bar{A} predicted to lead to S :

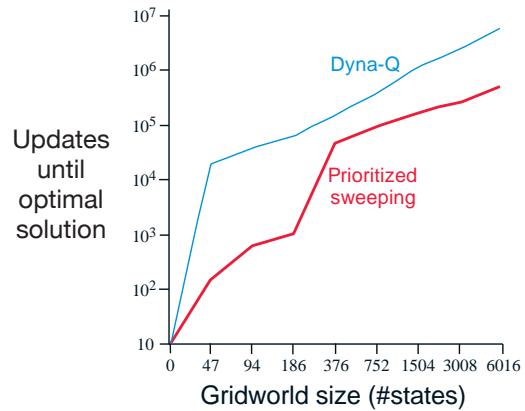
$$\bar{R} \leftarrow \text{predicted reward for } \bar{S}, \bar{A}, S$$

$$P \leftarrow |\bar{R} + \gamma \max_a Q(\bar{S}, a) - Q(\bar{S}, \bar{A})|.$$

if $P > \theta$ then insert \bar{S}, \bar{A} into $PQueue$ with priority P

Example 8.4: Prioritized Sweeping

on Mazes Prioritized sweeping has been found to dramatically increase the speed at which optimal solutions are found in maze tasks, often by a factor of 5 to 10. A typical example is shown to the right. These data are for a sequence of maze tasks of exactly the same structure as the one shown in Figure 8.2, except that they vary in the grid resolution. Prioritized sweeping maintained a decisive advantage over unprioritized Dyna-Q. Both systems made at most $n = 5$ updates per environmental interaction. Adapted from Peng and Williams (1993).

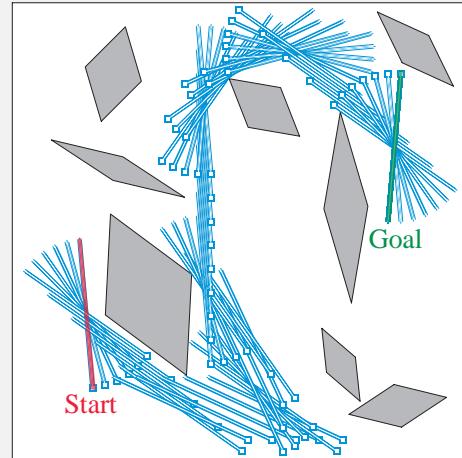


Extensions of prioritized sweeping to stochastic environments are straightforward. The model is maintained by keeping counts of the number of times each state-action pair has been experienced and of what the next states were. It is natural then to update each pair not with a sample update, as we have been using so far, but with an expected update, taking into account all possible next states and their probabilities of occurring.

Prioritized sweeping is just one way of distributing computations to improve planning efficiency, and probably not the best way. One of prioritized sweeping's limitations is that it uses *expected* updates, which in stochastic environments may waste lots of computation on low-probability transitions. As we show in the following section, sample updates

Example 8.5 Prioritized Sweeping for Rod Maneuvering

The objective in this task is to maneuver a rod around some awkwardly placed obstacles within a limited rectangular work space to a goal position in the fewest number of steps. The rod can be translated along its long axis or perpendicular to that axis, or it can be rotated in either direction around its center. The distance of each movement is approximately 1/20 of the work space, and the rotation increment is 10 degrees. Translations are deterministic and quantized to one of 20×20 positions. To the right is shown the obstacles and the shortest solution from start to goal, found by prioritized sweeping. This problem is deterministic, but has four actions and 14,400 potential states (some of these are unreachable because of the obstacles). This problem is probably too large to be solved with unprioritized methods. Figure reprinted from Moore and Atkeson (1993).



can in many cases get closer to the true value function with less computation despite the variance introduced by sampling. Sample updates can win because they break the overall backing-up computation into smaller pieces—those corresponding to individual transitions—which then enables it to be focused more narrowly on the pieces that will have the largest impact. This idea was taken to what may be its logical limit in the “small backups” introduced by van Seijen and Sutton (2013). These are updates along a single transition, like a sample update, but based on the probability of the transition without sampling, as in an expected update. By selecting the order in which small updates are done it is possible to greatly improve planning efficiency beyond that possible with prioritized sweeping.

We have suggested in this chapter that all kinds of state-space planning can be viewed as sequences of value updates, varying only in the type of update, expected or sample, large or small, and in the order in which the updates are done. In this section we have emphasized backward focusing, but this is just one strategy. For example, another would be to focus on states according to how easily they can be reached from the states that are visited frequently under the current policy, which might be called *forward focusing*. Peng and Williams (1993) and Barto, Bradtke and Singh (1995) have explored versions of forward focusing, and the methods introduced in the next few sections take it to an extreme form.

8.5 Expected vs. Sample Updates

The examples in the previous sections give some idea of the range of possibilities for combining methods of learning and planning. In the rest of this chapter, we analyze some of the component ideas involved, starting with the relative advantages of expected and sample updates.

Much of this book has been about different kinds of value-function updates, and we have considered a great many varieties. Focusing for the moment on one-step updates, they vary primarily along three binary dimensions. The first two dimensions are whether they update state values or action values and whether they estimate the value for the optimal policy or for an arbitrary given policy. These two dimensions give rise to four classes of updates for approximating the four value functions, q_* , v_* , q_π , and v_π . The other binary dimension is whether the updates are *expected* updates, considering all possible events that might happen, or *sample* updates, considering a single sample of what might happen. These three binary dimensions give rise to eight cases, seven of which correspond to specific algorithms, as shown in the figure to the right. (The eighth case does not seem to correspond to any useful update.) Any of these one-step updates can be used in planning methods. The Dyna-Q agents discussed earlier use q_* sample updates, but they could just as well use q_* expected updates, or either expected or sample q_π updates. The Dyna-AC system uses v_π sample updates together with a learning policy structure (as in Chapter 13). For stochastic problems, prioritized sweeping is always done using one of the expected updates.

When we introduced one-step sample updates in Chapter 6, we presented them as substitutes for expected updates. In the absence of a distribution model, expected updates are not possible, but sample updates can be done using sample transitions from the environment or a sample model. Implicit in that point of view is that expected updates, if possible, are preferable to sample updates. But are they? Expected

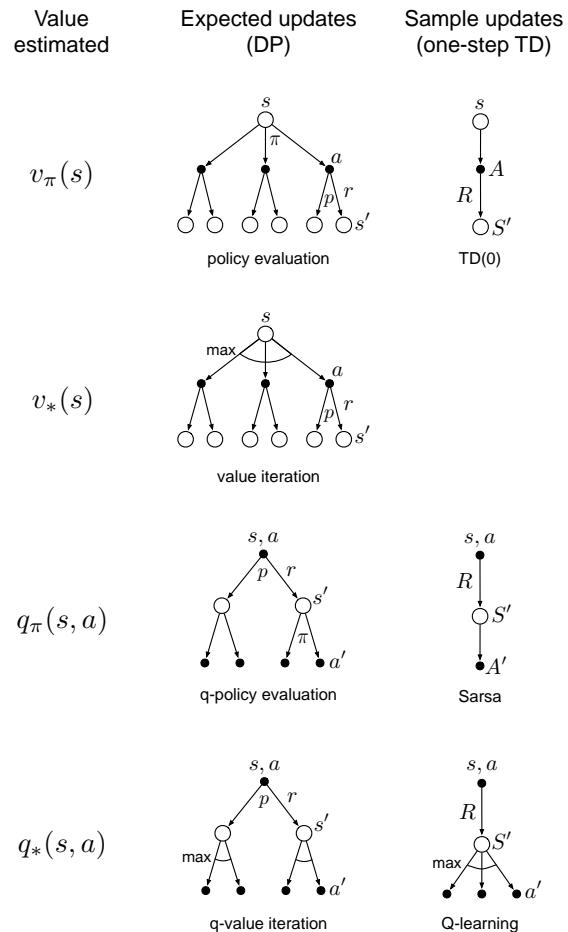


Figure 8.6: Backup diagrams for all the one-step updates considered in this book.

updates certainly yield a better estimate because they are uncorrupted by sampling error, but they also require more computation, and computation is often the limiting resource in planning. To properly assess the relative merits of expected and sample updates for planning we must control for their different computational requirements.

For concreteness, consider the expected and sample updates for approximating q_* , and the special case of discrete states and actions, a table-lookup representation of the approximate value function, Q , and a model in the form of estimated dynamics, $\hat{p}(s', r | s, a)$. The expected update for a state-action pair, s, a , is:

$$Q(s, a) \leftarrow \sum_{s', r} \hat{p}(s', r | s, a) \left[r + \gamma \max_{a'} Q(s', a') \right]. \quad (8.1)$$

The corresponding sample update for s, a , given a sample next state and reward, S' and R (from the model), is the Q-learning-like update:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(S', a') - Q(s, a) \right], \quad (8.2)$$

where α is the usual positive step-size parameter.

The difference between these expected and sample updates is significant to the extent that the environment is stochastic, specifically, to the extent that, given a state and action, many possible next states may occur with various probabilities. If only one next state is possible, then the expected and sample updates given above are identical (taking $\alpha = 1$). If there are many possible next states, then there may be significant differences. In favor of the expected update is that it is an exact computation, resulting in a new $Q(s, a)$ whose correctness is limited only by the correctness of the $Q(s', a')$ at successor states. The sample update is in addition affected by sampling error. On the other hand, the sample update is cheaper computationally because it considers only one next state, not all possible next states. In practice, the computation required by update operations is usually dominated by the number of state-action pairs at which Q is evaluated. For a particular starting pair, s, a , let b be the *branching factor* (i.e., the number of possible next states, s' , for which $\hat{p}(s' | s, a) > 0$). Then an expected update of this pair requires roughly b times as much computation as a sample update.

If there is enough time to complete an expected update, then the resulting estimate is generally better than that of b sample updates because of the absence of sampling error. But if there is insufficient time to complete an expected update, then sample updates are always preferable because they at least make some improvement in the value estimate with fewer than b updates. In a large problem with many state-action pairs, we are often in the latter situation. With so many state-action pairs, expected updates of all of them would take a very long time. Before that we may be much better off with a few sample updates at many state-action pairs than with expected updates at a few pairs. Given a unit of computational effort, is it better devoted to a few expected updates or to b times as many sample updates?

Figure 8.7 shows the results of an analysis that suggests an answer to this question. It shows the estimation error as a function of computation time for expected and sample updates for a variety of branching factors, b . The case considered is that in which all

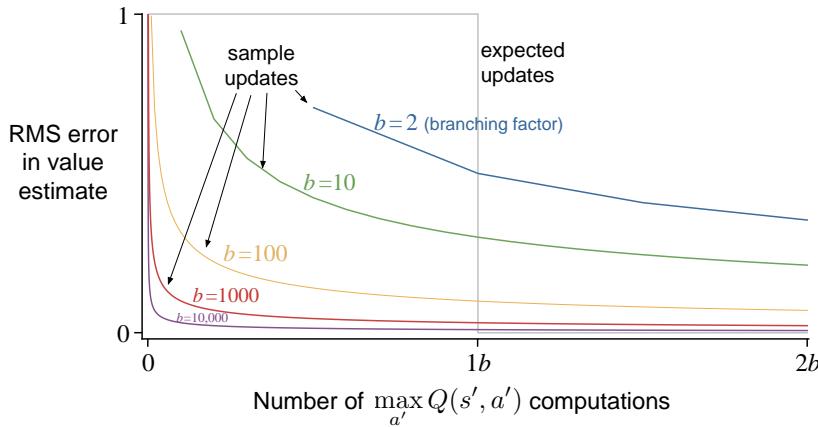


Figure 8.7: Comparison of efficiency of expected and sample updates.

b successor states are equally likely and in which the error in the initial estimate is 1. The values at the next states are assumed correct, so the expected update reduces the error to zero upon its completion. In this case, sample updates reduce the error according to $\sqrt{\frac{b-1}{bt}}$ where t is the number of sample updates that have been performed (assuming sample averages, i.e., $\alpha = 1/t$). The key observation is that for moderately large b the error falls dramatically with a tiny fraction of b updates. For these cases, many state-action pairs could have their values improved dramatically, to within a few percent of the effect of an expected update, in the same time that a single state-action pair could undergo an expected update.

The advantage of sample updates shown in Figure 8.7 is probably an underestimate of the real effect. In a real problem, the values of the successor states would be estimates that are themselves updated. By causing estimates to be more accurate sooner, sample updates will have a second advantage in that the values backed up from the successor states will be more accurate. These results suggest that sample updates are likely to be superior to expected updates on problems with large stochastic branching factors and too many states to be solved exactly.

Exercise 8.6 The analysis above assumed that all of the b possible next states were equally likely to occur. Suppose instead that the distribution was highly skewed, that some of the b states were much more likely to occur than most. Would this strengthen or weaken the case for sample updates over expected updates? Support your answer. \square

8.6 Trajectory Sampling

In this section we compare two ways of distributing updates. The classical approach, from dynamic programming, is to perform sweeps through the entire state (or state-action) space, updating each state (or state-action pair) once per sweep. This is problematic

on large tasks because there may not be time to complete even one sweep. In many tasks the vast majority of the states are irrelevant because they are visited only under very poor policies or with very low probability. Exhaustive sweeps implicitly devote equal time to all parts of the state space rather than focusing where it is needed. As we discussed in Chapter 4, exhaustive sweeps and the equal treatment of all states that they imply are not necessary properties of dynamic programming. In principle, updates can be distributed any way one likes (to assure convergence, all states or state-action pairs must be visited in the limit an infinite number of times; although an exception to this is discussed in Section 8.7 below), but in practice exhaustive sweeps are often used.

The second approach is to sample from the state or state-action space according to some distribution. One could sample uniformly, as in the Dyna-Q agent, but this would suffer from some of the same problems as exhaustive sweeps. More appealing is to distribute updates according to the on-policy distribution, that is, according to the distribution observed when following the current policy. One advantage of this distribution is that it is easily generated; one simply interacts with the model, following the current policy. In an episodic task, one starts in a start state (or according to the starting-state distribution) and simulates until the terminal state. In a continuing task, one starts anywhere and just keeps simulating. In either case, sample state transitions and rewards are given by the model, and sample actions are given by the current policy. In other words, one simulates explicit individual trajectories and performs updates at the state or state-action pairs encountered along the way. We call this way of generating experience and updates *trajectory sampling*.

It is hard to imagine any efficient way of distributing updates according to the on-policy distribution other than by trajectory sampling. If one had an explicit representation of the on-policy distribution, then one could sweep through all states, weighting the update of each according to the on-policy distribution, but this leaves us again with all the computational costs of exhaustive sweeps. Possibly one could sample and update individual state-action pairs from the distribution, but even if this could be done efficiently, what benefit would this provide over simulating trajectories? Even knowing the on-policy distribution in an explicit form is unlikely. The distribution changes whenever the policy changes, and computing the distribution requires computation comparable to a complete policy evaluation. Consideration of such other possibilities makes trajectory sampling seem both efficient and elegant.

Is the on-policy distribution of updates a good one? Intuitively it seems like a good choice, at least better than the uniform distribution. For example, if you are learning to play chess, you study positions that might arise in real games, not random positions of chess pieces. The latter may be valid states, but to be able to accurately value them is a different skill from evaluating positions in real games. We will also see in Part II that the on-policy distribution has significant advantages when function approximation is used. Whether or not function approximation is used, one might expect on-policy focusing to significantly improve the speed of planning.

Focusing on the on-policy distribution could be beneficial because it causes vast, uninteresting parts of the space to be ignored, or it could be detrimental because it causes the same old parts of the space to be updated over and over. We conducted a small

experiment to assess the effect empirically. To isolate the effect of the update distribution, we used entirely one-step expected tabular updates, as defined by (8.1). In the *uniform* case, we cycled through all state-action pairs, updating each in place, and in the *on-policy* case we simulated episodes, all starting in the same state, updating each state-action pair that occurred under the current ε -greedy policy ($\varepsilon=0.1$). The tasks were undiscounted episodic tasks, generated randomly as follows. From each of the $|\mathcal{S}|$ states, two actions were possible, each of which resulted in one of b next states, all equally likely, with a different random selection of b states for each state-action pair. The branching factor, b , was the same for all state-action pairs. In addition, on all transitions there was a 0.1 probability of transition to the terminal state, ending the episode. The expected reward on each transition was selected from a Gaussian distribution with mean 0 and variance 1.

At any point in the planning process one can stop and exhaustively compute $v_{\tilde{\pi}}(s_0)$, the true value of the start state under the greedy policy, $\tilde{\pi}$, given the current action-value function Q , as an indication of how well the agent would do on a new episode on which it acted greedily (all the while assuming the model is correct).

The upper part of the figure to the right shows results averaged over 200 sample tasks with 1000 states and branching factors of 1, 3, and 10. The quality of the policies found is plotted as a function of the number of expected updates completed. In all cases, sampling according to the on-policy distribution resulted in faster planning initially and retarded planning in the long run. The effect was stronger, and the initial period of faster planning was longer, at smaller branching factors. In other experiments, we found that these effects also became stronger as the number of states increased. For example, the lower part of the figure shows results for a branching factor of 1 for tasks with 10,000 states. In this case the advantage of on-policy focusing is large and long-lasting.

All of these results make sense. In the short term, sampling according to the on-policy distribution helps by focusing on states that are near descendants of

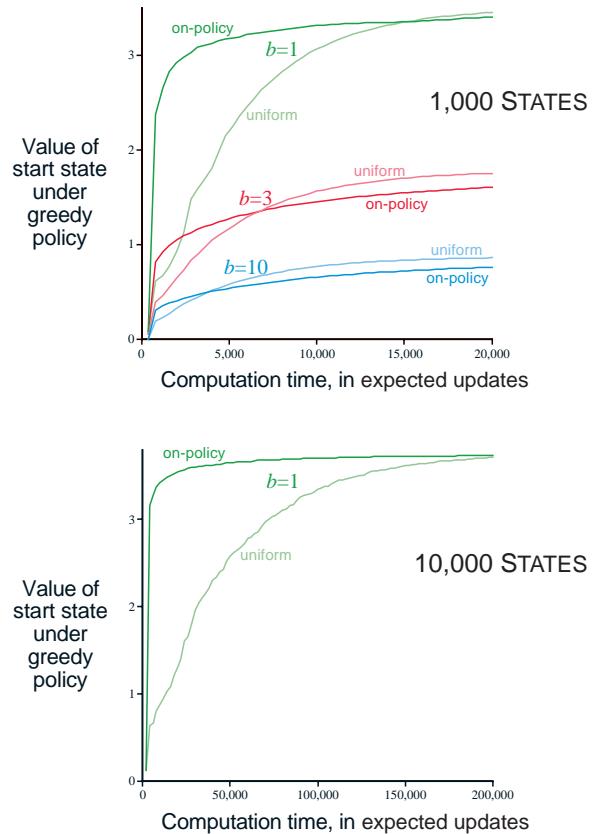


Figure 8.8: Relative efficiency of updates distributed uniformly across the state space versus focused on simulated on-policy trajectories, each starting in the same state. Results are for randomly generated tasks of two sizes and various branching factors, b .

the start state. If there are many states and a small branching factor, this effect will be large and long-lasting. In the long run, focusing on the on-policy distribution may hurt because the commonly occurring states all already have their correct values. Sampling them is useless, whereas sampling other states may actually perform some useful work. This presumably is why the exhaustive, unfocused approach does better in the long run, at least for small problems. These results are not conclusive because they are only for problems generated in a particular, random way, but they do suggest that sampling according to the on-policy distribution can be a great advantage for large problems, in particular for problems in which a small subset of the state-action space is visited under the on-policy distribution.

Exercise 8.7 Some of the graphs in Figure 8.8 seem to be scalloped in their early portions, particularly the upper graph for $b = 1$ and the uniform distribution. Why do you think this is? What aspects of the data shown support your hypothesis? \square

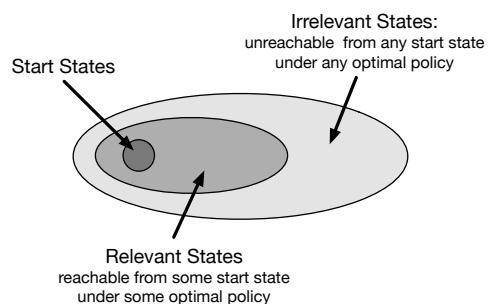
Exercise 8.8 (programming) Replicate the experiment whose results are shown in the lower part of Figure 8.8, then try the same experiment but with $b = 3$. Discuss the meaning of your results. \square

8.7 Real-time Dynamic Programming

Real-time dynamic programming, or RTDP, is an on-policy trajectory-sampling version of the value-iteration algorithm of dynamic programming (DP). Because it is closely related to conventional sweep-based policy iteration, RTDP illustrates in a particularly clear way some of the advantages that on-policy trajectory sampling can provide. RTDP updates the values of states visited in actual or simulated trajectories by means of expected tabular value-iteration updates as defined by (4.10). It is basically the algorithm that produced the on-policy results shown in Figure 8.8.

The close connection between RTDP and conventional DP makes it possible to derive some theoretical results by adapting existing theory. RTDP is an example of an *asynchronous* DP algorithm as described in Section 4.5. Asynchronous DP algorithms are not organized in terms of systematic sweeps of the state set; they update state values in any order whatsoever, using whatever values of other states happen to be available. In RTDP, the update order is dictated by the order states are visited in real or simulated trajectories.

If trajectories can start only from a designated set of start states, and if you are interested in the prediction problem for a given policy, then on-policy trajectory sampling allows the algorithm to completely skip states that cannot be reached by the given policy from any of the start states: such states are *irrelevant* to the prediction problem. For a control problem, where the goal is to find an optimal policy instead of evaluating a given policy, there might well be states that cannot be



reached by any optimal policy from any of the start states, and there is no need to specify optimal actions for these irrelevant states. What is needed is an *optimal partial policy*, meaning a policy that is optimal for the relevant states but can specify arbitrary actions, or even be undefined, for the irrelevant states.

But *finding* such an optimal partial policy with an on-policy trajectory-sampling control method, such as Sarsa (Section 6.4), in general requires visiting all state-action pairs—even those that will turn out to be irrelevant—an infinite number of times. This can be done, for example, by using exploring starts (Section 5.3). This is true for RTDP as well: for episodic tasks with exploring starts, RTDP is an asynchronous value-iteration algorithm that converges to optimal policies for discounted finite MDPs (and for the undiscounted case under certain conditions). Unlike the situation for a prediction problem, it is generally not possible to stop updating any state or state-action pair if convergence to an optimal policy is important.

The most interesting result for RTDP is that for certain types of problems satisfying reasonable conditions, RTDP is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all. Indeed, in some problems, only a small fraction of the states need to be visited. This can be a great advantage for problems with very large state sets, where even a single sweep may not be feasible.

The tasks for which this result holds are undiscounted episodic tasks for MDPs with absorbing goal states that generate zero rewards, as described in Section 3.4. At every step of a real or simulated trajectory, RTDP selects a greedy action (breaking ties randomly) and applies the expected value-iteration update operation to the current state. It can also update the values of an arbitrary collection of other states at each step; for example, it can update the values of states visited in a limited-horizon look-ahead search from the current state.

For these problems, with each episode beginning in a state randomly chosen from the set of start states and ending at a goal state, RTDP converges with probability one to a policy that is optimal for all the relevant states provided: (1) the initial value of every goal state is zero, (2) there exists at least one policy that guarantees that a goal state will be reached with probability one from any start state, (3) all rewards for transitions from non-goal states are strictly negative, and (4) all the initial values are equal to, or greater than, their optimal values (which can be satisfied by simply setting the initial values of all states to zero). This result was proved by Barto, Bradtke, and Singh (1995) by combining results for asynchronous DP with results about a heuristic search algorithm known as *learning real-time A** due to Korf (1990).

Tasks having these properties are examples of *stochastic optimal path problems*, which are usually stated in terms of cost minimization instead of as reward maximization as we do here. Maximizing the negative returns in our version is equivalent to minimizing the costs of paths from a start state to a goal state. Examples of this kind of task are minimum-time control tasks, where each time step required to reach a goal produces a reward of -1 , or problems like the Golf example in Section 3.5, whose objective is to hit the hole with the fewest strokes.

Example 8.6: RTDP on the Racetrack The racetrack problem of Exercise 5.12 (page 111) is a stochastic optimal path problem. Comparing RTDP and the conventional DP value iteration algorithm on an example racetrack problem illustrates some of the advantages of on-policy trajectory sampling.

Recall from the exercise that an agent has to learn how to drive a car around a turn like those shown in Figure 5.5 and cross the finish line as quickly as possible while staying on the track. Start states are all the zero-speed states on the starting line; the goal states are all the states that can be reached in one time step by crossing the finish line from inside the track. Unlike Exercise 5.12, here there is no limit on the car’s speed, so the state set is potentially infinite. However, the set of states that can be reached from the set of start states via any policy is finite and can be considered to be the state set of the problem. Each episode begins in a randomly selected start state and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random start state, and the episode continues.

A racetrack similar to the small racetrack on the left of Figure 5.5 has 9,115 states reachable from start states by any policy, only 599 of which are relevant, meaning that they are reachable from some start state via some optimal policy. (The number of relevant states was estimated by counting the states visited while executing optimal actions for 10^7 episodes.)

The table below compares solving this task by conventional DP and by RTDP. These results are averages over 25 runs, each begun with a different random number seed. Conventional DP in this case is value iteration using exhaustive sweeps of the state set, with values updated one state at a time in place, meaning that the update for each state uses the most recent values of the other states (This is the Gauss-Seidel version of value iteration, which was found to be approximately twice as fast as the Jacobi version on this problem. See Section 4.8.) No special attention was paid to the ordering of the updates; other orderings could have produced faster convergence. Initial values were all zero for each run of both methods. DP was judged to have converged when the maximum change in a state value over a sweep was less than 10^{-4} , and RTDP was judged to have converged when the average time to cross the finish line over 20 episodes appeared to stabilize at an asymptotic number of steps. This version of RTDP updated only the value of the current state on each step.

	DP	RTDP
Average computation to convergence	28 sweeps	4000 episodes
Average number of updates to convergence	252,784	127,600
Average number of updates per episode	—	31.9
% of states updated ≤ 100 times	—	98.45
% of states updated ≤ 10 times	—	80.51
% of states updated 0 times	—	3.18

Both methods produced policies averaging between 14 and 15 steps to cross the finish line, but RTDP required only roughly half of the updates that DP did. This is the result of RTDP’s on-policy trajectory sampling. Whereas the value of every state was updated

in each sweep of DP, RTDP focused updates on fewer states. In an average run, RTDP updated the values of 98.45% of the states no more than 100 times and 80.51% of the states no more than 10 times; the values of about 290 states were not updated at all in an average run. ■

Another advantage of RTDP is that as the value function approaches the optimal value function v_* , the policy used by the agent to generate trajectories approaches an optimal policy because it is always greedy with respect to the current value function. This is in contrast to the situation in conventional value iteration. In practice, value iteration terminates when the value function changes by only a small amount in a sweep, which is how we terminated it to obtain the results in the table above. At this point, the value function closely approximates v_* , and a greedy policy is close to an optimal policy. However, it is possible that policies that are greedy with respect to the latest value function were optimal, or nearly so, long before value iteration terminates. (Recall from Chapter 4 that optimal policies can be greedy with respect to many different value functions, not just v_* .) Checking for the emergence of an optimal policy before value iteration converges is not a part of the conventional DP algorithm and requires considerable additional computation.

In the racetrack example, by running many test episodes after each DP sweep, with actions selected greedily according to the result of that sweep, it was possible to estimate the earliest point in the DP computation at which the approximated optimal evaluation function was good enough so that the corresponding greedy policy was nearly optimal. For this racetrack, a close-to-optimal policy emerged after 15 sweeps of value iteration, or after 136,725 value-iteration updates. This is considerably less than the 252,784 updates DP needed to converge to v_* , but still more than the 127,600 updates RTDP required.

Although these simulations are certainly not definitive comparisons of the RTDP with conventional sweep-based value iteration, they illustrate some of advantages of on-policy trajectory sampling. Whereas conventional value iteration continued to update the value of all the states, RTDP strongly focused on subsets of the states that were relevant to the problem's objective. This focus became increasingly narrow as learning continued. Because the convergence theorem for RTDP applies to the simulations, we know that RTDP eventually would have focused only on relevant states, i.e., on states making up optimal paths. RTDP achieved nearly optimal control with about 50% of the computation required by sweep-based value iteration.

8.8 Planning at Decision Time

Planning can be used in at least two ways. The one we have considered so far in this chapter, typified by dynamic programming and Dyna, is to use planning to gradually improve a policy or value function on the basis of simulated experience obtained from a model (either a sample or a distribution model). Selecting actions is then a matter of comparing the current state's action values obtained from a table in the tabular case we have thus far considered, or by evaluating a mathematical expression in the approximate methods we consider in Part II below. Well before an action is selected for any current state S_t , planning has played a part in improving the table entries, or the function

approximation parameters, needed to select actions for many states, including S_t . Used this way, planning is not focused on the current state. We call planning used in this way *background planning*.

The other way to use planning is to begin and complete it *after* encountering each new state S_t , as a computation whose output is the selection of a single action A_t ; on the next step planning begins anew with S_{t+1} to produce A_{t+1} , and so on. The simplest, and almost degenerate, example of this use of planning is when only state values are available, and an action is selected by comparing the values of model-predicted next states for each action (or by comparing the values of afterstates as in the tic-tac-toe example in Chapter 1). More generally, planning used in this way can look much deeper than one-step-ahead and evaluate action choices leading to many different predicted state and reward trajectories. Unlike the first use of planning, here planning focuses on a particular state. We call this *decision-time planning*.

These two ways of thinking about planning—using simulated experience to gradually improve a policy or value function, or using simulated experience to select an action for the current state—can blend together in natural and interesting ways, but they have tended to be studied separately, and that is a good way to first understand them. Let us now take a closer look at decision-time planning.

Even when planning is only done at decision time, we can still view it, as we did in Section 8.1, as proceeding from simulated experience to updates and values, and ultimately to a policy. It is just that now the values and policy are specific to the current state and the action choices available there, so much so that the values and policy created by the planning process are typically discarded after being used to select the current action. In many applications this is not a great loss because there are very many states and we are unlikely to return to the same state for a long time. In general, one may want to do a mix of both: focus planning on the current state *and* store the results of planning so as to be that much farther along should one return to the same state later. Decision-time planning is most useful in applications in which fast responses are not required. In chess playing programs, for example, one may be permitted seconds or minutes of computation for each move, and strong programs may plan dozens of moves ahead within this time. On the other hand, if low latency action selection is the priority, then one is generally better off doing planning in the background to compute a policy that can then be rapidly applied to each newly encountered state.

8.9 Heuristic Search

The classical state-space planning methods in artificial intelligence are decision-time planning methods collectively known as *heuristic search*. In heuristic search, for each state encountered, a large tree of possible continuations is considered. The approximate value function is applied to the leaf nodes and then backed up toward the current state at the root. The backing up within the search tree is just the same as in the expected updates with maxes (those for v_* and q_*) discussed throughout this book. The backing up stops at the state-action nodes for the current state. Once the backed-up values of these nodes are computed, the best of them is chosen as the current action, and then all backed-up values are discarded.

In conventional heuristic search no effort is made to save the backed-up values by changing the approximate value function. In fact, the value function is generally designed by people and never changed as a result of search. However, it is natural to consider allowing the value function to be improved over time, using either the backed-up values computed during heuristic search or any of the other methods presented throughout this book. In a sense we have taken this approach all along. Our greedy, ϵ -greedy, and UCB (Section 2.7) action-selection methods are not unlike heuristic search, albeit on a smaller scale. For example, to compute the greedy action given a model and a state-value function, we must look ahead from each possible action to each possible next state, take into account the rewards and estimated values, and then pick the best action. Just as in conventional heuristic search, this process computes backed-up values of the possible actions, but does not attempt to save them. Thus, heuristic search can be viewed as an extension of the idea of a greedy policy beyond a single step.

The point of searching deeper than one step is to obtain better action selections. If one has a perfect model and an imperfect action-value function, then in fact deeper search will usually yield better policies.² Certainly, if the search is all the way to the end of the episode, then the effect of the imperfect value function is eliminated, and the action determined in this way must be optimal. If the search is of sufficient depth k such that γ^k is very small, then the actions will be correspondingly near optimal. On the other hand, the deeper the search, the more computation is required, usually resulting in a slower response time. A good example is provided by Tesauro's grandmaster-level backgammon player, TD-Gammon (Section 16.1). This system used TD learning to learn an afterstate value function through many games of self-play, using a form of heuristic search to make its moves. As a model, TD-Gammon used a priori knowledge of the probabilities of dice rolls and the assumption that the opponent always selected the actions that TD-Gammon rated as best for it. Tesauro found that the deeper the heuristic search, the better the moves made by TD-Gammon, but the longer it took to make each move. Backgammon has a large branching factor, yet moves must be made within a few seconds. It was only feasible to search ahead selectively a few steps, but even so the search resulted in significantly better action selections.

We should not overlook the most obvious way in which heuristic search focuses updates: on the current state. Much of the effectiveness of heuristic search is due to its search tree being tightly focused on the states and actions that might immediately follow the current state. You may spend more of your life playing chess than checkers, but when you play checkers, it pays to think about checkers and about your particular checkers position, your likely next moves, and successor positions. No matter how you select actions, it is these states and actions that are of highest priority for updates and where you most urgently want your approximate value function to be accurate. Not only should your computation be preferentially devoted to imminent events, but so should your limited memory resources. In chess, for example, there are far too many possible positions to store distinct value estimates for each of them, but chess programs based on heuristic search can easily store distinct estimates for the millions of positions they encounter

²There are interesting exceptions to this (see Pearl, 1984).

looking ahead from a single position. This great focusing of memory and computational resources on the current decision is presumably the reason why heuristic search can be so effective.

The distribution of updates can be altered in similar ways to focus on the current state and its likely successors. As a limiting case we might use exactly the methods of heuristic search to construct a search tree, and then perform the individual, one-step updates from bottom up, as suggested by Figure 8.9. If the updates are ordered in this way and a tabular representation is used, then exactly the same overall update would be achieved as in depth-first heuristic search. Any state-space search can be viewed in this way as the piecing together of a large number of individual one-step updates. Thus, the performance improvement observed with deeper searches is not due to the use of multistep updates as such. Instead, it is due to the focus and concentration of updates on states and actions immediately downstream from the current state. By devoting a large amount of computation specifically relevant to the candidate actions, decision-time planning can produce better decisions than can be produced by relying on unfocused updates.

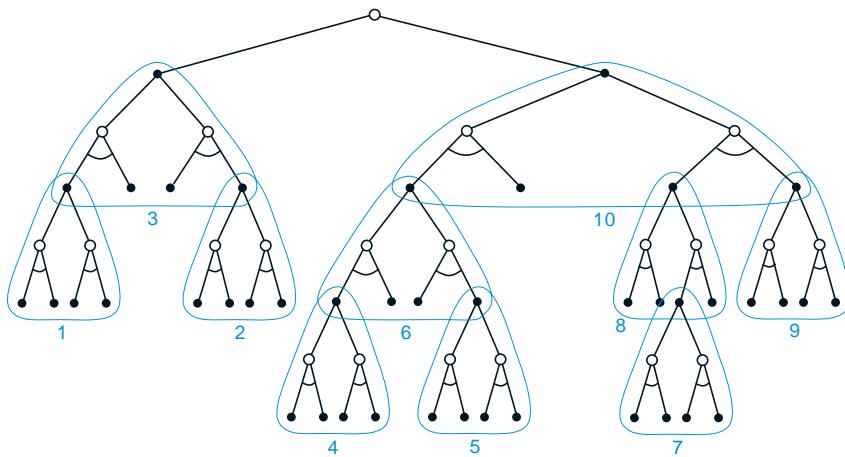


Figure 8.9: Heuristic search can be implemented as a sequence of one-step updates (shown here outlined in blue) backing up values from the leaf nodes toward the root. The ordering shown here is for a selective depth-first search.

8.10 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy. When the action-value estimates are considered to be accurate enough, the action (or one of the

actions) having the highest estimated value is executed, after which the process is carried out anew from the resulting next state. As explained by Tesauro and Galperin (1997), who experimented with rollout algorithms for playing backgammon, the term “rollout” comes from estimating the value of a backgammon position by playing out, i.e., “rolling out,” the position many times to the game’s end with randomly generated sequences of dice rolls, where the moves of both players are made by some fixed policy.

Unlike the Monte Carlo control algorithms described in Chapter 5, the goal of a rollout algorithm is not to estimate a complete optimal action-value function, q_* , or a complete action-value function, q_π , for a given policy π . Instead, they produce Monte Carlo estimates of action values only for each current state and for a given policy usually called the *rollout policy*. As decision-time planning algorithms, rollout algorithms make immediate use of these action-value estimates, then discard them. This makes rollout algorithms relatively simple to implement because there is no need to sample outcomes for every state-action pair, and there is no need to approximate a function over either the state space or the state-action space.

What then do rollout algorithms accomplish? The policy improvement theorem described in Section 4.2 tells us that given any two policies π and π' that are identical except that $\pi'(s) = a \neq \pi(s)$ for some state s , if $q_\pi(s, a) \geq v_\pi(s)$, then policy π' is as good as, or better, than π . Moreover, if the inequality is strict, then π' is in fact better than π . This applies to rollout algorithms where s is the current state and π is the rollout policy. Averaging the returns of the simulated trajectories produces estimates of $q_\pi(s, a')$ for each action $a' \in \mathcal{A}(s)$. Then the policy that selects an action in s that maximizes these estimates and thereafter follows π is a good candidate for a policy that improves over π . The result is like one step of the policy-iteration algorithm of dynamic programming discussed in Section 4.3 (though it is more like one step of *asynchronous* value iteration described in Section 4.5 because it changes the action for just the current state).

In other words, the aim of a rollout algorithm is to improve upon the rollout policy; not to find an optimal policy. Experience has shown that rollout algorithms can be surprisingly effective. For example, Tesauro and Galperin (1997) were surprised by the dramatic improvements in backgammon playing ability produced by the rollout method. In some applications, a rollout algorithm can produce good performance even if the rollout policy is completely random. But the performance of the improved policy depends on properties of the rollout policy and the ranking of actions produced by the Monte Carlo value estimates. Intuition suggests that the better the rollout policy and the more accurate the value estimates, the better the policy produced by a rollout algorithm is likely be (but see Gelly and Silver, 2007).

This involves important tradeoffs because better rollout policies typically mean that more time is needed to simulate enough trajectories to obtain good value estimates. As decision-time planning methods, rollout algorithms usually have to meet strict time constraints. The computation time needed by a rollout algorithm depends on the number of actions that have to be evaluated for each decision, the number of time steps in the simulated trajectories needed to obtain useful sample returns, the time it takes the rollout policy to make decisions, and the number of simulated trajectories needed to obtain good Monte Carlo action-value estimates.

Balancing these factors is important in any application of rollout methods, though there are several ways to ease the challenge. Because the Monte Carlo trials are independent of one another, it is possible to run many trials in parallel on separate processors. Another approach is to truncate the simulated trajectories short of complete episodes, correcting the truncated returns by means of a stored evaluation function (which brings into play all that we have said about truncated returns and updates in the preceding chapters). It is also possible, as Tesauro and Galperin (1997) suggest, to monitor the Monte Carlo simulations and prune away candidate actions that are unlikely to turn out to be the best, or whose values are close enough to that of the current best that choosing them instead would make no real difference (though Tesauro and Galperin point out that this would complicate a parallel implementation).

We do not ordinarily think of rollout algorithms as *learning* algorithms because they do not maintain long-term memories of values or policies. However, these algorithms take advantage of some of the features of reinforcement learning that we have emphasized in this book. As instances of Monte Carlo control, they estimate action values by averaging the returns of a collection of sample trajectories, in this case trajectories of simulated interactions with a sample model of the environment. In this way they are like reinforcement learning algorithms in avoiding the exhaustive sweeps of dynamic programming by trajectory sampling, and in avoiding the need for distribution models by relying on sample, instead of expected, updates. Finally, rollout algorithms take advantage of the policy improvement property by acting greedily with respect to the estimated action values.

8.11 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning. At its base, MCTS is a rollout algorithm as described above, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo simulations in order to successively direct simulations toward more highly-rewarding trajectories. MCTS is largely responsible for the improvement in computer Go from a weak amateur level in 2005 to a grandmaster level (6 dan or more) in 2015. Many variations of the basic algorithm have been developed, including a variant that we discuss in Section 16.6 that was critical for the stunning 2016 victories of the program AlphaGo over an 18-time world champion Go player. MCTS has proved to be effective in a wide variety of competitive settings, including general game playing (e.g., see Finnsson and Björnsson, 2008; Genesereth and Thielscher, 2014), but it is not limited to games; it can be effective for single-agent sequential decision problems if there is an environment model simple enough for fast multistep simulation.

MCTS is executed after encountering each new state to select the agent’s action for that state; it is executed again to select the action for the next state, and so on. As in a rollout algorithm, each execution is an iterative process that simulates many trajectories starting from the current state and running to a terminal state (or until discounting makes any further reward negligible as a contribution to the return). The core idea of MCTS is to successively focus multiple simulations starting at the current state by

extending the initial portions of trajectories that have received high evaluations from earlier simulations. MCTS does not have to retain approximate value functions or policies from one action selection to the next, though in many implementations it retains selected action values likely to be useful for its next execution.

For the most part, the actions in the simulated trajectories are generated using a simple policy, usually called a rollout policy as it is for simpler rollout algorithms. When both the rollout policy and the model do not require a lot of computation, many simulated trajectories can be generated in a short period of time. As in any tabular Monte Carlo method, the value of a state-action pair is estimated as the average of the (simulated) returns from that pair. Monte Carlo value estimates are maintained only for the subset of state-action pairs that are most likely to be reached in a few steps, which form a tree rooted at the current state, as illustrated in Figure 8.10. MCTS incrementally extends the tree by adding nodes representing states that look promising based on the results of the simulated trajectories. Any simulated trajectory will pass through the tree and then exit it at some leaf node. Outside the tree and at the leaf nodes the rollout policy is used for action selections, but at the states inside the tree something better is possible. For these states we have value estimates for at least some of the actions, so we can pick among them using an informed policy, called the *tree policy*, that balances exploration

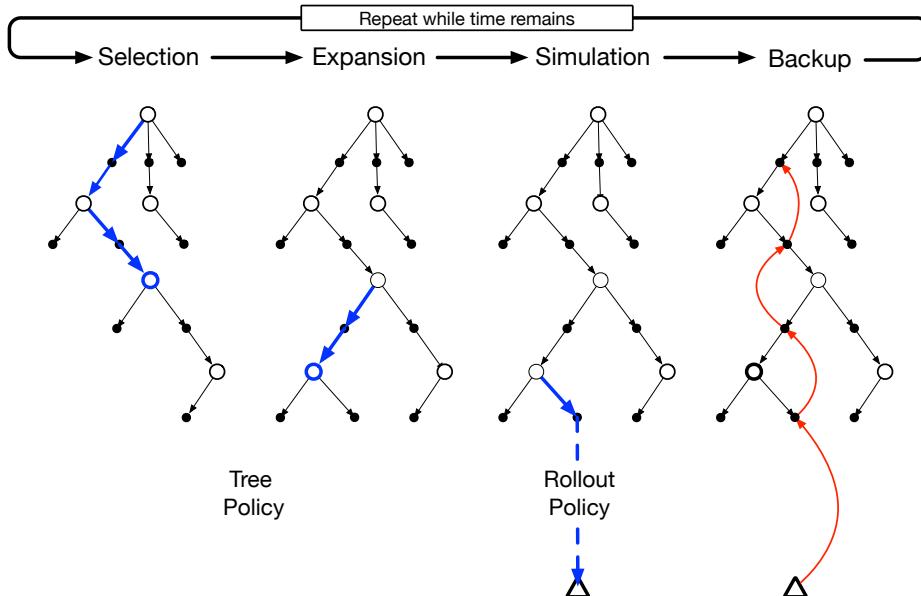


Figure 8.10: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations **Selection**, **Expansion** (though possibly skipped on some iterations), **Simulation**, and **Backup**, as explained in the text and illustrated by the bold arrows in the trees. Adapted from Chaslot, Bakkes, Szita, and Spronck (2008).

and exploitation. For example, the tree policy could select actions using an ε -greedy or UCB selection rule (Chapter 2).

In more detail, each iteration of a basic version of MCTS consists of the following four steps as illustrated in Figure 8.10:

1. **Selection.** Starting at the root node, a *tree policy* based on the action values attached to the edges of the tree traverses the tree to select a leaf node.
2. **Expansion.** On some iterations (depending on details of the application), the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.
3. **Simulation.** From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.
4. **Backup.** The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree. Figure 8.10 illustrates this by showing a backup from the terminal state of the simulated trajectory directly to the state-action node in the tree where the rollout policy began (though in general, the entire return over the simulated trajectory is backed up to this state-action node).

MCTS continues executing these four steps, starting each time at the tree’s root node, until no more time is left, or some other computational resource is exhausted. Then, finally, an action from the root node (which still represents the current state of the environment) is selected according to some mechanism that depends on the accumulated statistics in the tree; for example, it may be an action having the largest action value of all the actions available from the root state, or perhaps the action with the largest visit count to avoid selecting outliers. This is the action MCTS actually selects. After the environment transitions to a new state, MCTS is run again, sometimes starting with a tree of a single root node representing the new state, but often starting with a tree containing any descendants of this node left over from the tree constructed by the previous execution of MCTS; all the remaining nodes are discarded, along with the action values associated with them.

MCTS was first proposed to select moves in programs playing two-person competitive games, such as Go. For game playing, each simulated episode is one complete play of the game in which both players select actions by the tree and rollout policies. Section 16.6 describes an extension of MCTS used in the AlphaGo program that combines the Monte Carlo evaluations of MCTS with action values learned by a deep artificial neural network via self-play reinforcement learning.

Relating MCTS to the reinforcement learning principles we describe in this book provides some insight into how it achieves such impressive results. At its base, MCTS is a decision-time planning algorithm based on Monte Carlo control applied to simulations

that start from the root state; that is, it is a kind of rollout algorithm as described in the previous section. It therefore benefits from online, incremental, sample-based value estimation and policy improvement. Beyond this, it saves action-value estimates attached to the tree edges and updates them using reinforcement learning’s sample updates. This has the effect of focusing the Monte Carlo trials on trajectories whose initial segments are common to high-return trajectories previously simulated. Further, by incrementally expanding the tree, MCTS effectively grows a lookup table to store a partial action-value function, with memory allocated to the estimated values of state-action pairs visited in the initial segments of high-yielding sample trajectories. MCTS thus avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration.

The striking success of decision-time planning by MCTS has deeply influenced artificial intelligence, and many researchers are studying modifications and extensions of the basic procedure for use in both games and single-agent applications.

8.12 Summary of the Chapter

Planning requires a model of the environment. A *distribution model* consists of the probabilities of next states and rewards for possible actions; a sample model produces single transitions and rewards generated according to these probabilities. Dynamic programming requires a distribution model because it uses *expected updates*, which involve computing expectations over all the possible next states and rewards. A *sample model*, on the other hand, is what is needed to simulate interacting with the environment during which *sample updates*, like those used by many reinforcement learning algorithms, can be used. Sample models are generally much easier to obtain than distribution models.

We have presented a perspective emphasizing the surprisingly close relationships between planning optimal behavior and learning optimal behavior. Both involve estimating the same value functions, and in both cases it is natural to update the estimates incrementally, in a long series of small backing-up operations. This makes it straightforward to integrate learning and planning processes simply by allowing both to update the same estimated value function. In addition, any of the learning methods can be converted into planning methods simply by applying them to simulated (model-generated) experience rather than to real experience. In this case learning and planning become even more similar; they are possibly identical algorithms operating on two different sources of experience.

It is straightforward to integrate incremental planning methods with acting and model-learning. Planning, acting, and model-learning interact in a circular fashion (as in the diagram on page 162), each producing what the other needs to improve; no other interaction among them is either required or prohibited. The most natural approach is for all processes to proceed asynchronously and in parallel. If the processes must share computational resources, then the division can be handled almost arbitrarily—by whatever organization is most convenient and efficient for the task at hand.

In this chapter we have touched upon a number of dimensions of variation among state-space planning methods. One dimension is the variation in the size of updates. The

smaller the updates, the more incremental the planning methods can be. Among the smallest updates are one-step sample updates, as in Dyna. Another important dimension is the distribution of updates, that is, of the focus of search. Prioritized sweeping focuses backward on the predecessors of states whose values have recently changed. On-policy trajectory sampling focuses on states or state-action pairs that the agent is likely to encounter when controlling its environment. This can allow computation to skip over parts of the state space that are irrelevant to the prediction or control problem. Real-time dynamic programming, an on-policy trajectory sampling version of value iteration, illustrates some of the advantages this strategy has over conventional sweep-based policy iteration.

Planning can also focus forward from pertinent states, such as states actually encountered during an agent-environment interaction. The most important form of this is when planning is done at decision time, that is, as part of the action-selection process. Classical heuristic search as studied in artificial intelligence is an example of this. Other examples are rollout algorithms and Monte Carlo Tree Search that benefit from online, incremental, sample-based value estimation and policy improvement.

8.13 Summary of Part I: Dimensions

This chapter concludes Part I of this book. In it we have tried to present reinforcement learning not as a collection of individual methods, but as a coherent set of ideas cutting across methods. Each idea can be viewed as a dimension along which methods vary. The set of such dimensions spans a large space of possible methods. By exploring this space at the level of dimensions we hope to obtain the broadest and most lasting understanding. In this section we use the concept of dimensions in method space to recapitulate the view of reinforcement learning developed so far in this book.

All of the methods we have explored so far in this book have three key ideas in common: first, they all seek to estimate value functions; second, they all operate by backing up values along actual or possible state trajectories; and third, they all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other. These three ideas are central to the subjects covered in this book. We suggest that value functions, backing up value updates, and GPI are powerful organizing principles potentially relevant to any model of intelligence, whether artificial or natural.

Two of the most important dimensions along which the methods vary are shown in Figure 8.11. These dimensions have to do with the kind of update used to improve the value function. The horizontal dimension is whether they are sample updates (based on a sample trajectory) or expected updates (based on a distribution of possible trajectories). Expected updates require a distribution model, whereas sample updates need only a sample model, or can be done from actual experience with no model at all (another dimension of variation). The vertical dimension of Figure 8.11 corresponds to the depth of updates, that is, to the degree of bootstrapping. At three of the four corners of the space are the three primary methods for estimating values: dynamic programming, TD, and Monte Carlo. Along the left edge of the space are the sample-update methods,

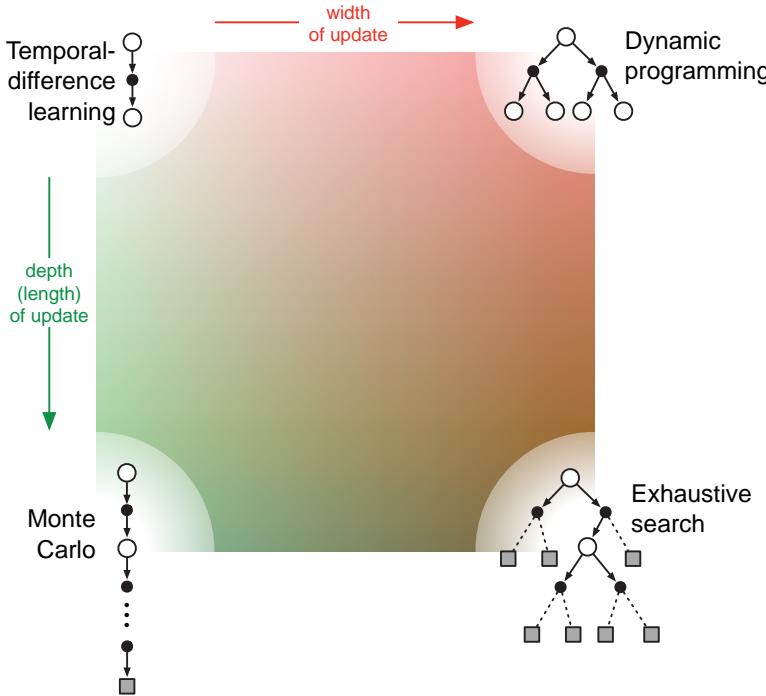


Figure 8.11: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

ranging from one-step TD updates to full-return Monte Carlo updates. Between these is a spectrum including methods based on n -step updates (and in Chapter 12 we will extend this to mixtures of n -step updates such as the λ -updates implemented by eligibility traces).

Dynamic programming methods are shown in the extreme upper-right corner of the space because they involve one-step expected updates. The lower-right corner is the extreme case of expected updates so deep that they run all the way to terminal states (or, in a continuing task, until discounting has reduced the contribution of any further rewards to a negligible level). This is the case of exhaustive search. Intermediate methods along this dimension include heuristic search and related methods that search and update up to a limited depth, perhaps selectively. There are also methods that are intermediate along the horizontal dimension. These include methods that mix expected and sample updates, as well as the possibility of methods that mix samples and distributions within a single update. The interior of the square is filled in to represent the space of all such intermediate methods.

A third dimension that we have emphasized in this book is the binary distinction between on-policy and off-policy methods. In the former case, the agent learns the value function for the policy it is currently following, whereas in the latter case it learns the

value function for the policy for a different policy, often the one that the agent currently thinks is best. The policy generating behavior is typically different from what is currently thought best because of the need to explore. This third dimension might be visualized as perpendicular to the plane of the page in Figure 8.11.

In addition to the three dimensions just discussed, we have identified a number of others throughout the book:

Definition of return Is the task episodic or continuing, discounted or undiscounted?

Action values vs. state values vs. afterstate values What kind of values should be estimated? If only state values are estimated, then either a model or a separate policy (as in actor-critic methods) is required for action selection.

Action selection/exploration How are actions selected to ensure a suitable trade-off between exploration and exploitation? We have considered only the simplest ways to do this: ϵ -greedy, optimistic initialization of values, soft-max, and upper confidence bound.

Synchronous vs. asynchronous Are the updates for all states performed simultaneously or one by one in some order?

Real vs. simulated Should one update based on real experience or simulated experience? If both, how much of each?

Location of updates What states or state-action pairs should be updated? Model-free methods can choose only among the states and state-action pairs actually encountered, but model-based methods can choose arbitrarily. There are many possibilities here.

Timing of updates Should updates be done as part of selecting actions, or only afterward?

Memory for updates How long should updated values be retained? Should they be retained permanently, or only while computing an action selection, as in heuristic search?

Of course, these dimensions are neither exhaustive nor mutually exclusive. Individual algorithms differ in many other ways as well, and many algorithms lie in several places along several dimensions. For example, Dyna methods use both real and simulated experience to affect the same value function. It is also perfectly sensible to maintain multiple value functions computed in different ways or over different state and action representations. These dimensions do, however, constitute a coherent set of ideas for describing and exploring a wide space of possible methods.

The most important dimension not mentioned here, and not covered in Part I of this book, is that of function approximation. Function approximation can be viewed as an orthogonal spectrum of possibilities ranging from tabular methods at one extreme through state aggregation, a variety of linear methods, and then a diverse set of nonlinear methods. This dimension is explored in Part II.

Bibliographical and Historical Remarks

- 8.1** The overall view of planning and learning presented here has developed gradually over a number of years, in part by the authors (Sutton, 1990, 1991a, 1991b; Barto, Bradtke, and Singh, 1991, 1995; Sutton and Pinette, 1985; Sutton and Barto, 1981b); it has been strongly influenced by Agre and Chapman (1990; Agre 1988), Bertsekas and Tsitsiklis (1989), Singh (1993), and others. The authors were also strongly influenced by psychological studies of latent learning (Tolman, 1932) and by psychological views of the nature of thought (e.g., Galanter and Gerstenhaber, 1956; Craik, 1943; Campbell, 1960; Dennett, 1978). In Part III of the book, Section 14.6 relates model-based and model-free methods to psychological theories of learning and behavior, and Section 15.11 discusses ideas about how the brain might implement these types of methods.
- 8.2** The terms *direct* and *indirect*, which we use to describe different kinds of reinforcement learning, are from the adaptive control literature (e.g., Goodwin and Sin, 1984), where they are used to make the same kind of distinction. The term *system identification* is used in adaptive control for what we call *model-learning* (e.g., Goodwin and Sin, 1984; Ljung and Söderstrom, 1983; Young, 1984). The Dyna architecture is due to Sutton (1990), and the results in this and the next section are based on results reported there. Barto and Singh (1990) consider some of the issues in comparing direct and indirect reinforcement learning methods. Early work extending Dyna to linear function approximation was done by Sutton, Szepesvári, Geramifard, and Bowling (2008) and by Parr, Li, Taylor, Painter-Wakefield, and Littman (2008).
- 8.3** There have been several works with model-based reinforcement learning that take the idea of exploration bonuses and optimistic initialization to its logical extreme, in which all incompletely explored choices are assumed maximally rewarding and optimal paths are computed to test them. The E³ algorithm of Kearns and Singh (2002) and the R-max algorithm of Brafman and Tennenholtz (2003) are guaranteed to find a near-optimal solution in time polynomial in the number of states and actions. This is usually too slow for practical algorithms but is probably the best that can be done in the worst case.
- 8.4** Prioritized sweeping was developed simultaneously and independently by Moore and Atkeson (1993) and Peng and Williams (1993). The results in the box on page 170 are due to Peng and Williams (1993). The results in the box on page 171 are due to Moore and Atkeson. Key subsequent work in this area includes that by McMahan and Gordon (2005) and by van Seijen and Sutton (2013).
- 8.5** This section was strongly influenced by the experiments of Singh (1993).
- 8.6–7** Trajectory sampling has implicitly been a part of reinforcement learning from the outset, but it was most explicitly emphasized by Barto, Bradtke, and Singh (1995) in their introduction of RTDP. They recognized that Korf's (1990) *learning*

*real-time A** (LRTA*) algorithm is an asynchronous DP algorithm that applies to stochastic problems as well as the deterministic problems on which Korf focused. Beyond LRTA*, RTDP includes the option of updating the values of many states in the time intervals between the execution of actions. Barto et al. (1995) proved the convergence result described here by combining Korf’s (1990) convergence proof for LRTA* with the result of Bertsekas (1982) (also Bertsekas and Tsitsiklis, 1989) ensuring convergence of asynchronous DP for stochastic shortest path problems in the undiscounted case. Combining model-learning with RTDP is called *Adaptive* RTDP, also presented by Barto et al. (1995) and discussed by Barto (2011).

- 8.9** For further reading on heuristic search, the reader is encouraged to consult texts and surveys such as those by Russell and Norvig (2009) and Korf (1988). Peng and Williams (1993) explored a forward focusing of updates much as is suggested in this section.
- 8.10** Abramson’s (1990) expected-outcome model is a rollout algorithm applied to two-person games in which the play of both simulated players is random. He argued that even with random play, it is a “powerful heuristic” that is “precise, accurate, easily estimable, efficiently calculable, and domain-independent.” Tesauro and Galperin (1997) demonstrated the effectiveness of rollout algorithms for improving the play of backgammon programs, adopting the term “rollout” from its use in evaluating backgammon positions by playing out positions with different randomly generating sequences of dice rolls. Bertsekas, Tsitsiklis, and Wu (1997) examine rollout algorithms applied to combinatorial optimization problems, and Bertsekas (2013) surveys their use in discrete deterministic optimization problems, remarking that they are “often surprisingly effective.”
- 8.11** The central ideas of MCTS were introduced by Coulom (2006) and by Kocsis and Szepesvári (2006). They built upon previous research with Monte Carlo planning algorithms as reviewed by these authors. Browne, Powley, Whitehouse, Lucas, Cowling, Rohlfsagen, Tavener, Perez, Samothrakis, and Colton (2012) is an excellent survey of MCTS methods and their applications. David Silver contributed to the ideas and presentation in this section.

Part II: Approximate Solution Methods

In the second part of the book we extend the tabular methods presented in the first part to apply to problems with arbitrarily large state spaces. In many of the tasks to which we would like to apply reinforcement learning the state space is combinatorial and enormous; the number of possible camera images, for example, is much larger than the number of atoms in the universe. In such cases we cannot expect to find an optimal policy or the optimal value function even in the limit of infinite time and data; our goal instead is to find a good approximate solution using limited computational resources. In this part of the book we explore such approximate solution methods.

The problem with large state spaces is not just the memory needed for large tables, but the time and data needed to fill them accurately. In many of our target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one. In other words, the key issue is that of *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

Fortunately, generalization from examples has already been extensively studied, and we do not need to invent totally new methods for use in reinforcement learning. To some extent we need only combine reinforcement learning methods with existing generalization methods. The kind of generalization we require is often called *function approximation* because it takes examples from a desired function (e.g., a value function) and attempts to generalize from them to construct an approximation of the entire function. Function approximation is an instance of *supervised learning*, the primary topic studied in machine learning, artificial neural networks, pattern recognition, and statistical curve fitting. In theory, any of the methods studied in these fields can be used in the role of function approximator within reinforcement learning algorithms, although in practice some fit more easily into this role than others.

Reinforcement learning with function approximation involves a number of new issues that do not normally arise in conventional supervised learning, such as nonstationarity, bootstrapping, and delayed targets. We introduce these and other issues successively over the five chapters of this part. Initially we restrict attention to on-policy training, treating in Chapter 9 the prediction case, in which the policy is given and only its value function is approximated, and then in Chapter 10 the control case, in which an approximation to the optimal policy is found. The challenging problem of off-policy learning with function approximation is treated in Chapter 11. In each of these three chapters we will have

to return to first principles and re-examine the objectives of the learning to take into account function approximation. Chapter 12 introduces and analyzes the algorithmic mechanism of *eligibility traces*, which dramatically improves the computational properties of multi-step reinforcement learning methods in many cases. The final chapter of this part explores a different approach to control, *policy-gradient methods*, which approximate the optimal policy directly and need never form an approximate value function (although they may be much more efficient if they do approximate a value function as well the policy).

Chapter 9

On-policy Prediction with Approximation

In this chapter, we begin our study of function approximation in reinforcement learning by considering its use in estimating the state-value function from on-policy data, that is, in approximating v_π from experience generated using a known policy π . The novelty in this chapter is that the approximate value function is represented not as a table but as a parameterized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$. We will write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector \mathbf{w} . For example, \hat{v} might be a linear function in features of the state, with \mathbf{w} the vector of feature weights. More generally, \hat{v} might be the function computed by a multi-layer artificial neural network, with \mathbf{w} the vector of connection weights in all the layers. By adjusting the weights, any of a wide range of different functions can be implemented by the network. Or \hat{v} might be the function computed by a decision tree, where \mathbf{w} is all the numbers defining the split points and leaf values of the tree. Typically, the number of weights (the dimensionality of \mathbf{w}) is much less than the number of states ($d \ll |\mathcal{S}|$), and changing one weight changes the estimated value of many states. Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states. Such *generalization* makes the learning potentially more powerful but also potentially more difficult to manage and understand.

Perhaps surprisingly, extending reinforcement learning to function approximation also makes it applicable to partially observable problems, in which the full state is not available to the agent. If the parameterized function form for \hat{v} does not allow the estimated value to depend on certain aspects of the state, then it is just as if those aspects are unobservable. In fact, all the theoretical results for methods using function approximation presented in this part of the book apply equally well to cases of partial observability. What function approximation can't do, however, is augment the state representation with memories of past observations. Some such possible further extensions are discussed briefly in Section 17.3.

9.1 Value-function Approximation

All of the prediction methods covered in this book have been described as updates to an estimated value function that shift its value at particular states toward a “backed-up value,” or *update target*, for that state. Let us refer to an individual update by the notation $s \mapsto u$, where s is the state updated and u is the update target that s ’s estimated value is shifted toward. For example, the Monte Carlo update for value prediction is $S_t \mapsto G_t$, the TD(0) update is $S_t \mapsto R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$, and the n -step TD update is $S_t \mapsto G_{t:t+n}$. In the DP (dynamic programming) policy-evaluation update, $s \mapsto \mathbb{E}_{\pi}[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) \mid S_t = s]$, an arbitrary state s is updated, whereas in the other cases the state encountered in actual experience, S_t , is updated.

It is natural to interpret each update as specifying an example of the desired input–output behavior of the value function. In a sense, the update $s \mapsto u$ means that the estimated value for state s should be more like the update target u . Up to now, the actual update has been trivial: the table entry for s ’s estimated value has simply been shifted a fraction of the way toward u , and the estimated values of all other states were left unchanged. Now we permit arbitrarily complex and sophisticated methods to implement the update, and updating at s generalizes so that the estimated values of many other states are changed as well. Machine learning methods that learn to mimic input–output examples in this way are called *supervised learning* methods, and when the outputs are numbers, like u , the process is often called *function approximation*. Function approximation methods expect to receive examples of the desired input–output behavior of the function they are trying to approximate. We use these methods for value prediction simply by passing to them the $s \mapsto u$ of each update as a training example. We then interpret the approximate function they produce as an estimated value function.

Viewing each update as a conventional training example in this way enables us to use any of a wide range of existing function approximation methods for value prediction. In principle, we can use any method for supervised learning from examples, including artificial neural networks, decision trees, and various kinds of multivariate regression. However, not all function approximation methods are equally well suited for use in reinforcement learning. The most sophisticated artificial neural network and statistical methods all assume a static training set over which multiple passes are made. In reinforcement learning, however, it is important that learning be able to occur online, while the agent interacts with its environment or with a model of its environment. To do this requires methods that are able to learn efficiently from incrementally acquired data. In addition, reinforcement learning generally requires function approximation methods able to handle nonstationary target functions (target functions that change over time). For example, in control methods based on GPI (generalized policy iteration) we often seek to learn q_{π} while π changes. Even if the policy remains the same, the target values of training examples are nonstationary if they are generated by bootstrapping methods (DP and TD learning). Methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning.

9.2 The Prediction Objective (\overline{VE})

Up to now we have not specified an explicit objective for prediction. In the tabular case a continuous measure of prediction quality was not necessary because the learned value function could come to equal the true value function exactly. Moreover, the learned values at each state were decoupled—an update at one state affected no other. But with genuine approximation, an update at one state affects many others, and it is not possible to get the values of all states exactly correct. By assumption we have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate. We are obligated then to say which states we care most about. We must specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s . By the error in a state s we mean the square of the difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$. Weighting this over the state space by μ , we obtain a natural objective function, the *mean square value error*, denoted \overline{VE} :

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2. \quad (9.1)$$

The square root of this measure, the root \overline{VE} , gives a rough measure of how much the approximate values differ from the true values and is often used in plots. Often $\mu(s)$ is chosen to be the fraction of time spent in s . Under on-policy training this is called the *on-policy distribution*; we focus entirely on this case in this chapter. In continuing tasks, the on-policy distribution is the stationary distribution under π .

The on-policy distribution in episodic tasks

In an episodic task, the on-policy distribution is a little different in that it depends on how the initial states of episodes are chosen. Let $h(s)$ denote the probability that an episode begins in each state s , and let $\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode. Time is spent in a state s if episodes start in s , or if transitions are made into s from a preceding state \bar{s} in which time is spent:

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s}, a), \quad \text{for all } s \in \mathcal{S}. \quad (9.2)$$

This system of equations can be solved for the expected number of visits $\eta(s)$. The on-policy distribution is then the fraction of time spent in each state normalized to sum to one:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \text{for all } s \in \mathcal{S}. \quad (9.3)$$

This is the natural choice without discounting. If there is discounting ($\gamma < 1$) it should be treated as a form of termination, which can be done simply by including a factor of γ in the second term of (9.2).

The two cases, continuing and episodic, behave similarly, but with approximation they must be treated separately in formal analyses, as we will see repeatedly in this part of the book. This completes the specification of the learning objective.

It is not completely clear that the \overline{VE} is the right performance objective for reinforcement learning. Remember that our ultimate purpose—the reason we are learning a value function—is to find a better policy. The best value function for this purpose is not necessarily the best for minimizing \overline{VE} . Nevertheless, it is not yet clear what a more useful alternative goal for value prediction might be. For now, we will focus on \overline{VE} .

An ideal goal in terms of \overline{VE} would be to find a *global optimum*, a weight vector \mathbf{w}^* for which $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for all possible \mathbf{w} . Reaching this goal is sometimes possible for simple function approximators such as linear ones, but is rarely possible for complex function approximators such as artificial neural networks and decision trees. Short of this, complex function approximators may seek to converge instead to a *local optimum*, a weight vector \mathbf{w}^* for which $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w})$ for all \mathbf{w} in some neighborhood of \mathbf{w}^* . Although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear function approximators, and often it is enough. Still, for many cases of interest in reinforcement learning there is no guarantee of convergence to an optimum, or even to within a bounded distance of an optimum. Some methods may in fact diverge, with their \overline{VE} approaching infinity in the limit.

In the last two sections we outlined a framework for combining a wide range of reinforcement learning methods for value prediction with a wide range of function approximation methods, using the updates of the former to generate training examples for the latter. We also described a \overline{VE} performance measure which these methods may aspire to minimize. The range of possible function approximation methods is far too large to cover all, and anyway too little is known about most of them to make a reliable evaluation or recommendation. Of necessity, we consider only a few possibilities. In the rest of this chapter we focus on function approximation methods based on gradient principles, and on linear gradient-descent methods in particular. We focus on these methods in part because we consider them to be particularly promising and because they reveal key theoretical issues, but also because they are simple and our space is limited.

9.3 Stochastic-gradient and Semi-gradient Methods

We now develop in detail one class of learning methods for function approximation in value prediction, those based on stochastic gradient descent (SGD). SGD methods are among the most widely used of all function approximation methods and are particularly well suited to online reinforcement learning.

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w} \doteq (w_1, w_2, \dots, w_d)^\top$,¹ and the approximate value function $\hat{v}(s, \mathbf{w})$ is a differentiable function of \mathbf{w} for all $s \in \mathcal{S}$. We will be updating \mathbf{w} at each of a series of discrete time steps, $t = 0, 1, 2, 3, \dots$, so we will need a notation \mathbf{w}_t for the

¹The $^\top$ denotes transpose, needed here to turn the horizontal row vector in the text into a vertical column vector; in this book vectors are generally taken to be column vectors unless explicitly written out horizontally or transposed.

weight vector at each step. For now, let us assume that, on each step, we observe a new example $S_t \mapsto v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy. These states might be successive states from an interaction with the environment, but for now we do not assume so. Even though we are given the exact, correct values, $v_\pi(S_t)$ for each S_t , there is still a difficult problem because our function approximator has limited resources and thus limited resolution. In particular, there is generally no \mathbf{w} that gets all the states, or even all the examples, exactly correct. In addition, we must generalize to all the other states that have not appeared in examples.

We assume that states appear in examples with the same distribution, μ , over which we are trying to minimize the $\overline{\text{VE}}$ as given by (9.1). A good strategy in this case is to try to minimize error on the observed examples. *Stochastic gradient-descent* (SGD) methods do this by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \frac{1}{2}\alpha \nabla \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 \quad (9.4)$$

$$= \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t), \quad (9.5)$$

where α is a positive step-size parameter, and $\nabla f(\mathbf{w})$, for any scalar expression $f(\mathbf{w})$ that is a function of a vector (here \mathbf{w}), denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \dots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^\top. \quad (9.6)$$

This derivative vector is the *gradient* of f with respect to \mathbf{w} . SGD methods are “gradient descent” methods because the overall step in \mathbf{w}_t is proportional to the negative gradient of the example’s squared error (9.4). This is the direction in which the error falls most rapidly. Gradient descent methods are called “stochastic” when the update is done, as here, on only a single example, which might have been selected stochastically. Over many examples, making small steps, the overall effect is to minimize an average performance measure such as the $\overline{\text{VE}}$.

It may not be immediately apparent why SGD takes only a small step in the direction of the gradient. Could we not move all the way in this direction and completely eliminate the error on the example? In many cases this could be done, but usually it is not desirable. Remember that we do not seek or expect to find a value function that has zero error for all states, but only an approximation that balances the errors in different states. If we completely corrected each example in one step, then we would not find such a balance. In fact, the convergence results for SGD methods assume that α decreases over time. If it decreases in such a way as to satisfy the standard stochastic approximation conditions (2.7), then the SGD method (9.5) is guaranteed to converge to a local optimum.

We turn now to the case in which the target output, here denoted $U_t \in \mathbb{R}$, of the t th training example, $S_t \mapsto U_t$, is not the true value, $v_\pi(S_t)$, but some, possibly random, approximation to it. For example, U_t might be a noise-corrupted version of $v_\pi(S_t)$, or it might be one of the bootstrapping targets using \hat{v} mentioned in the previous section. In

in these cases we cannot perform the exact update (9.5) because $v_\pi(S_t)$ is unknown, but we can approximate it by substituting U_t in place of $v_\pi(S_t)$. This yields the following general SGD method for state-value prediction:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [U_t - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t). \quad (9.7)$$

If U_t is an *unbiased* estimate, that is, if $\mathbb{E}[U_t | S_t = s] = v_\pi(s)$, for each t , then \mathbf{w}_t is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (2.7) for decreasing α .

For example, suppose the states in the examples are the states generated by interaction (or simulated interaction) with the environment using policy π . Because the true value of a state is the expected value of the return following it, the Monte Carlo target $U_t \doteq G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$. With this choice, the general SGD method (9.7) converges to a locally optimal approximation to $v_\pi(S_t)$. Thus, the gradient-descent version of Monte Carlo state-value prediction is guaranteed to find a locally optimal solution. Pseudocode for a complete algorithm is shown in the box below.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

```

Input: the policy  $\pi$  to be evaluated
Input: a differentiable function  $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$ 
Algorithm parameter: step size  $\alpha > 0$ 
Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )
Loop forever (for each episode):
  Generate an episode  $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$  using  $\pi$ 
  Loop for each step of episode,  $t = 0, 1, \dots, T - 1$ :
     $\mathbf{w} \leftarrow \mathbf{w} + \alpha [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$ 

```

One does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t in (9.7). Bootstrapping targets such as n -step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r} \pi(a|S_t)p(s',r|S_t,a)[r + \gamma \hat{v}(s',\mathbf{w}_t)]$ all depend on the current value of the weight vector \mathbf{w}_t , which implies that they will be biased and that they will not produce a true gradient-descent method. One way to look at this is that the key step from (9.4) to (9.5) relies on the target being independent of \mathbf{w}_t . This step would not be valid if a bootstrapping estimate were used in place of $v_\pi(S_t)$. Bootstrapping methods are not in fact instances of true gradient descent (Barnard, 1993). They take into account the effect of changing the weight vector \mathbf{w}_t on the estimate, but ignore its effect on the target. They include only a part of the gradient and, accordingly, we call them *semi-gradient methods*.

Although semi-gradient (bootstrapping) methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the linear case discussed in the next section. Moreover, they offer important advantages that make them often clearly preferred. One reason for this is that they typically enable significantly faster learning, as we have seen in Chapters 6 and 7. Another is that they enable learning to

be continual and online, without waiting for the end of an episode. This enables them to be used on continuing problems and provides computational advantages. A prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ as its target. Complete pseudocode for this method is given in the box below.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

```

Input: the policy  $\pi$  to be evaluated
Input: a differentiable function  $\hat{v} : S^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  such that  $\hat{v}(\text{terminal}, \cdot) = 0$ 
Algorithm parameter: step size  $\alpha > 0$ 
Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = \mathbf{0}$ )
Loop for each episode:
    Initialize  $S$ 
    Loop for each step of episode:
        Choose  $A \sim \pi(\cdot | S)$ 
        Take action  $A$ , observe  $R, S'$ 
         $\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$ 
         $S \leftarrow S'$ 
    until  $S$  is terminal

```

State aggregation is a simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector \mathbf{w}) for each group. The value of a state is estimated as its group's component, and when the state is updated, that component alone is updated. State aggregation is a special case of SGD (9.7) in which the gradient, $\nabla \hat{v}(S_t, \mathbf{w}_t)$, is 1 for S_t 's group's component and 0 for the other components.

Example 9.1: State Aggregation on the 1000-state Random Walk Consider a 1000-state version of the random walk task (Examples 6.2 and 7.1 on pages 125 and 144). The states are numbered from 1 to 1000, left to right, and all episodes begin near the center, in state 500. State transitions are from the current state to one of the 100 neighboring states to its left, or to one of the 100 neighboring states to its right, all with equal probability. Of course, if the current state is near an edge, then there may be fewer than 100 neighbors on that side of it. In this case, all the probability that would have gone into those missing neighbors goes into the probability of terminating on that side (thus, state 1 has a 0.5 chance of terminating on the left, and state 950 has a 0.25 chance of terminating on the right). As usual, termination on the left produces a reward of -1 , and termination on the right produces a reward of $+1$. All other transitions have a reward of zero. We use this task as a running example throughout this section.

Figure 9.1 shows the true value function v_π for this task. It is nearly a straight line, curving very slightly toward the horizontal for the last 100 states at each end. Also shown is the final approximate value function learned by the gradient Monte-Carlo algorithm with state aggregation after 100,000 episodes with a step size of $\alpha = 2 \times 10^{-5}$. For the state aggregation, the 1000 states were partitioned into 10 groups of 100 states each (i.e., states 1–100 were one group, states 101–200 were another, and so on). The staircase effect

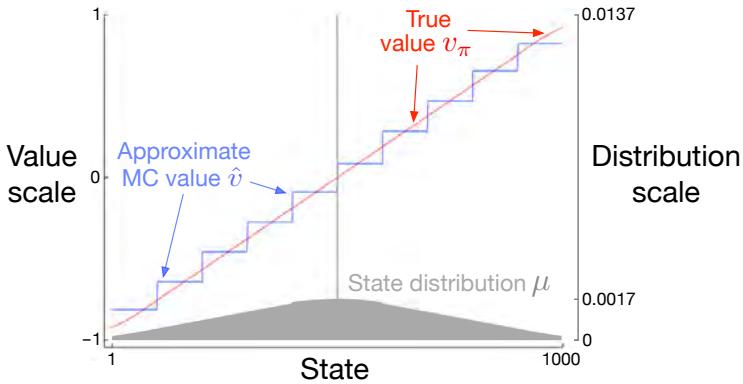


Figure 9.1: Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm (page 202).

shown in the figure is typical of state aggregation; within each group, the approximate value is constant, and it changes abruptly from one group to the next. These approximate values are close to the global minimum of the \overline{VE} (9.1).

Some of the details of the approximate values are best appreciated by reference to the state distribution μ for this task, shown in the lower portion of the figure with a right-side scale. State 500, in the center, is the first state of every episode, but is rarely visited again. On average, about 1.37% of the time steps are spent in the start state. The states reachable in one step from the start state are the second most visited, with about 0.17% of the time steps being spent in each of them. From there μ falls off almost linearly, reaching about 0.0147% at the extreme states 1 and 1000. The most visible effect of the distribution is on the leftmost groups, whose values are clearly shifted higher than the unweighted average of the true values of states within the group, and on the rightmost groups, whose values are clearly shifted lower. This is due to the states in these areas having the greatest asymmetry in their weightings by μ . For example, in the leftmost group, state 100 is weighted more than 3 times more strongly than state 1. Thus the estimate for the group is biased toward the true value of state 100, which is higher than the true value of state 1. ■

9.4 Linear Methods

One of the most important special cases of function approximation is that in which the approximate function, $\hat{v}(\cdot, \mathbf{w})$, is a linear function of the weight vector, \mathbf{w} . Corresponding to every state s , there is a real-valued vector $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \dots, x_d(s))^\top$, with the same number of components as \mathbf{w} . Linear methods approximate the state-value function

by the inner product between \mathbf{w} and $\mathbf{x}(s)$:

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^d w_i x_i(s). \quad (9.8)$$

In this case the approximate value function is said to be *linear in the weights*, or simply *linear*.

The vector $\mathbf{x}(s)$ is called a *feature vector* representing state s . Each component $x_i(s)$ of $\mathbf{x}(s)$ is the value of a function $x_i : \mathcal{S} \rightarrow \mathbb{R}$. We think of a *feature* as the entirety of one of these functions, and we call its value for a state s a *feature of s* . For linear methods, features are *basis functions* because they form a linear basis for the set of approximate functions. Constructing d -dimensional feature vectors to represent states is the same as selecting a set of d basis functions. Features may be defined in many different ways; we cover a few possibilities in the next sections.

It is natural to use SGD updates with linear function approximation. The gradient of the approximate value function with respect to \mathbf{w} in this case is

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s).$$

Thus, in the linear case the general SGD update (9.7) reduces to a particularly simple form:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [U_t - \hat{v}(S_t, \mathbf{w}_t)] \mathbf{x}(S_t).$$

Because it is so simple, the linear SGD case is one of the most favorable for mathematical analysis. Almost all useful convergence results for learning systems of all kinds are for linear (or simpler) function approximation methods.

In particular, in the linear case there is only one optimum (or, in degenerate cases, one set of equally good optima), and thus any method that is guaranteed to converge to or near a local optimum is automatically guaranteed to converge to or near the global optimum. For example, the gradient Monte Carlo algorithm presented in the previous section converges to the global optimum of the $\overline{\text{VE}}$ under linear function approximation if α is reduced over time according to the usual conditions.

The semi-gradient TD(0) algorithm presented in the previous section also converges under linear function approximation, but this does not follow from general results on SGD; a separate theorem is necessary. The weight vector converged to is also not the global optimum, but rather a point near the local optimum. It is useful to consider this important case in more detail, specifically for the continuing case. The update at each time t is

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha \left(R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t \right) \mathbf{x}_t \\ &= \mathbf{w}_t + \alpha \left(R_{t+1} \mathbf{x}_t - \mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top \mathbf{w}_t \right), \end{aligned} \quad (9.9)$$

where here we have used the notational shorthand $\mathbf{x}_t = \mathbf{x}(S_t)$. Once the system has reached steady state, for any given \mathbf{w}_t , the expected next weight vector can be written

$$\mathbb{E}[\mathbf{w}_{t+1} | \mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t), \quad (9.10)$$

where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \quad \text{and} \quad \mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top] \in \mathbb{R}^{d \times d} \quad (9.11)$$

From (9.10) it is clear that, if the system converges, it must converge to the weight vector \mathbf{w}_{TD} at which

$$\begin{aligned} \mathbf{b} - \mathbf{A}\mathbf{w}_{\text{TD}} &= \mathbf{0} \\ \Rightarrow \mathbf{b} &= \mathbf{A}\mathbf{w}_{\text{TD}} \\ \Rightarrow \mathbf{w}_{\text{TD}} &\doteq \mathbf{A}^{-1}\mathbf{b}. \end{aligned} \quad (9.12)$$

This quantity is called the *TD fixed point*. In fact linear semi-gradient TD(0) converges to this point. Some of the theory proving its convergence, and the existence of the inverse above, is given in the box.

Proof of Convergence of Linear TD(0)

What properties assure convergence of the linear TD(0) algorithm (9.9)? Some insight can be gained by rewriting (9.10) as

$$\mathbb{E}[\mathbf{w}_{t+1} | \mathbf{w}_t] = (\mathbf{I} - \alpha\mathbf{A})\mathbf{w}_t + \alpha\mathbf{b}. \quad (9.13)$$

Note that the matrix \mathbf{A} multiplies the weight vector \mathbf{w}_t and not \mathbf{b} ; only \mathbf{A} is important to convergence. To develop intuition, consider the special case in which \mathbf{A} is a diagonal matrix. If any of the diagonal elements are negative, then the corresponding diagonal element of $\mathbf{I} - \alpha\mathbf{A}$ will be greater than one, and the corresponding component of \mathbf{w}_t will be amplified, which will lead to divergence if continued. On the other hand, if the diagonal elements of \mathbf{A} are all positive, then α can be chosen smaller than one over the largest of them, such that $\mathbf{I} - \alpha\mathbf{A}$ is diagonal with all diagonal elements between 0 and 1. In this case the first term of the update tends to shrink \mathbf{w}_t , and stability is assured. In general, \mathbf{w}_t will be reduced toward zero whenever \mathbf{A} is *positive definite*, meaning $y^\top \mathbf{A} y > 0$ for any real vector $y \neq 0$. Positive definiteness also ensures that the inverse \mathbf{A}^{-1} exists.

For linear TD(0), in the continuing case with $\gamma < 1$, the \mathbf{A} matrix (9.11) can be written

$$\begin{aligned} \mathbf{A} &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{r,s'} p(r,s'|s,a) \mathbf{x}(s) (\mathbf{x}(s) - \gamma \mathbf{x}(s'))^\top \\ &= \sum_s \mu(s) \sum_{s'} p(s'|s) \mathbf{x}(s) (\mathbf{x}(s) - \gamma \mathbf{x}(s'))^\top \\ &= \sum_s \mu(s) \mathbf{x}(s) \left(\mathbf{x}(s) - \gamma \sum_{s'} p(s'|s) \mathbf{x}(s') \right)^\top \\ &= \mathbf{X}^\top \mathbf{D} (\mathbf{I} - \gamma \mathbf{P}) \mathbf{X}, \end{aligned}$$

where $\mu(s)$ is the stationary distribution under π , $p(s'|s)$ is the probability of transition from s to s' under policy π , \mathbf{P} is the $|\mathcal{S}| \times |\mathcal{S}|$ matrix of these probabilities,

\mathbf{D} is the $|S| \times |S|$ diagonal matrix with the $\mu(s)$ on its diagonal, and \mathbf{X} is the $|S| \times d$ matrix with $\mathbf{x}(s)$ as its rows. From here it is clear that the inner matrix $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$ is key to determining the positive definiteness of \mathbf{A} .

For a key matrix of this form, positive definiteness is assured if all of its columns sum to a nonnegative number. This was shown by Sutton (1988, p. 27) based on two previously established theorems. One theorem says that any matrix \mathbf{M} is positive definite if and only if the symmetric matrix $\mathbf{S} = \mathbf{M} + \mathbf{M}^\top$ is positive definite (Sutton 1988, appendix). The second theorem says that any symmetric real matrix \mathbf{S} is positive definite if all of its diagonal entries are positive and greater than the sum of the absolute values of the corresponding off-diagonal entries (Varga 1962, p. 23). For our key matrix, $\mathbf{D}(\mathbf{I} - \gamma\mathbf{P})$, the diagonal entries are positive and the off-diagonal entries are negative, so all we have to show is that each row sum plus the corresponding column sum is positive. The row sums are all positive because \mathbf{P} is a stochastic matrix and $\gamma < 1$. Thus it only remains to show that the column sums are nonnegative. Note that the row vector of the column sums of any matrix \mathbf{M} can be written as $\mathbf{1}^\top \mathbf{M}$, where $\mathbf{1}$ is the column vector with all components equal to 1. Let $\boldsymbol{\mu}$ denote the $|S|$ -vector of the $\mu(s)$, where $\boldsymbol{\mu} = \mathbf{P}^\top \boldsymbol{\mu}$ by virtue of $\boldsymbol{\mu}$ being the stationary distribution. The column sums of our key matrix, then, are:

$$\begin{aligned}\mathbf{1}^\top \mathbf{D}(\mathbf{I} - \gamma\mathbf{P}) &= \boldsymbol{\mu}^\top (\mathbf{I} - \gamma\mathbf{P}) \\ &= \boldsymbol{\mu}^\top - \gamma\boldsymbol{\mu}^\top \mathbf{P} \\ &= \boldsymbol{\mu}^\top - \gamma\boldsymbol{\mu}^\top \quad (\text{because } \boldsymbol{\mu} \text{ is the stationary distribution}) \\ &= (1 - \gamma)\boldsymbol{\mu}^\top,\end{aligned}$$

all components of which are positive. Thus, the key matrix and its \mathbf{A} matrix are positive definite, and on-policy TD(0) is stable. (Additional conditions and a schedule for reducing α over time are needed to prove convergence with probability one.)

At the TD fixed point, it has also been proven (in the continuing case) that the $\overline{\text{VE}}$ is within a bounded expansion of the lowest possible error:

$$\overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \leq \frac{1}{1 - \gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \quad (9.14)$$

That is, the asymptotic error of the TD method is no more than $\frac{1}{1-\gamma}$ times the smallest possible error, that attained in the limit by the Monte Carlo method. Because γ is often near one, this expansion factor can be quite large, so there is substantial potential loss in asymptotic performance with the TD method. On the other hand, recall that the TD methods are often of vastly reduced variance compared to Monte Carlo methods, and thus faster, as we saw in Chapters 6 and 7. Which method will be best depends on the nature of the approximation and problem, and on how long learning continues.

A bound analogous to (9.14) applies to other on-policy bootstrapping methods as well. For example, linear semi-gradient DP (Eq. 9.7 with $U_t \doteq \sum_a \pi(a|S_t) \sum_{s',r} p(s',r|S_t,a)[r + \gamma \hat{v}(s',\mathbf{w}_t)]$) with updates according to the on-policy distribution will also converge to the TD fixed point. One-step semi-gradient *action-value* methods, such as semi-gradient Sarsa(0) covered in the next chapter converge to an analogous fixed point and an analogous bound. For episodic tasks, there is a slightly different but related bound (see Bertsekas and Tsitsiklis, 1996). There are also a few technical conditions on the rewards, features, and decrease in the step-size parameter, which we have omitted here. The full details can be found in the original paper (Tsitsiklis and Van Roy, 1997).

Critical to these convergence results is that states are updated according to the on-policy distribution. For other update distributions, bootstrapping methods using function approximation may actually diverge to infinity. Examples of this and a discussion of possible solution methods are given in Chapter 11.

Example 9.2: Bootstrapping on the 1000-state Random Walk State aggregation is a special case of linear function approximation, so let's return to the 1000-state random walk to illustrate some of the observations made in this chapter. The left panel of Figure 9.2 shows the final value function learned by the semi-gradient TD(0) algorithm (page 203) using the same state aggregation as in Example 9.1. We see that the near-asymptotic TD approximation is indeed farther from the true values than the Monte Carlo approximation shown in Figure 9.1.

Nevertheless, TD methods retain large potential advantages in learning rate, and generalize Monte Carlo methods, as we investigated fully with n -step TD methods in Chapter 7. The right panel of Figure 9.2 shows results with an n -step semi-gradient TD method using state aggregation on the 1000-state random walk that are strikingly similar to those we obtained earlier with tabular methods and the 19-state random walk (Figure 7.2). To obtain such quantitatively similar results we switched the state aggregation to 20 groups of 50 states each. The 20 groups were then quantitatively close

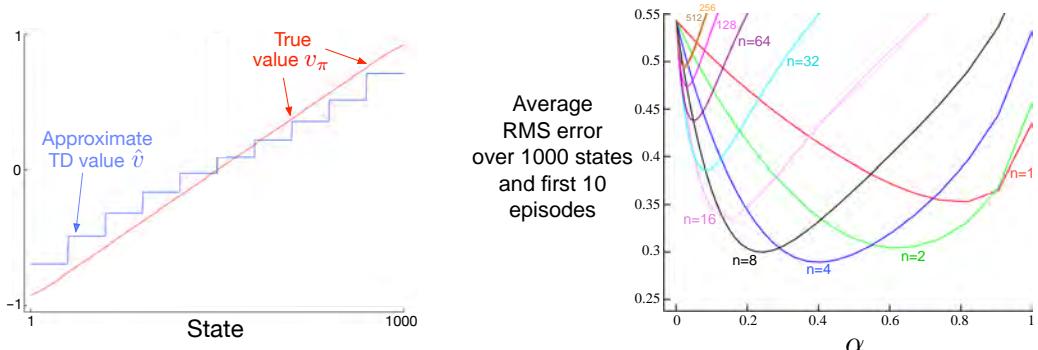


Figure 9.2: Bootstrapping with state aggregation on the 1000-state random walk task. *Left:* Asymptotic values of semi-gradient TD are worse than the asymptotic Monte Carlo values in Figure 9.1. *Right:* Performance of n -step methods with state-aggregation are strikingly similar to those with tabular representations (cf. Figure 7.2). These data are averages over 100 runs.

to the 19 states of the tabular problem. In particular, recall that state transitions were up to 100 states to the left or right. A typical transition would then be of 50 states to the right or left, which is quantitatively analogous to the single-state state transitions of the 19-state tabular system. To complete the match, we use here the same performance measure—an unweighted average of the RMS error over all states and over the first 10 episodes—rather than a \overline{VE} objective as is otherwise more appropriate when using function approximation. ■

The semi-gradient n -step TD algorithm used in the example above is the natural extension of the tabular n -step TD algorithm presented in Chapter 7 to semi-gradient function approximation. Pseudocode is given in the box below.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated

Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameters: step size $\alpha > 0$, a positive integer n

Initialize value-function weights w arbitrarily (e.g., $w = 0$)

All store and access operations (S_t and R_t) can take their index mod $n + 1$

Loop for each episode:

Initialize and store $S_0 \neq \text{terminal}$

$$T \leftarrow \infty$$

Loop for $t = 0, 1, 2, \dots$:

If $t < T$, then:

Take an action according to $\pi(\cdot|S_t)$

Observe and store the next reward as R_{t+1} and the next state as S_{t+1}

If S_{t+1} is terminal, then $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)

If $\tau \geq 0$:

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$$

If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$

$$(G_{\tau:\tau+n})$$

$\mathbf{w} \leftarrow \mathbf{w} +$

The key equation of this algorithm, analogous to (7.2), is

$$\mathbf{w}_{t+n}^* \doteq \mathbf{w}_{t+n-1} + \alpha [G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T; \quad (9.15)$$

where the η -step return is generalized from (7.1) to

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{W}_{t+n-1}), \quad 0 \leq t \leq T-n. \quad (9.16)$$

Exercise 9.1 Show that tabular methods such as presented in Part I of this book are a special case of linear function approximation. What would the feature vectors be? \square

9.5 Feature Construction for Linear Methods

Linear methods are interesting because of their convergence guarantees, but also because in practice they can be very efficient in terms of both data and computation. Whether or not this is so depends critically on how the states are represented in terms of features, which we investigate in this large section. Choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems. Intuitively, the features should correspond to the aspects of the state space along which generalization may be appropriate. If we are valuing geometric objects, for example, we might want to have features for each possible shape, color, size, or function. If we are valuing states of a mobile robot, then we might want to have features for locations, degrees of remaining battery power, recent sonar readings, and so on.

A limitation of the linear form is that it cannot take into account any interactions between features, such as the presence of feature i being good only in the absence of feature j . For example, in the pole-balancing task (Example 3.4), high angular velocity can be either good or bad depending on the angle. If the angle is high, then high angular velocity means an imminent danger of falling—a bad state—whereas if the angle is low, then high angular velocity means the pole is righting itself—a good state. A linear value function could not represent this if its features coded separately for the angle and the angular velocity. It needs instead, or in addition, features for combinations of these two underlying state dimensions. In the following subsections we consider a variety of general ways of doing this.

9.5.1 Polynomials

The states of many problems are initially expressed as numbers, such as positions and velocities in the pole-balancing task (Example 3.4), the number of cars in each lot in the Jack’s car rental problem (Example 4.2), or the gambler’s capital in the gambler problem (Example 4.3). In these types of problems, function approximation for reinforcement learning has much in common with the familiar tasks of interpolation and regression. Various families of features commonly used for interpolation and regression can also be used in reinforcement learning. Polynomials make up one of the simplest families of features used for interpolation and regression. While the basic polynomial features we discuss here do not work as well as other types of features in reinforcement learning, they serve as a good introduction because they are simple and familiar.

As an example, suppose a reinforcement learning problem has states with two numerical dimensions. For a single representative state s , let its two numbers be $s_1 \in \mathbb{R}$ and $s_2 \in \mathbb{R}$. You might choose to represent s simply by its two state dimensions, so that $\mathbf{x}(s) = (s_1, s_2)^\top$, but then you would not be able to take into account any interactions between these dimensions. In addition, if both s_1 and s_2 were zero, then the approximate value would have to also be zero. Both limitations can be overcome by instead representing s by the four-dimensional feature vector $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2)^\top$. The initial 1 feature allows the representation of affine functions in the original state numbers, and the final product feature, $s_1 s_2$, enables interactions to be taken into account. Or you might choose to use higher-dimensional feature vectors like $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$ to

take more complex interactions into account. Such feature vectors enable approximations as arbitrary quadratic functions of the state numbers—even though the approximation is still linear in the weights that have to be learned. Generalizing this example from two to k numbers, we can represent highly-complex interactions among a problem’s state dimensions:

Suppose each state s corresponds to k numbers, s_1, s_2, \dots, s_k , with each $s_i \in \mathbb{R}$. For this k -dimensional state space, each order- n polynomial-basis feature x_i can be written as

$$x_i(s) = \prod_{j=1}^k s_j^{c_{i,j}}, \quad (9.17)$$

where each $c_{i,j}$ is an integer in the set $\{0, 1, \dots, n\}$ for an integer $n \geq 0$. These features make up the order- n polynomial basis for dimension k , which contains $(n+1)^k$ different features.

Higher-order polynomial bases allow for more accurate approximations of more complicated functions. But because the number of features in an order- n polynomial basis grows exponentially with the dimension k of the natural state space (if $n > 0$), it is generally necessary to select a subset of them for function approximation. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods developed for polynomial regression can be adapted to deal with the incremental and nonstationary nature of reinforcement learning.

Exercise 9.2 Why does (9.17) define $(n+1)^k$ distinct features for dimension k ? □

Exercise 9.3 What n and $c_{i,j}$ produce the feature vectors $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1 s_2^2, s_1^2 s_2, s_1^2 s_2^2)^\top$? □

9.5.2 Fourier Basis

Another linear function approximation method is based on the time-honored Fourier series, which expresses periodic functions as weighted sums of sine and cosine basis functions (features) of different frequencies. (A function f is periodic if $f(x) = f(x + \tau)$ for all x and some period τ .) The Fourier series and the more general Fourier transform are widely used in applied sciences in part because if a function to be approximated is known, then the basis function weights are given by simple formulae and, further, with enough basis functions essentially any function can be approximated as accurately as desired. In reinforcement learning, where the functions to be approximated are unknown, Fourier basis functions are of interest because they are easy to use and can perform well in a range of reinforcement learning problems.

First consider the one-dimensional case. The usual Fourier series representation of a function of one dimension having period τ represents the function as a linear combination of sine and cosine functions that are each periodic with periods that evenly divide τ (in other words, whose frequencies are integer multiples of a fundamental frequency $1/\tau$). But if you are interested in approximating an aperiodic function defined over a bounded

interval, then you can use these Fourier basis features with τ set to the length of the interval. The function of interest is then just one period of the periodic linear combination of the sine and cosine features.

Furthermore, if you set τ to twice the length of the interval of interest and restrict attention to the approximation over the half interval $[0, \tau/2]$, then you can use just the cosine features. This is possible because you can represent any *even* function, that is, any function that is symmetric about the origin, with just the cosine basis. So any function over the half-period $[0, \tau/2]$ can be approximated as closely as desired with enough cosine features. (Saying “any function” is not exactly correct because the function has to be mathematically well-behaved, but we skip this technicality here.) Alternatively, it is possible to use just sine features, linear combinations of which are always *odd* functions, that is functions that are anti-symmetric about the origin. But it is generally better to keep just the cosine features because “half-even” functions tend to be easier to approximate than “half-odd” functions because the latter are often discontinuous at the origin. Of course, this does not rule out using both sine and cosine features to approximate over the interval $[0, \tau/2]$, which might be advantageous in some circumstances.

Following this logic and letting $\tau = 2$ so that the features are defined over the half- τ interval $[0, 1]$, the one-dimensional order- n Fourier cosine basis consists of the $n + 1$ features

$$x_i(s) = \cos(i\pi s), \quad s \in [0, 1],$$

for $i = 0, \dots, n$. Figure 9.3 shows one-dimensional Fourier cosine features x_i , for $i = 1, 2, 3, 4$; x_0 is a constant function.

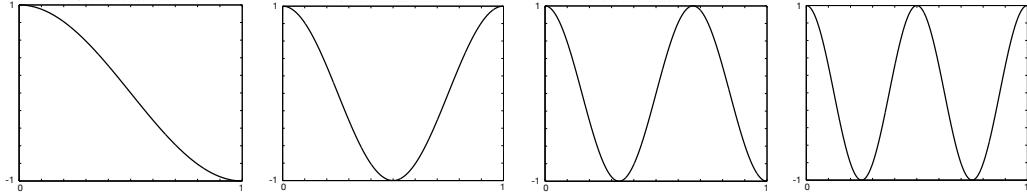


Figure 9.3: One-dimensional Fourier cosine-basis features x_i , $i = 1, 2, 3, 4$, for approximating functions over the interval $[0, 1]$. After Konidaris et al. (2011).

This same reasoning applies to the Fourier cosine series approximation in the multi-dimensional case as described in the box below.

Suppose each state s corresponds to a vector of k numbers, $\mathbf{s} = (s_1, s_2, \dots, s_k)^\top$, with each $s_i \in [0, 1]$. The i th feature in the order- n Fourier cosine basis can then be written

$$x_i(s) = \cos(\pi \mathbf{s}^\top \mathbf{c}^i), \tag{9.18}$$

where $\mathbf{c}^i = (c_1^i, \dots, c_k^i)^\top$, with $c_j^i \in \{0, \dots, n\}$ for $j = 1, \dots, k$ and $i = 1, \dots, (n+1)^k$. This defines a feature for each of the $(n+1)^k$ possible integer vectors \mathbf{c}^i . The inner

product $\mathbf{s}^\top \mathbf{c}^i$ has the effect of assigning an integer in $\{0, \dots, n\}$ to each dimension of \mathbf{s} . As in the one-dimensional case, this integer determines the feature's frequency along that dimension. The features can of course be shifted and scaled to suit the bounded state space of a particular application.

As an example, consider the $k = 2$ case in which $\mathbf{s} = (s_1, s_2)^\top$, where each $\mathbf{c}^i = (c_1^i, c_2^i)^\top$. Figure 9.4 shows a selection of six Fourier cosine features, each labeled by the vector \mathbf{c}^i that defines it (s_1 is the horizontal axis and \mathbf{c}^i is shown as a row vector with the index i omitted). Any zero in \mathbf{c} means the feature is constant along that state dimension. So if $\mathbf{c} = (0, 0)^\top$, the feature is constant over both dimensions; if $\mathbf{c} = (c_1, 0)^\top$ the feature is constant over the second dimension and varies over the first with frequency depending on c_1 ; and similarly, for $\mathbf{c} = (0, c_2)^\top$. When $\mathbf{c} = (c_1, c_2)^\top$ with neither $c_j = 0$, the feature varies along both dimensions and represents an interaction between the two state variables. The values of c_1 and c_2 determine the frequency along each dimension, and their ratio gives the direction of the interaction.

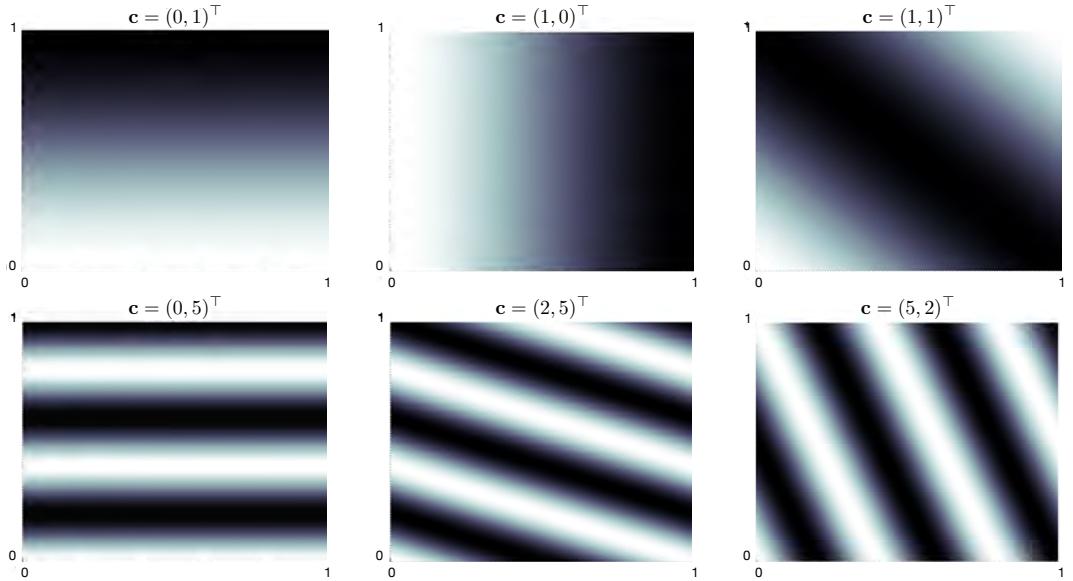


Figure 9.4: A selection of six two-dimensional Fourier cosine features, each labeled by the vector \mathbf{c}^i that defines it (s_1 is the horizontal axis, and \mathbf{c}^i is shown with the index i omitted). After Konidaris et al. (2011).

When using Fourier cosine features with a learning algorithm such as (9.7), semi-gradient TD(0), or semi-gradient Sarsa, it may be helpful to use a different step-size parameter for each feature. If α is the basic step-size parameter, then Konidaris, Osentoski, and Thomas (2011) suggest setting the step-size parameter for feature x_i to $\alpha_i = \alpha / \sqrt{(c_1^i)^2 + \dots + (c_k^i)^2}$ (except when each $c_j^i = 0$, in which case $\alpha_i = \alpha$).

Fourier cosine features with Sarsa can produce good performance compared to several other collections of basis functions, including polynomial and radial basis functions. Not surprisingly, however, Fourier features have trouble with discontinuities because it is difficult to avoid “ringing” around points of discontinuity unless very high frequency basis functions are included.

The number of features in the order- n Fourier basis grows exponentially with the dimension of the state space, but if that dimension is small enough (e.g., $k \leq 5$), then one can select n so that all of the order- n Fourier features can be used. This makes the selection of features more-or-less automatic. For high dimension state spaces, however, it is necessary to select a subset of these features. This can be done using prior beliefs about the nature of the function to be approximated, and some automated selection methods can be adapted to deal with the incremental and nonstationary nature of reinforcement learning. An advantage of Fourier basis features in this regard is that it is easy to select features by setting the \mathbf{c}^i vectors to account for suspected interactions among the state variables and by limiting the values in the \mathbf{c}^j vectors so that the approximation can filter out high frequency components considered to be noise. On the other hand, because Fourier features are non-zero over the entire state space (with the few zeros excepted), they represent global properties of states, which can make it difficult to find good ways to represent local properties.

Figure 9.5 shows learning curves comparing the Fourier and polynomial bases on the 1000-state random walk example. In general, we do not recommend using polynomials for online learning.²

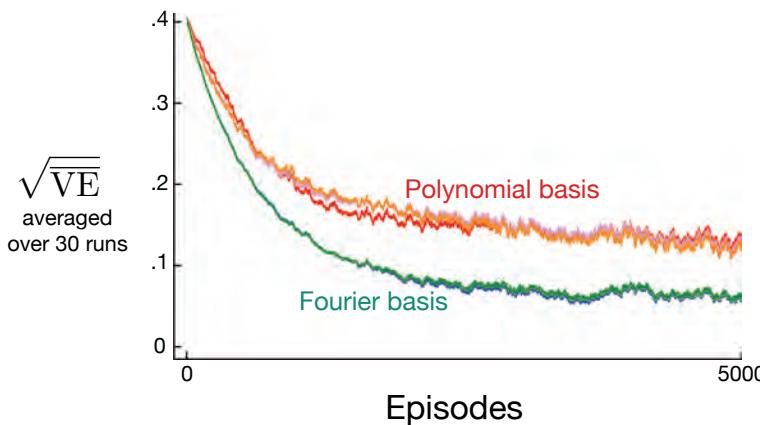


Figure 9.5: Fourier basis vs polynomials on the 1000-state random walk. Shown are learning curves for the gradient Monte Carlo method with Fourier and polynomial bases of order 5, 10, and 20. The step-size parameters were roughly optimized for each case: $\alpha = 0.0001$ for the polynomial basis and $\alpha = 0.00005$ for the Fourier basis. The performance measure (y-axis) is the root mean square value error (9.1).

²There are families of polynomials more complicated than those we have discussed, for example, different families of orthogonal polynomials, and these might work better, but at present there is little experience with them in reinforcement learning.

9.5.3 Coarse Coding

Consider a task in which the natural representation of the state set is a continuous two-dimensional space. One kind of representation for this case is made up of features corresponding to *circles* in state space, as shown to the right. If the state is inside a circle, then the corresponding feature has the value 1 and is said to be *present*; otherwise the feature is 0 and is said to be *absent*. This kind of 1–0-valued feature is called a *binary feature*. Given a state, which binary features are present indicate within which circles the state lies, and thus coarsely code for its location. Representing a state with features that overlap in this way (although they need not be circles or binary) is known as *coarse coding*.

Assuming linear gradient-descent function approximation, consider the effect of the size and density of the circles. Corresponding to each circle is a single weight (a component of \mathbf{w}) that is affected by learning. If we train at one state, a point in the space, then the weights of all circles intersecting that state will be affected. Thus, by (9.8), the approximate value function will be affected at all states within the union of the circles, with a greater effect the more circles a point has “in common” with the state, as shown in Figure 9.6. If the circles are small, then the generalization will be over a short distance, as in Figure 9.7 (left), whereas if they are large, it will be over a large distance, as in Figure 9.7 (middle). Moreover,

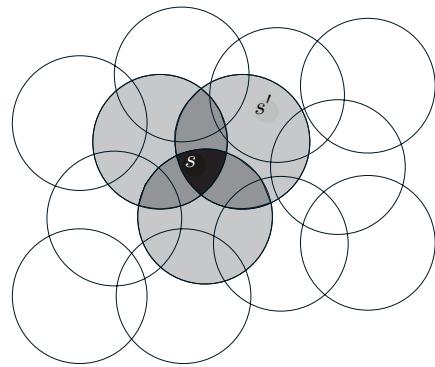


Figure 9.6: Coarse coding. Generalization from state s to state s' depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.

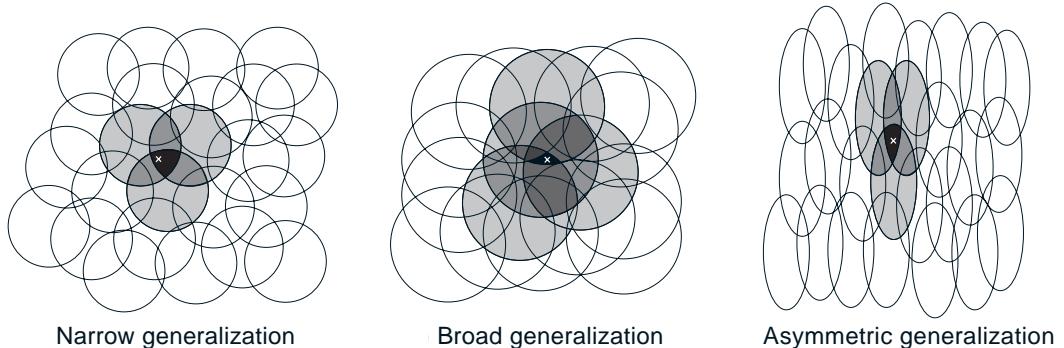


Figure 9.7: Generalization in linear function approximation methods is determined by the sizes and shapes of the features’ receptive fields. All three of these cases have roughly the same number and density of features.

the shape of the features will determine the nature of the generalization. For example, if they are not strictly circular, but are elongated in one direction, then generalization will be similarly affected, as in Figure 9.7 (right).

Features with large receptive fields give broad generalization, but might also seem to limit the learned function to a coarse approximation, unable to make discriminations much finer than the width of the receptive fields. Happily, this is not the case. Initial generalization from one point to another is indeed controlled by the size and shape of the receptive fields, but acuity, the finest discrimination ultimately possible, is controlled more by the total number of features.

Example 9.3: Coarseness of Coarse Coding This example illustrates the effect on learning of the size of the receptive fields in coarse coding. Linear function approximation based on coarse coding and (9.7) was used to learn a one-dimensional square-wave function (shown at the top of Figure 9.8). The values of this function were used as the targets, U_t . With just one dimension, the receptive fields were intervals rather than circles. Learning was repeated with three different sizes of the intervals: narrow, medium, and broad, as shown at the bottom of the figure. All three cases had the same density of features, about 50 over the extent of the function being learned. Training examples were generated uniformly at random over this extent. The step-size parameter was $\alpha = \frac{0.2}{n}$, where n is the number of features that were present at one time. Figure 9.8 shows the functions learned in all three cases over the course of learning. Note that the width of the features had a strong effect early in learning. With broad features, the generalization tended to be broad; with narrow features, only the close neighbors of each trained point were changed, causing the function learned to be more bumpy. However, the final function learned was affected only slightly by the width of the features. Receptive field shape tends to have a strong effect on generalization but little effect on asymptotic solution quality.

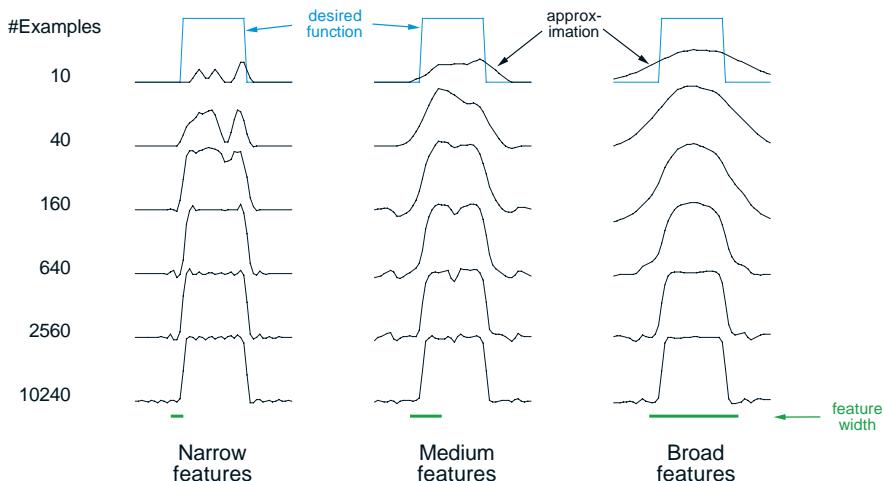


Figure 9.8: Example of feature width's strong effect on initial generalization (first row) and weak effect on asymptotic accuracy (last row). ■

9.5.4 Tile Coding

Tile coding is a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient. It may be the most practical feature representation for modern sequential digital computers.

In tile coding the receptive fields of the features are grouped into partitions of the state space. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. For example, the simplest tiling of a two-dimensional state space is a uniform grid such as that shown on the left side of Figure 9.9. The tiles or receptive field here are squares rather than the circles in Figure 9.6. If just this single tiling were used, then the state indicated by the white spot would be represented by the single feature whose tile it falls within; generalization would be complete to all states within the same tile and nonexistent to states outside it. With just one tiling, we would not have coarse coding but just a case of state aggregation.

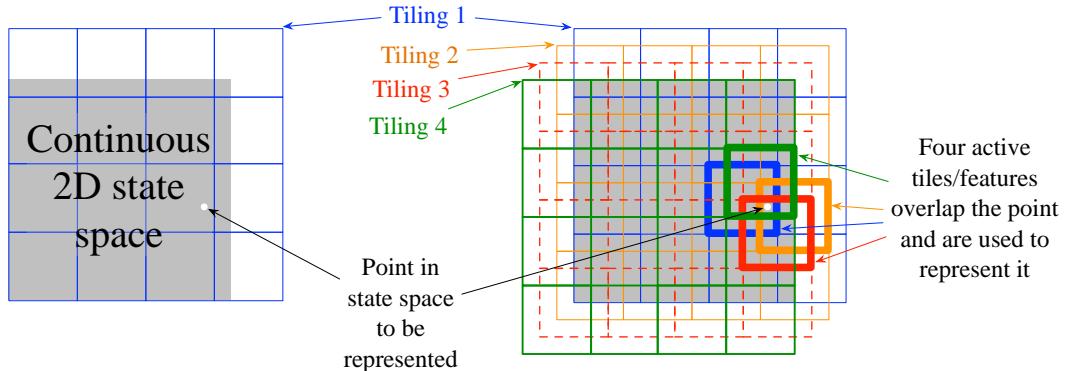


Figure 9.9: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

To get the strengths of coarse coding requires overlapping receptive fields, and by definition the tiles of a partition do not overlap. To get true coarse coding with tile coding, multiple tilings are used, each offset by a fraction of a tile width. A simple case with four tilings is shown on the right side of Figure 9.9. Every state, such as that indicated by the white spot, falls in exactly one tile in each of the four tilings. These four tiles correspond to four features that become active when the state occurs. Specifically, the feature vector $\mathbf{x}(s)$ has one component for each tile in each tiling. In this example there are $4 \times 4 \times 4 = 64$ components, all of which will be 0 except for the four corresponding to the tiles that s falls within. Figure 9.10 shows the advantage of multiple offset tilings (coarse coding) over a single tiling on the 1000-state random walk example.

An immediate practical advantage of tile coding is that, because it works with partitions, the overall number of features that are active at one time is the same for any state. Exactly one feature is present in each tiling, so the total number of features present is always the same as the number of tilings. This allows the step-size parameter, α , to

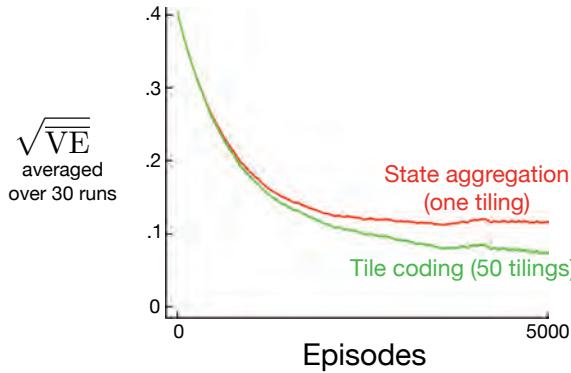


Figure 9.10: Why we use coarse coding. Shown are learning curves on the 1000-state random walk example for the gradient Monte Carlo algorithm with a single tiling and with multiple tilings. The space of 1000 states was treated as a single continuous dimension, covered with tiles each 200 states wide. The multiple tilings were offset from each other by 4 states. The step-size parameter was set so that the initial learning rate in the two cases was the same, $\alpha = 0.0001$ for the single tiling and $\alpha = 0.0001/50$ for the 50 tilings.

be set in an easy, intuitive way. For example, choosing $\alpha = \frac{1}{n}$, where n is the number of tilings, results in exact one-trial learning. If the example $s \mapsto v$ is trained on, then whatever the prior estimate, $\hat{v}(s, \mathbf{w}_t)$, the new estimate will be $\hat{v}(s, \mathbf{w}_{t+1}) = v$. Usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs. For example, one might choose $\alpha = \frac{1}{10n}$, in which case the estimate for the trained state would move one-tenth of the way to the target in one update, and neighboring states will be moved less, proportional to the number of tiles they have in common.

Tile coding also gains computational advantages from its use of binary feature vectors. Because each component is either 0 or 1, the weighted sum making up the approximate value function (9.8) is almost trivial to compute. Rather than performing d multiplications and additions, one simply computes the indices of the $n \ll d$ active features and then adds up the n corresponding components of the weight vector.

Generalization occurs to states other than the one trained if those states fall within any of the same tiles, proportional to the number of tiles in common. Even the choice of how to offset the tilings from each other affects generalization. If they are offset uniformly in each dimension, as they were in Figure 9.9, then different states can generalize in qualitatively different ways, as shown in the upper half of Figure 9.11. Each of the eight subfigures show the pattern of generalization from a trained state to nearby points. In this example there are eight tilings, thus 64 subregions within a tile that generalize distinctly, but all according to one of these eight patterns. Note how uniform offsets result in a strong effect along the diagonal in many patterns. These artifacts can be avoided if the tilings are offset asymmetrically, as shown in the lower half of the figure. These lower generalization patterns are better because they are all well centered on the trained state with no obvious asymmetries.

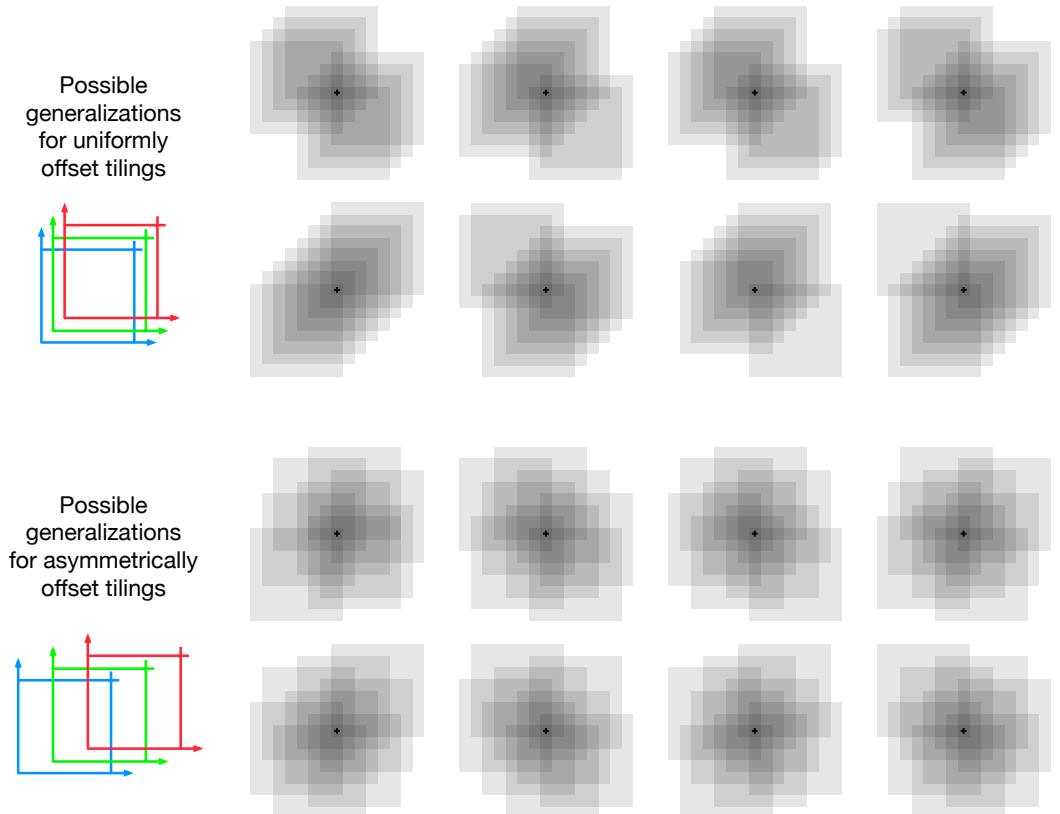


Figure 9.11: Why tile asymmetrical offsets are preferred in tile coding. Shown is the strength of generalization from a trained state, indicated by the small black plus, to nearby states, for the case of eight tilings. If the tilings are uniformly offset (above), then there are diagonal artifacts and substantial variations in the generalization, whereas with asymmetrically offset tilings the generalization is more spherical and homogeneous.

Tilings in all cases are offset from each other by a fraction of a tile width in each dimension. If w denotes the tile width and n the number of tilings, then $\frac{w}{n}$ is a fundamental unit. Within small squares $\frac{w}{n}$ on a side, all states activate the same tiles, have the same feature representation, and the same approximated value. If a state is moved by $\frac{w}{n}$ in any cartesian direction, the feature representation changes by one component/tile. Uniformly offset tilings are offset from each other by exactly this unit distance. For a two-dimensional space, we say that each tiling is offset by the displacement vector $(1, 1)$, meaning that it is offset from the previous tiling by $\frac{w}{n}$ times this vector. In these terms, the asymmetrically offset tilings shown in the lower part of Figure 9.11 are offset by a displacement vector of $(1, 3)$.

Extensive studies have been made of the effect of different displacement vectors on the generalization of tile coding (Parks and Militzer, 1991; An, 1991; An, Miller and Parks,

1991; Miller, An, Glanz and Carter, 1990), assessing their homogeneity and tendency toward diagonal artifacts like those seen for the $(1, 1)$ displacement vectors. Based on this work, Miller and Glanz (1996) recommend using displacement vectors consisting of the first odd integers. In particular, for a continuous space of dimension k , a good choice is to use the first odd integers $(1, 3, 5, 7, \dots, 2k - 1)$, with n (the number of tilings) set to an integer power of 2 greater than or equal to $4k$. This is what we have done to produce the tilings in the lower half of Figure 9.11, in which $k = 2$, $n = 2^3 \geq 4k$, and the displacement vector is $(1, 3)$. In a three-dimensional case, the first four tilings would be offset in total from a base position by $(0, 0, 0)$, $(1, 3, 5)$, $(2, 6, 10)$, and $(3, 9, 15)$. Open-source software that can efficiently make tilings like this for any k is readily available.

In choosing a tiling strategy, one has to pick the number of the tilings and the shape of the tiles. The number of tilings, along with the size of the tiles, determines the resolution or fineness of the asymptotic approximation, as in general coarse coding and illustrated in Figure 9.8. The shape of the tiles will determine the nature of generalization as in Figure 9.7. Square tiles will generalize roughly equally in each dimension as indicated in Figure 9.11 (lower). Tiles that are elongated along one dimension, such as the stripe tilings in Figure 9.12 (middle), will promote generalization along that dimension. The tilings in Figure 9.12 (middle) are also denser and thinner on the left, promoting discrimination along the horizontal dimension at lower values along that dimension. The diagonal stripe tiling in Figure 9.12 (right) will promote generalization along one diagonal. In higher dimensions, axis-aligned stripes correspond to ignoring some of the dimensions in some of the tilings, that is, to hyperplanar slices. Irregular tilings such as shown in Figure 9.12 (left) are also possible, though rare in practice and beyond the standard software.

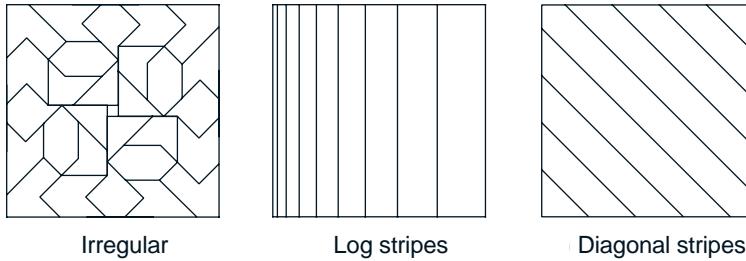
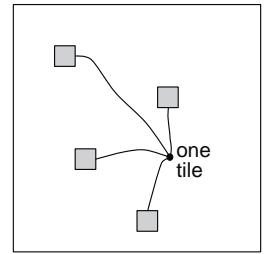


Figure 9.12: Tilings need not be grids. They can be arbitrarily shaped and non-uniform, while still in many cases being computationally efficient to compute.

In practice, it is often desirable to use different shaped tiles in different tilings. For example, one might use some vertical stripe tilings and some horizontal stripe tilings. This would encourage generalization along either dimension. However, with stripe tilings alone it is not possible to learn that a particular conjunction of horizontal and vertical coordinates has a distinctive value (whatever is learned for it will bleed into states with the same horizontal and vertical coordinates). For this one needs the conjunctive rectangular tiles such as originally shown in Figure 9.9. With multiple tilings—some horizontal, some vertical, and some conjunctive—one can get everything: a preference for generalizing along each dimension, yet the ability to learn specific values for conjunctions (see Sutton,

1996 for examples). The choice of tilings determines generalization, and until this choice can be effectively automated, it is important that tile coding enables the choice to be made flexibly and in a way that makes sense to people.

Another useful trick for reducing memory requirements is *hashing*—a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles. Hashing produces tiles consisting of noncontiguous, disjoint regions randomly spread throughout the state space, but that still form an exhaustive partition. For example, one tile might consist of the four subtiles shown to the right. Through hashing, memory requirements are often reduced by large factors with little loss of performance. This is possible because high resolution is needed in only a small fraction of the state space. Hashing frees us from the curse of dimensionality in the sense that memory requirements need not be exponential in the number of dimensions, but need merely match the real demands of the task. Open-source implementations of tile coding commonly include efficient hashing.



Exercise 9.4 Suppose we believe that one of two state dimensions is more likely to have an effect on the value function than is the other, that generalization should be primarily across this dimension rather than along it. What kind of tilings could be used to take advantage of this prior knowledge? □

9.5.5 Radial Basis Functions

Radial basis functions (RBFs) are the natural generalization of coarse coding to continuous-valued features. Rather than each feature being either 0 or 1, it can be anything in the interval $[0, 1]$, reflecting various *degrees* to which the feature is present. A typical RBF feature, x_i , has a Gaussian (bell-shaped) response $x_i(s)$ dependent only on the distance between the state, s , and the feature's prototypical or center state, c_i , and relative to the feature's width, σ_i :

$$x_i(s) \doteq \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right).$$

The norm or distance metric of course can be chosen in whatever way seems most appropriate to the states and task at hand. The figure below shows a one-dimensional example with a Euclidean distance metric.

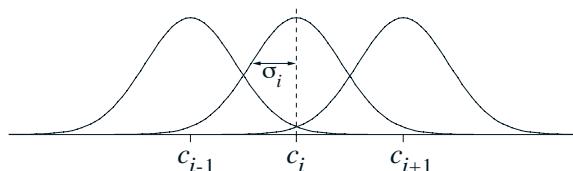


Figure 9.13: One-dimensional radial basis functions.

The primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable. Although this is appealing, in most cases it has no practical significance. Nevertheless, extensive studies have been made of graded response functions such as RBFs in the context of tile coding (An, 1991; Miller et al., 1991; An et al., 1991; Lane, Handelman and Gelfand, 1992). All of these methods require substantial additional computational complexity (over tile coding) and often reduce performance when there are more than two state dimensions. In high dimensions the edges of tiles are much more important, and it has proven difficult to obtain well controlled graded tile activations near the edges.

An *RBF network* is a linear function approximator using RBFs for its features. Learning is defined by equations (9.7) and (9.8), exactly as in other linear function approximators. In addition, some learning methods for RBF networks change the centers and widths of the features as well, bringing them into the realm of nonlinear function approximators. Nonlinear methods may be able to fit target functions much more precisely. The downside to RBF networks, and to nonlinear RBF networks especially, is greater computational complexity and, often, more manual tuning before learning is robust and efficient.

9.6 Selecting Step-Size Parameters Manually

Most SGD methods require the designer to select an appropriate step-size parameter α . Ideally this selection would be automated, and in some cases it has been, but for most cases it is still common practice to set it manually. To do this, and to better understand the algorithms, it is useful to develop some intuitive sense of the role of the step-size parameter. Can we say in general how it should be set?

Theoretical considerations are unfortunately of little help. The theory of stochastic approximation gives us conditions (2.7) on a slowly decreasing step-size sequence that are sufficient to guarantee convergence, but these tend to result in learning that is too slow. The classical choice $\alpha_t = 1/t$, which produces sample averages in tabular MC methods, is not appropriate for TD methods, for nonstationary problems, or for any method using function approximation. For linear methods, there are recursive least-squares methods that set an optimal *matrix* step size, and these methods can be extended to temporal-difference learning as in the LSTD method described in Section 9.8, but these require $O(d^2)$ step-size parameters, or d times more parameters than we are learning. For this reason we rule them out for use on large problems where function approximation is most needed.

To get some intuitive feel for how to set the step-size parameter manually, it is best to go back momentarily to the tabular case. There we can understand that a step size of $\alpha = 1$ will result in a complete elimination of the sample error after one target (see (2.4) with a step size of one). As discussed on page 201, we usually want to learn slower than this. In the tabular case, a step size of $\alpha = \frac{1}{10}$ would take about 10 experiences to converge approximately to their mean target, and if we wanted to learn in 100 experiences we would use $\alpha = \frac{1}{100}$. In general, if $\alpha = \frac{1}{\tau}$, then the tabular estimate for a state will approach the mean of its targets, with the most recent targets having the greatest effect, after about τ experiences with the state.

With general function approximation there is not such a clear notion of *number of experiences* with a state, as each state may be similar to and dissimilar from all the others to various degrees. However, there is a similar rule that gives similar behavior in the case of linear function approximation. Suppose you wanted to learn in about τ experiences with substantially the same feature vector. A good rule of thumb for setting the step-size parameter of linear SGD methods is then

$$\alpha \doteq (\tau \mathbb{E}[\mathbf{x}^\top \mathbf{x}])^{-1}, \quad (9.19)$$

where \mathbf{x} is a random feature vector chosen from the same distribution as input vectors will be in the SGD. This method works best if the feature vectors do not vary greatly in length; ideally $\mathbf{x}^\top \mathbf{x}$ is a constant.

Exercise 9.5 Suppose you are using tile coding to transform a seven-dimensional continuous state space into binary feature vectors to estimate a state value function $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$. You believe that the dimensions do not interact strongly, so you decide to use eight tilings of each dimension separately (stripe tilings), for $7 \times 8 = 56$ tilings. In addition, in case there are some pairwise interactions between the dimensions, you also take all $\binom{7}{2} = 21$ pairs of dimensions and tile each pair conjunctively with rectangular tiles. You make two tilings for each pair of dimensions, making a grand total of $21 \times 2 + 56 = 98$ tilings. Given these feature vectors, you suspect that you still have to average out some noise, so you decide that you want learning to be gradual, taking about 10 presentations with the same feature vector before learning nears its asymptote. What step-size parameter α should you use? Why? \square

Exercise 9.6 If $\tau = 1$ and $\mathbf{x}(S_t)^\top \mathbf{x}(S_t) = \mathbb{E}[\mathbf{x}^\top \mathbf{x}]$, prove that (9.19) together with (9.7) and linear function approximation results in the error being reduced to zero in one update.

9.7 Nonlinear Function Approximation: Artificial Neural Networks

Artificial neural networks (ANNs) are widely used for nonlinear function approximation. An ANN is a network of interconnected units that have some of the properties of neurons, the main components of nervous systems. ANNs have a long history, with the latest advances in training deeply-layered ANNs (deep learning) being responsible for some of the most impressive abilities of machine learning systems, including reinforcement learning systems. In Chapter 16 we describe several impressive examples of reinforcement learning systems that use ANN function approximation.

Figure 9.14 shows a generic feedforward ANN, meaning that there are no loops in the network, that is, there are no paths within the network by which a unit's output can influence its input. The network in the figure has an output layer consisting of two output units, an input layer with four input units, and two “hidden layers”: layers that are neither input nor output layers. A real-valued weight is associated with each link. A weight roughly corresponds to the efficacy of a synaptic connection in a real neural network (see Section 15.1). If an ANN has at least one loop in its connections, it is a recurrent rather

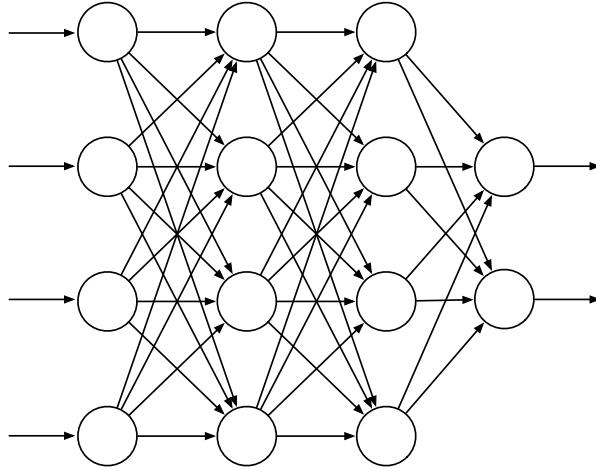


Figure 9.14: A generic feedforward ANN with four input units, two output units, and two hidden layers.

than a feedforward ANN. Although both feedforward and recurrent ANNs have been used in reinforcement learning, here we look only at the simpler feedforward case.

The units (the circles in Figure 9.14) are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result a nonlinear function, called the *activation function*, to produce the unit’s output, or activation. Different activation functions are used, but they are typically S-shaped, or sigmoid, functions such as the logistic function $f(x) = 1/(1 + e^{-x})$, though sometimes the rectifier nonlinearity $f(x) = \max(0, x)$ is used. A step function like $f(x) = 1$ if $x \geq \theta$, and 0 otherwise, results in a binary unit with threshold θ . The units in a network’s input layer are somewhat different in having their activations set to externally-supplied values that are the inputs to the function the network is approximating.

The activation of each output unit of a feedforward ANN is a nonlinear function of the activation patterns over the network’s input units. The functions are parameterized by the network’s connection weights. An ANN with no hidden layers can represent only a very small fraction of the possible input-output functions. However an ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network’s input space to any degree of accuracy (Cybenko, 1989). This is also true for other nonlinear activation functions that satisfy mild conditions, but nonlinearity is essential: if all the units in a multi-layer feedforward ANN have linear activation functions, the entire network is equivalent to a network with no hidden layers (because linear functions of linear functions are themselves linear).

Despite this “universal approximation” property of one-hidden-layer ANNs, both experience and theory show that approximating the complex functions needed for many artificial intelligence tasks is made easier—indeed may require—abstractions that are hierarchical compositions of many layers of lower-level abstractions, that is, abstractions

produced by deep architectures such as ANNs with many hidden layers. (See Bengio, 2009, for a thorough review.) The successive layers of a deep ANN compute increasingly abstract representations of the network’s “raw” input, with each unit providing a feature contributing to a hierarchical representation of the overall input-output function of the network.

Training the hidden layers of an ANN is therefore a way to automatically create features appropriate for a given problem so that hierarchical representations can be produced without relying exclusively on hand-crafted features. This has been an enduring challenge for artificial intelligence and explains why learning algorithms for ANNs with hidden layers have received so much attention over the years. ANNs typically learn by a stochastic gradient method (Section 9.3). Each weight is adjusted in a direction aimed at improving the network’s overall performance as measured by an objective function to be either minimized or maximized. In the most common supervised learning case, the objective function is the expected error, or loss, over a set of labeled training examples. In reinforcement learning, ANNs can use TD errors to learn value functions, or they can aim to maximize expected reward as in a gradient bandit (Section 2.8) or a policy-gradient algorithm (Chapter 13). In all of these cases it is necessary to estimate how a change in each connection weight would influence the network’s overall performance, in other words, to estimate the partial derivative of an objective function with respect to each weight, given the current values of all the network’s weights. The gradient is the vector of these partial derivatives.

The most successful way to do this for ANNs with hidden layers (provided the units have differentiable activation functions) is the backpropagation algorithm, which consists of alternating forward and backward passes through the network. Each forward pass computes the activation of each unit given the current activations of the network’s input units. After each forward pass, a backward pass efficiently computes a partial derivative for each weight. (As in other stochastic gradient learning algorithms, the vector of these partial derivatives is an estimate of the true gradient.) In Section 15.10 we discuss methods for training ANNs with hidden layers that use reinforcement learning principles instead of backpropagation. These methods are less efficient than the backpropagation algorithm, but they may be closer to how real neural networks learn.

The backpropagation algorithm can produce good results for shallow networks having 1 or 2 hidden layers, but it may not work well for deeper ANNs. In fact, training a network with $k + 1$ hidden layers can actually result in poorer performance than training a network with k hidden layers, even though the deeper network can represent all the functions that the shallower network can (Bengio, 2009). Explaining results like these is not easy, but several factors are important. First, the large number of weights in a typical deep ANN makes it difficult to avoid the problem of overfitting, that is, the problem of failing to generalize correctly to cases on which the network has not been trained. Second, backpropagation does not work well for deep ANNs because the partial derivatives computed by its backward passes either decay rapidly toward the input side of the network, making learning by deep layers extremely slow, or the partial derivatives grow rapidly toward the input side of the network, making learning unstable. Methods for dealing with these problems are largely responsible for many impressive recent results

achieved by systems that use deep ANNs.

Overfitting is a problem for any function approximation method that adjusts functions with many degrees of freedom on the basis of limited training data. It is less of a problem for online reinforcement learning that does not rely on limited training sets, but generalizing effectively is still an important issue. Overfitting is a problem for ANNs in general, but especially so for deep ANNs because they tend to have very large numbers of weights. Many methods have been developed for reducing overfitting. These include stopping training when performance begins to decrease on validation data different from the training data (cross validation), modifying the objective function to discourage complexity of the approximation (regularization), and introducing dependencies among the weights to reduce the number of degrees of freedom (e.g., weight sharing).

A particularly effective method for reducing overfitting by deep ANNs is the dropout method introduced by Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014). During training, units are randomly removed from the network (dropped out) along with their connections. This can be thought of as training a large number of “thinned” networks. Combining the results of these thinned networks at test time is a way to improve generalization performance. The dropout method efficiently approximates this combination by multiplying each outgoing weight of a unit by the probability that that unit was retained during training. Srivastava et al. found that this method significantly improves generalization performance. It encourages individual hidden units to learn features that work well with random collections of other features. This increases the versatility of the features formed by the hidden units so that the network does not overly specialize to rarely-occurring cases.

Hinton, Osindero, and Teh (2006) took a major step toward solving the problem of training the deep layers of a deep ANN in their work with deep belief networks, layered networks closely related to the deep ANNs discussed here. In their method, the deepest layers are trained one at a time using an unsupervised learning algorithm. Without relying on the overall objective function, unsupervised learning can extract features that capture statistical regularities of the input stream. The deepest layer is trained first, then with input provided by this trained layer, the next deepest layer is trained, and so on, until the weights in all, or many, of the network’s layers are set to values that now act as initial values for supervised learning. The network is then fine-tuned by backpropagation with respect to the overall objective function. Studies show that this approach generally works much better than backpropagation with weights initialized with random values. The better performance of networks trained with weights initialized this way could be due to many factors, but one idea is that this method places the network in a region of weight space from which a gradient-based algorithm can make good progress.

Batch normalization (Ioffe and Szegedy, 2015) is another technique that makes it easier to train deep ANNs. It has long been known that ANN learning is easier if the network input is normalized, for example, by adjusting each input variable to have zero mean and unit variance. Batch normalization for training deep ANNs normalizes the output of deep layers before they feed into the following layer. Ioffe and Szegedy (2015) used statistics from subsets, or “mini-batches,” of training examples to normalize these between-layer signals to improve the learning rate of deep ANNs.

Another technique useful for training deep ANNs is *deep residual learning* (He, Zhang, Ren, and Sun, 2016). Sometimes it is easier to learn how a function differs from the identity function than to learn the function itself. Then adding this difference, or residual function, to the input produces the desired function. In deep ANNs, a block of layers can be made to learn a residual function simply by adding shortcut, or skip, connections around the block. These connections add the input to the block to its output, and no additional weights are needed. He et al. (2016) evaluated this method using deep convolutional networks with skip connections around every pair of adjacent layers, finding substantial improvement over networks without the skip connections on benchmark image classification tasks. Both batch normalization and deep residual learning were used in the reinforcement learning application to the game of Go that we describe in Chapter 16.

A type of deep ANN that has proven to be very successful in applications, including impressive reinforcement learning applications (Chapter 16), is the *deep convolutional network*. This type of network is specialized for processing high-dimensional data arranged in spatial arrays, such as images. It was inspired by how early visual processing works in the brain (LeCun, Bottou, Bengio and Haffner, 1998). Because of its special architecture, a deep convolutional network can be trained by backpropagation without resorting to methods like those described above to train the deep layers.

Figure 9.15 illustrates the architecture of a deep convolutional network. This instance, from LeCun et al. (1998), was designed to recognize hand-written characters. It consists of alternating convolutional and subsampling layers, followed by several fully connected final layers. Each convolutional layer produces a number of feature maps. A feature map is a pattern of activity over an array of units, where each unit performs the same operation on data in its receptive field, which is the part of the data it “sees” from the preceding layer (or from the external input in the case of the first convolutional layer). The units of a feature map are identical to one another except that their receptive fields, which are all the same size and shape, are shifted to different locations on the arrays of incoming data. Units in the same feature map share the same weights. This means that a feature map detects the same feature no matter where it is located in the input

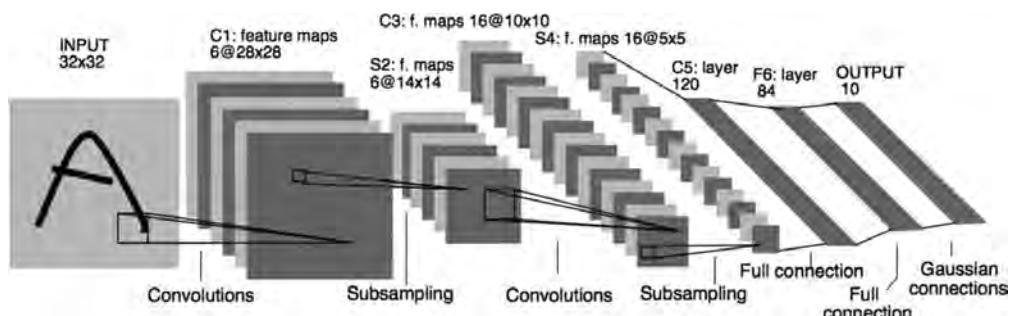


Figure 9.15: Deep Convolutional Network. Republished with permission of Proceedings of the IEEE, from Gradient-based learning applied to document recognition, LeCun, Bottou, Bengio, and Haffner, volume 86, 1998; permission conveyed through Copyright Clearance Center, Inc.

array. In the network in Figure 9.15, for example, the first convolutional layer produces 6 feature maps, each consisting of 28×28 units. Each unit in each feature map has a 5×5 receptive field, and these receptive fields overlap (in this case by four columns and four rows). Consequently, each of the 6 feature maps is specified by just 25 adjustable weights.

The subsampling layers of a deep convolutional network reduce the spatial resolution of the feature maps. Each feature map in a subsampling layer consists of units that average over a receptive field of units in the feature maps of the preceding convolutional layer. For example, each unit in each of the 6 feature maps in the first subsampling layer of the network of Figure 9.15 averages over a 2×2 non-overlapping receptive field over one of the feature maps produced by the first convolutional layer, resulting in six 14×14 feature maps. Subsampling layers reduce the network’s sensitivity to the spatial locations of the features detected, that is, they help make the network’s responses spatially invariant. This is useful because a feature detected at one place in an image is likely to be useful at other places as well.

Advances in the design and training of ANNs—of which we have only mentioned a few—all contribute to reinforcement learning. Although current reinforcement learning theory is mostly limited to methods using tabular or linear function approximation methods, the impressive performances of notable reinforcement learning applications owe much of their success to nonlinear function approximation by multi-layer ANNs. We discuss several of these applications in Chapter 16.

9.8 Least-Squares TD

All the methods we have discussed so far in this chapter have required computation per time step proportional to the number of parameters. With more computation, however, one can do better. In this section we present a method for linear function approximation that is arguably the best that can be done for this case.

As we established in Section 9.4 TD(0) with linear function approximation converges asymptotically (for appropriately decreasing step sizes) to the TD fixed point:

$$\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b},$$

where

$$\mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top] \quad \text{and} \quad \mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t].$$

Why, one might ask, must we compute this solution iteratively? This is wasteful of data! Could one not do better by computing estimates of \mathbf{A} and \mathbf{b} , and then directly computing the TD fixed point? The *Least-Squares TD* algorithm, commonly known as *LSTD*, does exactly this. It forms the natural estimates

$$\hat{\mathbf{A}}_t \doteq \sum_{k=0}^{t-1} \mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^\top + \varepsilon\mathbf{I} \quad \text{and} \quad \hat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{k+1}\mathbf{x}_k, \quad (9.20)$$

where \mathbf{I} is the identity matrix, and $\varepsilon\mathbf{I}$, for some small $\varepsilon > 0$, ensures that $\widehat{\mathbf{A}}_t$ is always invertible. It might seem that these estimates should both be divided by t , and indeed they should; as defined here, these are really estimates of t times \mathbf{A} and t times \mathbf{b} . However, the extra t factors cancel out when LSTD uses these estimates to estimate the TD fixed point as

$$\mathbf{w}_t \doteq \widehat{\mathbf{A}}_t^{-1} \widehat{\mathbf{b}}_t. \quad (9.21)$$

This algorithm is the most data efficient form of linear TD(0), but it is also more expensive computationally. Recall that semi-gradient TD(0) requires memory and per-step computation that is only $O(d)$.

How complex is LSTD? As it is written above the complexity seems to increase with t , but the two approximations in (9.20) could be implemented incrementally using the techniques we have covered earlier (e.g., in Chapter 2) so that they can be done in constant time per step. Even so, the update for $\widehat{\mathbf{A}}_t$ would involve an outer product (a column vector times a row vector) and thus would be a matrix update; its computational complexity would be $O(d^2)$, and of course the memory required to hold the $\widehat{\mathbf{A}}_t$ matrix would be $O(d^2)$.

A potentially greater problem is that our final computation (9.21) uses the inverse of $\widehat{\mathbf{A}}_t$, and the computational complexity of a general inverse computation is $O(d^3)$. Fortunately, an inverse of a matrix of our special form—a sum of outer products—can also be updated incrementally with only $O(d^2)$ computations, as

$$\begin{aligned} \widehat{\mathbf{A}}_t^{-1} &= \left(\widehat{\mathbf{A}}_{t-1} + \mathbf{x}_{t-1}(\mathbf{x}_{t-1} - \gamma \mathbf{x}_t)^\top \right)^{-1} && \text{(from (9.20))} \\ &= \widehat{\mathbf{A}}_{t-1}^{-1} - \frac{\widehat{\mathbf{A}}_{t-1}^{-1} \mathbf{x}_{t-1} (\mathbf{x}_{t-1} - \gamma \mathbf{x}_t)^\top \widehat{\mathbf{A}}_{t-1}^{-1}}{1 + (\mathbf{x}_{t-1} - \gamma \mathbf{x}_t)^\top \widehat{\mathbf{A}}_{t-1}^{-1} \mathbf{x}_{t-1}}, \end{aligned} \quad (9.22)$$

for $t > 0$, with $\widehat{\mathbf{A}}_0 \doteq \varepsilon\mathbf{I}$. Although the identity (9.22), known as *the Sherman-Morrison formula*, is superficially complicated, it involves only vector-matrix and vector-vector multiplications and thus is only $O(d^2)$. Thus we can store the inverse matrix $\widehat{\mathbf{A}}_t^{-1}$, maintain it with (9.22), and then use it in (9.21), all with only $O(d^2)$ memory and per-step computation. The complete algorithm is given in the box on the next page.

Of course, $O(d^2)$ is still significantly more expensive than the $O(d)$ of semi-gradient TD. Whether the greater data efficiency of LSTD is worth this computational expense depends on how large d is, how important it is to learn quickly, and the expense of other parts of the system. The fact that LSTD requires no step-size parameter is sometimes also touted, but the advantage of this is probably overstated. LSTD does not require a step size, but it does require ε ; if ε is chosen too small the sequence of inverses can vary wildly, and if ε is chosen too large then learning is slowed. In addition, LSTD's lack of a step-size parameter means that it never forgets. This is sometimes desirable, but it is problematic if the target policy π changes as it does in reinforcement learning and GPI. In control applications, LSTD typically has to be combined with some other mechanism to induce forgetting, mooting any initial advantage of not requiring a step-size parameter.

LSTD for estimating $\hat{v} = \mathbf{w}^\top \mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x} : \mathcal{S}^+ \rightarrow \mathbb{R}^d$ such that $\mathbf{x}(\text{terminal}) = \mathbf{0}$

Algorithm parameter: small $\varepsilon > 0$

$$\widehat{\mathbf{A}}^{-1} \leftarrow \varepsilon^{-1} \mathbf{I}$$

A $d \times d$ matrix

$$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$$

A d -dimensional vector

Loop for each episode:

 Initialize S ; $\mathbf{x} \leftarrow \mathbf{x}(S)$

 Loop for each step of episode:

 Choose and take action $A \sim \pi(\cdot | S)$, observe R, S' ; $\mathbf{x}' \leftarrow \mathbf{x}(S')$

$$\mathbf{v} \leftarrow \widehat{\mathbf{A}}^{-1}^\top (\mathbf{x} - \gamma \mathbf{x}')$$

$$\widehat{\mathbf{A}}^{-1} \leftarrow \widehat{\mathbf{A}}^{-1} - (\widehat{\mathbf{A}}^{-1} \mathbf{x}) \mathbf{v}^\top / (1 + \mathbf{v}^\top \mathbf{x})$$

$$\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R \mathbf{x}$$

$$\mathbf{w} \leftarrow \widehat{\mathbf{A}}^{-1} \widehat{\mathbf{b}}$$

$$S \leftarrow S'; \mathbf{x} \leftarrow \mathbf{x}'$$

 until S' is terminal

9.9 Memory-based Function Approximation

So far we have discussed the *parametric* approach to approximating value functions. In this approach, a learning algorithm adjusts the parameters of a functional form intended to approximate the value function over a problem's entire state space. Each update, $s \mapsto g$, is a training example used by the learning algorithm to change the parameters with the aim of reducing the approximation error. After the update, the training example can be discarded (although it might be saved to be used again). When an approximate value of a state (which we will call the *query state*) is needed, the function is simply evaluated at that state using the latest parameters produced by the learning algorithm.

Memory-based function approximation methods are very different. They simply save training examples in memory as they arrive (or at least save a subset of the examples) without updating any parameters. Then, whenever a query state's value estimate is needed, a set of examples is retrieved from memory and used to compute a value estimate for the query state. This approach is sometimes called *lazy learning* because processing training examples is postponed until the system is queried to provide an output.

Memory-based function approximation methods are prime examples of *nonparametric* methods. Unlike parametric methods, the approximating function's form is not limited to a fixed parameterized class of functions, such as linear functions or polynomials, but is instead determined by the training examples themselves, together with some means for combining them to output estimated values for query states. As more training examples accumulate in memory, one expects nonparametric methods to produce increasingly accurate approximations of any target function.

There are many different memory-based methods depending on how the stored training examples are selected and how they are used to respond to a query. Here, we focus on *local-learning* methods that approximate a value function only locally in the neighborhood of the current query state. These methods retrieve a set of training examples from memory whose states are judged to be the most relevant to the query state, where relevance usually depends on the distance between states: the closer a training example's state is to the query state, the more relevant it is considered to be, where distance can be defined in many different ways. After the query state is given a value, the local approximation is discarded.

The simplest example of the memory-based approach is the *nearest neighbor* method, which simply finds the example in memory whose state is closest to the query state and returns that example's value as the approximate value of the query state. In other words, if the query state is s , and $s' \mapsto g$ is the example in memory in which s' is the closest state to s , then g is returned as the approximate value of s . Slightly more complicated are *weighted average* methods that retrieve a set of nearest neighbor examples and return a weighted average of their target values, where the weights generally decrease with increasing distance between their states and the query state. *Locally weighted regression* is similar, but it fits a surface to the values of a set of nearest states by means of a parametric approximation method that minimizes a weighted error measure like (9.1), where the weights depend on distances from the query state. The value returned is the evaluation of the locally-fitted surface at the query state, after which the local approximation surface is discarded.

Being nonparametric, memory-based methods have the advantage over parametric methods of not limiting approximations to pre-specified functional forms. This allows accuracy to improve as more data accumulates. Memory-based *local* approximation methods have other properties that make them well suited for reinforcement learning. Because trajectory sampling is of such importance in reinforcement learning, as discussed in Section 8.6, memory-based local methods can focus function approximation on local neighborhoods of states (or state-action pairs) visited in real or simulated trajectories. There may be no need for global approximation because many areas of the state space will never (or almost never) be reached. In addition, memory-based methods allow an agent's experience to have a relatively immediate affect on value estimates in the neighborhood of the current state, in contrast with a parametric method's need to incrementally adjust parameters of a global approximation.

Avoiding global approximation is also a way to address the curse of dimensionality. For example, for a state space with k dimensions, a tabular method storing a global approximation requires memory exponential in k . On the other hand, in storing examples for a memory-based method, each example requires memory proportional to k , and the memory required to store, say, n examples is linear in n . Nothing is exponential in k or n . Of course, the critical remaining issue is whether a memory-based method can answer queries quickly enough to be useful to an agent. A related concern is how speed degrades as the size of the memory grows. Finding nearest neighbors in a large database can take too long to be practical in many applications.

Proponents of memory-based methods have developed ways to accelerate the nearest neighbor search. Using parallel computers or special purpose hardware is one approach; another is the use of special multi-dimensional data structures to store the training data. One data structure studied for this application is the *k-d tree* (short for *k*-dimensional tree), which recursively splits a *k*-dimensional space into regions arranged as nodes of a binary tree. Depending on the amount of data and how it is distributed over the state space, nearest-neighbor search using *k-d* trees can quickly eliminate large regions of the space in the search for neighbors, making the searches feasible in some problems where naive searches would take too long.

Locally weighted regression additionally requires fast ways to do the local regression computations which have to be repeated to answer each query. Researchers have developed many ways to address these problems, including methods for forgetting entries in order to keep the size of the database within bounds. The Bibliographic and Historical Comments section at the end of this chapter points to some of the relevant literature, including a selection of papers describing applications of memory-based learning to reinforcement learning.

9.10 Kernel-based Function Approximation

Memory-based methods such as the weighted average and locally weighted regression methods described above depend on assigning weights to examples $s' \mapsto g$ in the database depending on the distance between s' and a query states s . The function that assigns these weights is called a *kernel function*, or simply a *kernel*. In the weighted average and locally weighted regressions methods, for example, a kernel function $k : \mathbb{R} \rightarrow \mathbb{R}$ assigns weights to distances between states. More generally, weights do not have to depend on distances; they can depend on some other measure of similarity between states. In this case, $k : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$, so that $k(s, s')$ is the weight given to data about s' in its influence on answering queries about s .

Viewed slightly differently, $k(s, s')$ is a measure of the strength of generalization from s' to s . Kernel functions numerically express how *relevant* knowledge about any state is to any other state. As an example, the strengths of generalization for tile coding shown in Figure 9.11 correspond to different kernel functions resulting from uniform and asymmetrical tile offsets. Although tile coding does not explicitly use a kernel function in its operation, it generalizes according to one. In fact, as we discuss more below, the strength of generalization resulting from linear parametric function approximation can always be described by a kernel function.

Kernel regression is the memory-based method that computes a kernel weighted average of the targets of *all* examples stored in memory, assigning the result to the query state. If \mathcal{D} is the set of stored examples, and $g(s')$ denotes the target for state s' in a stored example, then kernel regression approximates the target function, in this case a value function depending on \mathcal{D} , as

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s')g(s'). \quad (9.23)$$

The weighted average method described above is a special case in which $k(s, s')$ is non-zero only when s and s' are close to one another so that the sum need not be computed over all of \mathcal{D} .

A common kernel is the Gaussian radial basis function (RBF) used in RBF function approximation as described in Section 9.5.5. In the method described there, RBFs are features whose centers and widths are either fixed from the start, with centers presumably concentrated in areas where many examples are expected to fall, or are adjusted in some way during learning. Barring methods that adjust centers and widths, this is a linear parametric method whose parameters are the weights of each RBF, which are typically learned by stochastic gradient, or semi-gradient, descent. The form of the approximation is a linear combination of the pre-determined RBFs. Kernel regression with an RBF kernel differs from this in two ways. First, it is memory-based: the RBFs are centered on the states of the stored examples. Second, it is nonparametric: there are no parameters to learn; the response to a query is given by (9.23).

Of course, many issues have to be addressed for practical implementation of kernel regression, issues that are beyond the scope of our brief discussion. However, it turns out that any linear parametric regression method like those we described in Section 9.4, with states represented by feature vectors $\mathbf{x}(s) = (x_1(s), x_2(s), \dots, x_d(s))^\top$, can be recast as kernel regression where $k(s, s')$ is the inner product of the feature vector representations of s and s' ; that is

$$k(s, s') = \mathbf{x}(s)^\top \mathbf{x}(s'). \quad (9.24)$$

Kernel regression with this kernel function produces the same approximation that a linear parametric method would if it used these feature vectors and learned with the same training data.

We skip the mathematical justification for this, which can be found in any modern machine learning text, such as Bishop (2006), and simply point out an important implication. Instead of constructing features for linear parametric function approximators, one can instead construct kernel functions directly without referring at all to feature vectors. Not all kernel functions can be expressed as inner products of feature vectors as in (9.24), but a kernel function that can be expressed like this can offer significant advantages over the equivalent parametric method. For many sets of feature vectors, (9.24) has a compact functional form that can be evaluated without any computation taking place in the d -dimensional feature space. In these cases, kernel regression is much less complex than directly using a linear parametric method with states represented by these feature vectors. This is the so-called “kernel trick” that allows effectively working in the high-dimension of an expansive feature space while actually working only with the set of stored training examples. The kernel trick is the basis of many machine learning methods, and researchers have shown how it can sometimes benefit reinforcement learning.

9.11 Looking Deeper at On-policy Learning: Interest and Emphasis

The algorithms we have considered so far in this chapter have treated all the states encountered equally, as if they were all equally important. In some cases, however, we are more interested in some states than others. In discounted episodic problems, for example, we may be more interested in accurately valuing early states in the episode than in later states where discounting may have made the rewards much less important to the value of the start state. Or, if an action-value function is being learned, it may be less important to accurately value poor actions whose value is much less than the greedy action. Function approximation resources are always limited, and if they were used in a more targeted way, then performance could be improved.

One reason we have treated all states encountered equally is that then we are updating according to the on-policy distribution, for which stronger theoretical results are available for semi-gradient methods. Recall that the on-policy distribution was defined as the distribution of states encountered in an MDP while following the target policy. Now we will generalize this concept significantly. Rather than having one on-policy distribution for the MDP, we will have many. All of them will have in common that they are a distribution of states encountered in trajectories while following the target policy, but they will vary in how the trajectories are, in a sense, initiated.

We now introduce some new concepts. First we introduce a non-negative scalar measure, a random variable I_t called *interest*, indicating the degree to which we are interested in accurately valuing the state (or state-action pair) at time t . If we don't care at all about the state, then the interest should be zero; if we fully care, it might be one, though it is formally allowed to take any non-negative value. The interest can be set in any causal way; for example, it may depend on the trajectory up to time t or the learned parameters at time t . The distribution μ in the $\overline{\text{VE}}$ (9.1) is then defined as the distribution of states encountered while following the target policy, weighted by the interest. Second, we introduce another non-negative scalar random variable, the *emphasis* M_t . This scalar multiplies the learning update and thus emphasizes or de-emphasizes the learning done at time t . The general n -step learning rule, replacing (9.15), is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha M_t [G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t < T, \quad (9.25)$$

with the n -step return given by (9.16) and the emphasis determined recursively from the interest by:

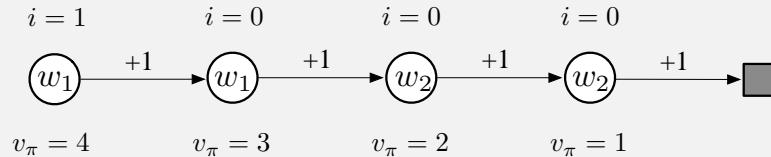
$$M_t = I_t + \gamma^n M_{t-n}, \quad 0 \leq t < T, \quad (9.26)$$

with $M_t \doteq 0$, for all $t < 0$. These equations are taken to include the Monte Carlo case, for which $G_{t:t+n} = G_t$, all the updates are made at end of the episode, $n = T - t$, and $M_t = I_t$.

Example 9.4 illustrates how interest and emphasis can result in more accurate value estimates.

Example 9.4: Interest and Emphasis

To see the potential benefits of using interest and emphasis, consider the four-state Markov reward process shown below:



Episodes start in the leftmost state, then transition one state to the right, with a reward of $+1$, on each step until the terminal state is reached. The true value of the first state is thus 4, of the second state 3, and so on as shown below each state. These are the true values; the estimated values can only approximate these because they are constrained by the parameterization. There are two components to the parameter vector $\mathbf{w} = (w_1, w_2)^\top$, and the parameterization is as written inside each state. The estimated values of the first two states are given by w_1 alone and thus must be the same even though their true values are different. Similarly, the estimated values of the third and fourth states are given by w_2 alone and must be the same even though their true values are different. Suppose that we are interested in accurately valuing only the leftmost state; we assign it an interest of 1 while all the other states are assigned an interest of 0, as indicated above the states.

First consider applying gradient Monte Carlo algorithms to this problem. The algorithms presented earlier in this chapter that do not take into account interest and emphasis (in (9.7) and the box on page 202) will converge (for decreasing step sizes) to the parameter vector $\mathbf{w}_\infty = (3.5, 1.5)$, which gives the first state—the only one we are interested in—a value of 3.5 (i.e., intermediate between the true values of the first and second states). The methods presented in this section that do use interest and emphasis, on the other hand, will learn the value of the first state exactly correctly; w_1 will converge to 4 while w_2 will never be updated because the emphasis is zero in all states save the leftmost.

Now consider applying two-step semi-gradient TD methods. The methods from earlier in this chapter without interest and emphasis (in (9.15) and (9.16) and the box on page 209) will again converge to $\mathbf{w}_\infty = (3.5, 1.5)$, while the methods with interest and emphasis converge to $\mathbf{w}_\infty = (4, 2)$. The latter produces the exactly correct values for the first state and for the third state (which the first state bootstraps from) while never making any updates corresponding to the second or fourth states.

9.12 Summary

Reinforcement learning systems must be capable of *generalization* if they are to be applicable to artificial intelligence or to large engineering applications. To achieve this, any of a broad range of existing methods for *supervised-learning function approximation* can be used simply by treating each update as a training example.

Perhaps the most suitable supervised learning methods are those using *parameterized function approximation*, in which the policy is parameterized by a weight vector \mathbf{w} . Although the weight vector has many components, the state space is much larger still, and we must settle for an approximate solution. We defined the *mean square value error*, $\overline{\text{VE}}(\mathbf{w})$, as a measure of the error in the values $v_{\pi_{\mathbf{w}}}(s)$ for a weight vector \mathbf{w} under the *on-policy distribution*, μ . The $\overline{\text{VE}}$ gives us a clear way to rank different value-function approximations in the on-policy case.

To find a good weight vector, the most popular methods are variations of *stochastic gradient descent* (SGD). In this chapter we have focused on the *on-policy* case with a *fixed policy*, also known as policy evaluation or prediction; a natural learning algorithm for this case is *n-step semi-gradient TD*, which includes gradient Monte Carlo and semi-gradient TD(0) algorithms as the special cases when $n = \infty$ and $n = 1$ respectively. Semi-gradient TD methods are not true gradient methods. In such bootstrapping methods (including DP), the weight vector appears in the update target, yet this is not taken into account in computing the gradient—thus they are *semi-gradient* methods. As such, they cannot rely on classical SGD results.

Nevertheless, good results can be obtained for semi-gradient methods in the special case of *linear* function approximation, in which the value estimates are sums of features times corresponding weights. The linear case is the most well understood theoretically and works well in practice when provided with appropriate features. Choosing the features is one of the most important ways of adding prior domain knowledge to reinforcement learning systems. They can be chosen as polynomials, but this case generalizes poorly in the online learning setting typically considered in reinforcement learning. Better is to choose features according the Fourier basis, or according to some form of coarse coding with sparse overlapping receptive fields. Tile coding is a form of coarse coding that is particularly computationally efficient and flexible. Radial basis functions are useful for one- or two-dimensional tasks in which a smoothly varying response is important. LSTD is the most data-efficient linear TD prediction method, but requires computation proportional to the square of the number of weights, whereas all the other methods are of complexity linear in the number of weights. Nonlinear methods include artificial neural networks trained by backpropagation and variations of SGD; these methods have become very popular in recent years under the name *deep reinforcement learning*.

Linear semi-gradient *n*-step TD is guaranteed to converge under standard conditions, for all n , to a $\overline{\text{VE}}$ that is within a bound of the optimal error (achieved asymptotically by Monte Carlo methods). This bound is always tighter for higher n and approaches zero as $n \rightarrow \infty$. However, in practice very high n results in very slow learning, and some degree of bootstrapping ($n < \infty$) is usually preferable, just as we saw in comparisons of tabular *n*-step methods in Chapter 7 and in comparisons of tabular TD and Monte Carlo methods in Chapter 6.

Exercise 9.7 One of the simplest artificial neural networks consists of a single semi-linear unit with a logistic nonlinearity. The need to handle approximate value functions of this form is common in games that end with either a win or a loss, in which case the value of a state can be interpreted as the probability of winning. Derive the learning algorithm for this case, from (9.7), such that no gradient notation appears.

**Exercise 9.8* Arguably, the squared error used to derive (9.7) is inappropriate for the case treated in the preceding exercise, and the right error measure is the *cross-entropy loss* (which you can find on Wikipedia). Repeat the derivation in Section 9.3, using the cross-entropy loss instead of the squared error in (9.4), all the way to an explicit form with no gradient or logarithm notation in it. Is your final form more complex, or simpler, than that you obtained in the preceding exercise?

Bibliographical and Historical Remarks

Generalization and function approximation have always been an integral part of reinforcement learning. Bertsekas and Tsitsiklis (1996), Bertsekas (2012), and Sugiyama et al. (2013) present the state of the art in function approximation in reinforcement learning. Some of the early work with function approximation in reinforcement learning is discussed at the end of this section.

9.3 Gradient-descent methods for minimizing mean square error in supervised learning are well known. Widrow and Hoff (1960) introduced the least-mean-square (LMS) algorithm, which is the prototypical incremental gradient-descent algorithm. Details of this and related algorithms are provided in many texts (e.g., Widrow and Stearns, 1985; Bishop, 1995; Duda and Hart, 1973).

Semi-gradient TD(0) was first explored by Sutton (1984, 1988), as part of the linear TD(λ) algorithm that we will treat in Chapter 12. The term “semi-gradient” to describe these bootstrapping methods is new to the second edition of this book.

The earliest use of state aggregation in reinforcement learning may have been Michie and Chambers’s BOXES system (1968). The theory of state aggregation in reinforcement learning has been developed by Singh, Jaakkola, and Jordan (1995) and Tsitsiklis and Van Roy (1996). State aggregation has been used in dynamic programming from its earliest days (e.g., Bellman, 1957a).

9.4 Sutton (1988) proved convergence of linear TD(0) in the mean to the minimal \overline{V} solution for the case in which the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, are linearly independent. Convergence with probability 1 was proved by several researchers at about the same time (Peng, 1993; Dayan and Sejnowski, 1994; Tsitsiklis, 1994; Gurvits, Lin, and Hanson, 1994). In addition, Jaakkola, Jordan, and Singh (1994) proved convergence under online updating. All of these results assumed linearly independent feature vectors, which implies at least as many components to \mathbf{w}_t as there are states. Convergence for the more important case of general (dependent) feature vectors was first shown by Dayan (1992). A significant

generalization and strengthening of Dayan’s result was proved by Tsitsiklis and Van Roy (1997). They proved the main result presented in this section, the bound on the asymptotic error of linear bootstrapping methods.

- 9.5** Our presentation of the range of possibilities for linear function approximation is based on that by Barto (1990).
- 9.5.2** Konidaris, Osentoski, and Thomas (2011) introduced the Fourier basis in a simple form suitable for reinforcement learning problems with multi-dimensional continuous state spaces and functions that do not have to be periodic.
- 9.5.3** The term *coarse coding* is due to Hinton (1984), and our Figure 9.6 is based on one of his figures. Waltz and Fu (1965) provide an early example of this type of function approximation in a reinforcement learning system.
- 9.5.4** Tile coding, including hashing, was introduced by Albus (1971, 1981). He described it in terms of his “cerebellar model articulator controller,” or CMAC, as tile coding is sometimes known in the literature. The term “tile coding” was new to the first edition of this book, though the idea of describing CMAC in these terms is taken from Watkins (1989). Tile coding has been used in many reinforcement learning systems (e.g., Shewchuk and Dean, 1990; Lin and Kim, 1991; Miller, Scalera, and Kim, 1994; Sofge and White, 1992; Tham, 1994; Sutton, 1996; Watkins, 1989) as well as in other types of learning control systems (e.g., Kraft and Campagna, 1990; Kraft, Miller, and Dietz, 1992). This section draws heavily on the work of Miller and Glanz (1996). General software for tile coding is available in several languages (e.g., see <http://incompleteideas.net/tiles/tiles3.html>).
- 9.5.5** Function approximation using radial basis functions has received wide attention ever since being related to ANNs by Broomhead and Lowe (1988). Powell (1987) reviewed earlier uses of RBFs, and Poggio and Girosi (1989, 1990) extensively developed and applied this approach.
- 9.6** Automatic methods for adapting the step-size parameter include RMSprop (Tieleman and Hinton, 2012), Adam (Kingma and Ba, 2015), stochastic meta-descent methods such as Delta-Bar-Delta (Jacobs, 1988), its incremental generalization (Sutton, 1992b, c; Mahmood et al., 2012), and nonlinear generalizations (Schraudolph, 1999, 2002). Methods explicitly designed for reinforcement learning include AlphaBound (Dabney and Barto, 2012), SID and NOSID (Dabney, 2014), TIDBD (Kearney et al., in preparation) and the application of stochastic meta-descent to policy gradient learning (Schraudolph, Yu, and Aberdeen, 2006).
- 9.7** The introduction of the threshold logic unit as an abstract model neuron by McCulloch and Pitts (1943) was the beginning of ANNs. The history of ANNs as learning methods for classification or regression has passed through several stages: roughly, the Perceptron (Rosenblatt, 1962) and ADALINE (ADaptive LINEar Element) (Widrow and Hoff, 1960) stage of learning by single-layer ANNs, the

error-backpropagation stage (LeCun, 1985; Rumelhart, Hinton, and Williams, 1986) of learning by multi-layer ANNs, and the current deep-learning stage with its emphasis on representation learning (e.g., Bengio, Courville, and Vincent, 2012; Goodfellow, Bengio, and Courville, 2016). Examples of the many books on ANNs are Haykin (1994), Bishop (1995), and Ripley (2007).

ANNs as function approximation for reinforcement learning goes back to the early work of Farley and Clark (1954), who used reinforcement-like learning to modify the weights of linear threshold functions representing policies. Widrow, Gupta, and Maitra (1973) presented a neuron-like linear threshold unit implementing a learning process they called *learning with a critic* or *selective bootstrap adaptation*, a reinforcement-learning variant of the ADALINE algorithm. Werbos (1987, 1994) developed an approach to prediction and control that uses ANNs trained by error backpropagation to learn policies and value functions using TD-like algorithms. Barto, Sutton, and Brouwer (1981) and Barto and Sutton (1981b) extended the idea of an associative memory network (e.g., Kohonen, 1977; Anderson, Silverstein, Ritz, and Jones, 1977) to reinforcement learning. Barto, Anderson, and Sutton (1982) used a two-layer ANN to learn a nonlinear control policy, and emphasized the first layer's role of learning a suitable representation. Hampson (1983, 1989) was an early proponent of multilayer ANNs for learning value functions. Barto, Sutton, and Anderson (1983) presented an actor-critic algorithm in the form of an ANN learning to balance a simulated pole (see Sections 15.7 and 15.8). Barto and Anandan (1985) introduced a stochastic version of Widrow et al.'s (1973) selective bootstrap algorithm called the *associative reward-penalty (A_{R-P}) algorithm*. Barto (1985, 1986) and Barto and Jordan (1987) described multi-layer ANNs consisting of A_{R-P} units trained with a globally-broadcast reinforcement signal to learn classification rules that are not linearly separable. Barto (1985) discussed this approach to ANNs and how this type of learning rule is related to others in the literature at that time. (See Section 15.10 for additional discussion of this approach to training multi-layer ANNs.) Anderson (1986, 1987, 1989) evaluated numerous methods for training multilayer ANNs and showed that an actor-critic algorithm in which both the actor and critic were implemented by two-layer ANNs trained by error backpropagation outperformed single-layer ANNs in the pole-balancing and tower of Hanoi tasks. Williams (1988) described several ways that backpropagation and reinforcement learning can be combined for training ANNs. Gullapalli (1990) and Williams (1992) devised reinforcement learning algorithms for neuron-like units having continuous, rather than binary, outputs. Barto, Sutton, and Watkins (1990) argued that ANNs can play significant roles for approximating functions required for solving sequential decision problems. Williams (1992) related REINFORCE learning rules (Section 13.3) to the error backpropagation method for training multi-layer ANNs. Tesauro's TD-Gammon (Tesauro 1992, 1994; Section 16.1) influentially demonstrated the learning abilities of $TD(\lambda)$ algorithm with function approximation by multi-layer ANNs in learning to play backgammon. The *AlphaGo*, *AlphaGo Zero*, and *AlphaZero* programs of Silver et al. (2016, 2017a, b; Section 16.6) used reinforcement learning with

deep convolutional ANNs in achieving impressive results with the game of Go. Schmidhuber (2015) reviews applications of ANNs in reinforcement learning, including applications of recurrent ANNs.

- 9.8** LSTD is due to Bradtke and Barto (see Bradtke, 1993, 1994; Bradtke and Barto, 1996; Bradtke, Ydstie, and Barto, 1994), and was further developed by Boyan (1999, 2002), Nedić and Bertsekas (2003), and Yu (2010). The incremental update of the inverse matrix has been known at least since 1949 (Sherman and Morrison, 1949). An extension of least-squares methods to control was introduced by Lagoudakis and Parr (2003; Buşoniu, Lazaric, Ghavamzadeh, Munos, Babuška, and De Schutter, 2012).
- 9.9** Our discussion of memory-based function approximation is largely based on the review of locally weighted learning by Atkeson, Moore, and Schaal (1997). Atkeson (1992) discussed the use of locally weighted regression in memory-based robot learning and supplied an extensive bibliography covering the history of the idea. Stanfill and Waltz (1986) influentially argued for the importance of memory based methods in artificial intelligence, especially in light of parallel architectures then becoming available, such as the Connection Machine. Baird and Klopf (1993) introduced a novel memory-based approach and used it as the function approximation method for Q-learning applied to the pole-balancing task. Schaal and Atkeson (1994) applied locally weighted regression to a robot juggling control problem, where it was used to learn a system model. Peng (1995) used the pole-balancing task to experiment with several nearest-neighbor methods for approximating value functions, policies, and environment models. Tadepalli and Ok (1996) obtained promising results with locally-weighted linear regression to learn a value function for a simulated automatic guided vehicle task. Bottou and Vapnik (1992) demonstrated surprising efficiency of several local learning algorithms compared to non-local algorithms in some pattern recognition tasks, discussing the impact of local learning on generalization.
- Bentley (1975) introduced k -d trees and reported observing average running time of $O(\log n)$ for nearest neighbor search over n records. Friedman, Bentley, and Finkel (1977) clarified the algorithm for nearest neighbor search with k -d trees. Omohundro (1987) discussed efficiency gains possible with hierarchical data structures such as k -d-trees. Moore, Schneider, and Deng (1997) introduced the use of k -d trees for efficient locally weighted regression.
- 9.10** The origin of kernel regression is the *method of potential functions* of Aizerman, Braverman, and Rozonoer (1964). They likened the data to point electric charges of various signs and magnitudes distributed over space. The resulting electric potential over space produced by summing the potentials of the point charges corresponded to the interpolated surface. In this analogy, the kernel function is the potential of a point charge, which falls off as the reciprocal of the distance from the charge. Connell and Utgoff (1987) applied an actor–critic method to the pole-balancing task in which the critic approximated the value function

using kernel regression with an inverse-distance weighting. Predating widespread interest in kernel regression in machine learning, these authors did not use the term kernel, but referred to “Shepard’s method” (Shepard, 1968). Other kernel-based approaches to reinforcement learning include those of Ormoneit and Sen (2002), Dietterich and Wang (2002), Xu, Xie, Hu, and Lu (2005), Taylor and Parr (2009), Barreto, Precup, and Pineau (2011), and Bhat, Farias, and Moallemi (2012).

9.11 For Emphatic-TD methods, see the bibliographical notes to Section 11.8.

The earliest example we know of in which function approximation methods were used for learning value functions was Samuel’s checkers player (1959, 1967). Samuel followed Shannon’s (1950) suggestion that a value function did not have to be exact to be a useful guide to selecting moves in a game and that it might be approximated by a linear combination of features. In addition to linear function approximation, Samuel experimented with lookup tables and hierarchical lookup tables called signature tables (Griffith, 1966, 1974; Page, 1977; Biermann, Fairfield, and Beres, 1982).

At about the same time as Samuel’s work, Bellman and Dreyfus (1959) proposed using function approximation methods with DP. (It is tempting to think that Bellman and Samuel had some influence on one another, but we know of no reference to the other in the work of either.) There is now a fairly extensive literature on function approximation methods and DP, such as multigrid methods and methods using splines and orthogonal polynomials (e.g., Bellman and Dreyfus, 1959; Bellman, Kalaba, and Kotkin, 1963; Daniel, 1976; Whitt, 1978; Reetz, 1977; Schweitzer and Seidmann, 1985; Chow and Tsitsiklis, 1991; Kushner and Dupuis, 1992; Rust, 1996).

Holland’s (1986) classifier system used a selective feature-match technique to generalize evaluation information across state–action pairs. Each classifier matched a subset of states having specified values for a subset of features, with the remaining features having arbitrary values (“wild cards”). These subsets were then used in a conventional state-aggregation approach to function approximation. Holland’s idea was to use a genetic algorithm to evolve a set of classifiers that collectively would implement a useful action-value function. Holland’s ideas influenced the early research of the authors on reinforcement learning, but we focused on different approaches to function approximation. As function approximators, classifiers are limited in several ways. First, they are state-aggregation methods, with concomitant limitations in scaling and in representing smooth functions efficiently. In addition, the matching rules of classifiers can implement only aggregation boundaries that are parallel to the feature axes. Perhaps the most important limitation of conventional classifier systems is that the classifiers are learned via the genetic algorithm, an evolutionary method. As we discussed in Chapter 1, there is available during learning much more detailed information about how to learn than can be used by evolutionary methods. This perspective led us to instead adapt supervised learning methods for use in reinforcement learning, specifically gradient-descent and ANN methods. These differences between Holland’s approach and ours are not surprising because Holland’s ideas were developed during a period when ANNs were generally regarded as being too weak in computational power to be useful, whereas our work was at the beginning of

the period that saw widespread questioning of that conventional wisdom. There remain many opportunities for combining aspects of these different approaches.

Christensen and Korf (1986) experimented with regression methods for modifying coefficients of linear value function approximations in the game of chess. Chapman and Kaelbling (1991) and Tan (1991) adapted decision-tree methods for learning value functions. Explanation-based learning methods have also been adapted for learning value functions, yielding compact representations (Yee, Saxena, Utgoff, and Barto, 1990; Dietterich and Flann, 1995).

Chapter 10

On-policy Control with Approximation

In this chapter we return to the control problem, now with parametric approximation of the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$, where $\mathbf{w} \in \mathbb{R}^d$ is a finite-dimensional weight vector. We continue to restrict attention to the on-policy case, leaving off-policy methods to Chapter 11. The present chapter features the semi-gradient Sarsa algorithm, the natural extension of semi-gradient TD(0) (last chapter) to action values and to on-policy control. In the episodic case, the extension is straightforward, but in the continuing case we have to take a few steps backward and re-examine how we have used discounting to define an optimal policy. Surprisingly, once we have genuine function approximation we have to give up discounting and switch to a new “average-reward” formulation of the control problem, with new “differential” value functions.

Starting first in the episodic case, we extend the function approximation ideas presented in the last chapter from state values to action values. Then we extend them to control following the general pattern of on-policy GPI, using ε -greedy for action selection. We show results for n -step linear Sarsa on the Mountain Car problem. Then we turn to the continuing case and repeat the development of these ideas for the average-reward case with differential values.

10.1 Episodic Semi-gradient Control

The extension of the semi-gradient prediction methods of Chapter 9 to action values is straightforward. In this case it is the approximate action-value function, $\hat{q} \approx q_\pi$, that is represented as a parameterized functional form with weight vector \mathbf{w} . Whereas before we considered random training examples of the form $S_t \mapsto U_t$, now we consider examples of the form $S_t, A_t \mapsto U_t$. The update target U_t can be any approximation of $q_\pi(S_t, A_t)$, including the usual backed-up values such as the full Monte Carlo return (G_t) or any of the n -step Sarsa returns (7.4). The general gradient-descent update for action-value

prediction is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t). \quad (10.1)$$

For example, the update for the one-step Sarsa method is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t). \quad (10.2)$$

We call this method *episodic semi-gradient one-step Sarsa*. For a constant policy, this method converges in the same way that TD(0) does, with the same kind of error bound (9.14).

To form control methods, we need to couple such action-value prediction methods with techniques for policy improvement and action selection. Suitable techniques applicable to continuous actions, or to actions from large discrete sets, are a topic of ongoing research with as yet no clear resolution. On the other hand, if the action set is discrete and not too large, then we can use the techniques already developed in previous chapters. That is, for each possible action a available in the next state S_{t+1} , we can compute $\hat{q}(S_{t+1}, a, \mathbf{w}_t)$ and then find the greedy action $A_{t+1}^* = \arg \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$. Policy improvement is then done (in the on-policy case treated in this chapter) by changing the estimation policy to a soft approximation of the greedy policy such as the ε -greedy policy. Actions are selected according to this same policy. Pseudocode for the complete algorithm is given in the box.

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:

$S, A \leftarrow$ initial state and action of episode (e.g., ε -greedy)

Loop for each step of episode:

Take action A , observe R, S'

If S' is terminal:

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$

Go to next episode

Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε -greedy)

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$

$S \leftarrow S'$

$A \leftarrow A'$

Example 10.1: Mountain Car Task Consider the task of driving an underpowered car up a steep mountain road, as suggested by the diagram in the upper left of Figure 10.1. The difficulty is that gravity is stronger than the car's engine, and even at full throttle the car cannot accelerate up the steep slope. The only solution is to first move away from the goal and up the opposite slope on the left. Then, by applying full throttle the car

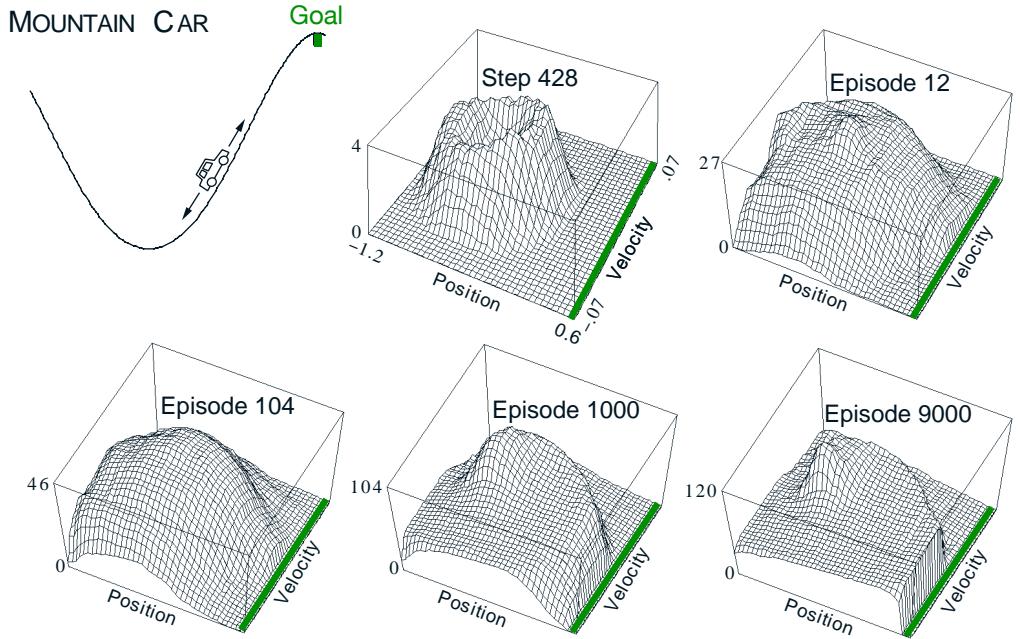


Figure 10.1: The Mountain Car task (upper left panel) and the cost-to-go function ($-\max_a \hat{q}(s, a, \mathbf{w})$) learned during one run.

can build up enough inertia to carry it up the steep slope even though it is slowing down the whole way. This is a simple example of a continuous control task where things have to get worse in a sense (farther from the goal) before they can get better. Many control methodologies have great difficulties with tasks of this kind unless explicitly aided by a human designer.

The reward in this problem is -1 on all time steps until the car moves past its goal position at the top of the mountain, which ends the episode. There are three possible actions: full throttle forward ($+1$), full throttle reverse (-1), and zero throttle (0). The car moves according to a simplified physics. Its position, x_t , and velocity, \dot{x}_t , are updated by

$$x_{t+1} \doteq \text{bound}[x_t + \dot{x}_{t+1}]$$

$$\dot{x}_{t+1} \doteq \text{bound}[\dot{x}_t + 0.001A_t - 0.0025 \cos(3x_t)],$$

where the *bound* operation enforces $-1.2 \leq x_{t+1} \leq 0.5$ and $-0.07 \leq \dot{x}_{t+1} \leq 0.07$. In addition, when x_{t+1} reached the left bound, \dot{x}_{t+1} was reset to zero. When it reached the right bound, the goal was reached and the episode was terminated. Each episode started from a random position $x_t \in [-0.6, -0.4]$ and zero velocity. To convert the two continuous state variables to binary features, we used grid-tilings as in Figure 9.9. We used 8 tilings, with each tile covering $1/8$ th of the bounded distance in each dimension,

and asymmetrical offsets as described in Section 9.5.4.¹ The feature vectors $\mathbf{x}(s, a)$ created by tile coding were then combined linearly with the parameter vector to approximate the action-value function:

$$\hat{q}(s, a, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s, a) = \sum_{i=1}^d w_i \cdot x_i(s, a), \quad (10.3)$$

for each pair of state, s , and action, a .

Figure 10.1 shows what typically happens while learning to solve this task with this form of function approximation.² Shown is the negative of the value function (the *cost-to-go* function) learned on a single run. The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur even though the exploration parameter, ε , was 0. This can be seen in the middle-top panel of the figure, labeled “Step 428”. At this time not even one episode had been completed, but the car has oscillated back and forth in the valley, following circular trajectories in state space. All the states visited frequently are valued worse than unexplored states, because the actual rewards have been worse than what was (unrealistically) expected. This continually drives the agent away from wherever it has been, to explore new states, until a solution is found.

Figure 10.2 shows several learning curves for semi-gradient Sarsa on this problem, with various step sizes.

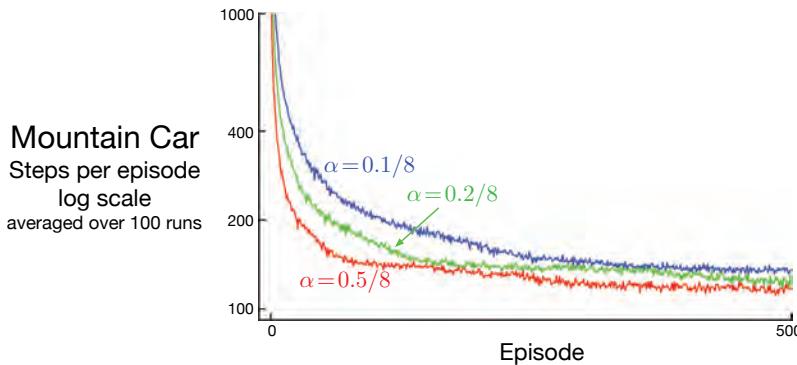


Figure 10.2: Mountain Car learning curves for the semi-gradient Sarsa method with tile-coding function approximation and ε -greedy action selection. ■

¹In particular, we used the tile-coding software, available at <http://incompleteideas.net/tiles/tiles3.html>, with `iht=IHT(4096)` and `tiles(iht,8,[8*x/(0.5+1.2),8*xdot/(0.07+0.07)],A)` to get the indices of the ones in the feature vector for state (x , $xdot$) and action A .

²This data is actually from the “semi-gradient Sarsa(λ)” algorithm that we will not meet until Chapter 12, but semi-gradient Sarsa would behave similarly.

10.2 Semi-gradient n -step Sarsa

We can obtain an n -step version of episodic semi-gradient Sarsa by using an n -step return as the update target in the semi-gradient Sarsa update equation (10.1). The n -step return immediately generalizes from its tabular form (7.4) to a function approximation form:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad t+n < T, \quad (10.4)$$

with $G_{t:t+n} \doteq G_t$ if $t + n \geq T$, as usual. The n -step update equation is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha [G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})] \nabla \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t < T. \quad (10.5)$$

Complete pseudocode is given in the box below.

Episodic semi-gradient n -step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$
 Input: a policy π (if estimating q_π)

Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer n

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

All store and access operations (S_t , A_t , and R_t) can take their index mod $n + 1$

Loop for each episode:

Initialize and store $S_0 \neq \text{terminal}$

Select and store an action $A_0 \sim \pi(\cdot | S_0)$ or ε -greedy wrt $\hat{q}(S_0, \cdot, w)$

$$T \leftarrow \infty$$

Loop for $t = 0, 1, 2, \dots$:

If $t < T$, then:

Take action A_t

Observe and store the ne

S_{t+1} is ter

$$T \leftarrow t + 1$$

else:

Se

$\leftarrow t - n + 1$ (

$\tau \geq 0$:

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$$

If $\tau + n <$
 $\mathbf{w} \leftarrow \mathbf{w} +$

As we have seen before, performance is best if an intermediate level of bootstrapping is used, corresponding to an n larger than 1. Figure 10.3 shows how this algorithm tends to learn faster and obtain a better asymptotic performance at $n=8$ than at $n=1$ on the Mountain Car task. Figure 10.4 shows the results of a more detailed study of the effect of the parameters α and n on the rate of learning on this task.

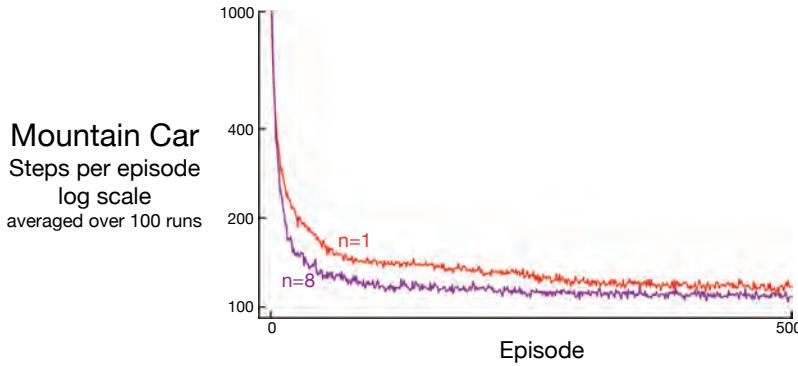


Figure 10.3: Performance of one-step vs 8-step semi-gradient Sarsa on the Mountain Car task. Good step sizes were used: $\alpha = 0.5/8$ for $n = 1$ and $\alpha = 0.3/8$ for $n = 8$.

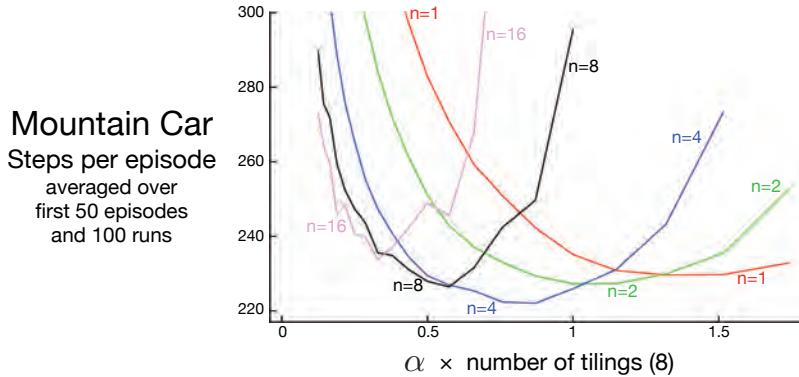


Figure 10.4: Effect of the α and n on early performance of n -step semi-gradient Sarsa and tile-coding function approximation on the Mountain Car task. As usual, an intermediate level of bootstrapping ($n = 4$) performed best. These results are for selected α values, on a log scale, and then connected by straight lines. The standard errors ranged from 0.5 (less than the line width) for $n = 1$ to about 4 for $n = 16$, so the main effects are all statistically significant.

Exercise 10.1 We have not explicitly considered or given pseudocode for any Monte Carlo methods in this chapter. What would they be like? Why is it reasonable not to give pseudocode for them? How would they perform on the Mountain Car task? \square

Exercise 10.2 Give pseudocode for semi-gradient one-step *Expected Sarsa* for control. \square

Exercise 10.3 Why do the results shown in Figure 10.4 have higher standard errors at large n than at small n ? \square

10.3 Average Reward: A New Problem Setting for Continuing Tasks

We now introduce a third classical setting—alongside the episodic and discounted settings—for formulating the goal in Markov decision problems (MDPs). Like the discounted setting, the *average reward* setting applies to continuing problems, problems for which the interaction between agent and environment goes on and on forever without termination or start states. Unlike that setting, however, there is no discounting—the agent cares just as much about delayed rewards as it does about immediate reward. The average-reward setting is one of the major settings commonly considered in the classical theory of dynamic programming and less-commonly in reinforcement learning. As we discuss in the next section, the discounted setting is problematic with function approximation, and thus the average-reward setting is needed to replace it.

In the average-reward setting, the quality of a policy π is defined as the average rate of reward, or simply *average reward*, while following that policy, which we denote as $r(\pi)$:

$$r(\pi) \doteq \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{t=1}^h \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \quad (10.6)$$

$$\begin{aligned} &= \lim_{t \rightarrow \infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi], \\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)r, \end{aligned} \quad (10.7)$$

where the expectations are conditioned on the initial state, S_0 , and on the subsequent actions, A_0, A_1, \dots, A_{t-1} , being taken according to π . The second and third equations hold if the steady-state distribution, $\mu_\pi(s) \doteq \lim_{t \rightarrow \infty} \Pr\{S_t = s \mid A_{0:t-1} \sim \pi\}$, exists and is independent of S_0 , in other words, if the MDP is *ergodic*. In an ergodic MDP, the starting state and any early decision made by the agent can have only a temporary effect; in the long run the expectation of being in a state depends only on the policy and the MDP transition probabilities. Ergodicity is sufficient but not necessary to guarantee the existence of the limit in (10.6).

There are subtle distinctions that can be drawn between different kinds of optimality in the undiscounted continuing case. Nevertheless, for most practical purposes it may be adequate simply to order policies according to their average reward per time step, in other words, according to their $r(\pi)$. This quantity is essentially the average reward under π , as suggested by (10.7), or the *reward rate*. In particular, we consider all policies that attain the maximal value of $r(\pi)$ to be optimal.

Note that the steady state distribution μ_π is the special distribution under which, if you select actions according to π , you remain in the same distribution. That is, for which

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s)p(s'|s,a) = \mu_\pi(s'). \quad (10.8)$$

In the average-reward setting, returns are defined in terms of differences between rewards and the average reward:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots \quad (10.9)$$

This is known as the *differential* return, and the corresponding value functions are known as *differential* value functions. Differential value functions are defined in terms of the new return just as conventional value functions were defined in terms of the discounted return; thus we will use the same notation, $v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s]$ and $q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$ (similarly for v_* and q_*), for differential value functions. Differential value functions also have Bellman equations, just slightly different from those we have seen earlier. We simply remove all γ s and replace all rewards by the difference between the reward and the true average reward:

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a) \left[r - r(\pi) + v_\pi(s') \right], \\ q_\pi(s, a) &= \sum_{r,s'} p(s', r|s, a) \left[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s', a') \right], \\ v_*(s) &= \max_a \sum_{r,s'} p(s', r|s, a) \left[r - \max_\pi r(\pi) + v_*(s') \right], \text{ and} \\ q_*(s, a) &= \sum_{r,s'} p(s', r|s, a) \left[r - \max_\pi r(\pi) + \max_{a'} q_*(s', a') \right] \end{aligned}$$

(cf. (3.14), Exercise 3.17, (3.19), and (3.20)).

There is also a differential form of the two TD errors:

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \quad (10.10)$$

and

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \quad (10.11)$$

where \bar{R}_t is an estimate at time t of the average reward $r(\pi)$. With these alternate definitions, most of our algorithms and many theoretical results carry through to the average-reward setting without change.

For example, an average reward version of semi-gradient Sarsa could be defined just as in (10.2) except with the differential version of the TD error. That is, by

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad (10.12)$$

with δ_t given by (10.11). Pseudocode for a complete algorithm is given in the box on the next page. One limitation of this algorithm is that it does not converge to the differential values but to the differential values plus an arbitrary offset. Notice that the Bellman equations and TD errors given above are unaffected if all the values are shifted by the same amount. Thus, the offset may not matter in practice. How this algorithm could be changed to eliminate the offset is an interesting question for future research.

Exercise 10.4 Give pseudocode for a differential version of semi-gradient Q-learning. \square

Exercise 10.5 What equations are needed (beyond 10.10) to specify the differential version of TD(0)? \square

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameters: step sizes $\alpha, \beta > 0$, small $\varepsilon > 0$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S , and action A

Loop for each step:

 Take action A , observe R, S'

 Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε -greedy)

$$\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$$

$$\bar{R} \leftarrow \bar{R} + \beta \delta$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \nabla \hat{q}(S, A, \mathbf{w})$$

$$S \leftarrow S'$$

$$A \leftarrow A'$$

Exercise 10.6 Suppose there is an MDP that under any policy produces the deterministic sequence of rewards $+1, 0, +1, 0, +1, 0, \dots$ going on forever. Technically, this violates ergodicity; there is no stationary limiting distribution μ_π and the limit (10.7) does not exist. Nevertheless, the average reward (10.6) is well defined. What is it? Now consider two states in this MDP. From A, the reward sequence is exactly as described above, starting with a +1, whereas, from B, the reward sequence starts with a 0 and then continues with $+1, 0, +1, 0, \dots$. We would like to compute the differential values of A and B. Unfortunately, the differential return (10.9) is not well defined when starting from these states as the implicit limit does not exist. To repair this, one could alternatively define the differential value of a state as

$$v_\pi(s) \doteq \lim_{\gamma \rightarrow 1} \lim_{h \rightarrow \infty} \sum_{t=0}^h \gamma^t \left(\mathbb{E}_\pi[R_{t+1} | S_0 = s] - r(\pi) \right). \quad (10.13)$$

Under this definition, what are the differential values of states A and B? \square

Exercise 10.7 Consider a Markov reward process consisting of a ring of three states A, B, and C, with state transitions going deterministically around the ring. A reward of +1 is received upon arrival in A and otherwise the reward is 0. What are the differential values of the three states, using (10.13)? \square

Exercise 10.8 The pseudocode in the box on page 251 updates \bar{R}_t using δ_t as an error rather than simply $R_{t+1} - \bar{R}_t$. Both errors work, but using δ_t is better. To see why, consider the ring MRP of three states from Exercise 10.7. The estimate of the average reward should tend towards its true value of $\frac{1}{3}$. Suppose it was already there and was held stuck there. What would the sequence of $R_{t+1} - \bar{R}_t$ errors be? What would the sequence of δ_t errors be (using Equation 10.10)? Which error sequence would produce a more stable estimate of the average reward if the estimate were allowed to change in response to the errors? Why? \square

Example 10.2: An Access-Control Queuing Task This is a decision task involving access control to a set of 10 servers. Customers of four different priorities arrive at a single queue. If given access to a server, the customers pay a reward of 1, 2, 4, or 8 to the server, depending on their priority, with higher priority customers paying more. In each time step, the customer at the head of the queue is either accepted (assigned to one of the servers) or rejected (removed from the queue, with a reward of zero). In either case, on the next time step the next customer in the queue is considered. The queue never empties, and the priorities of the customers in the queue are uniformly randomly distributed. Of course a customer cannot be served if there is no free server; the customer is always rejected in this case. Each busy server becomes free with probability $p = 0.06$ on each time step. Although we have just described them for definiteness, let us assume the statistics of arrivals and departures are unknown. The task is to decide on each step whether to accept or reject the next customer, on the basis of his priority and the number of free servers, so as to maximize long-term reward without discounting.

In this example we consider a tabular solution to this problem. Although there is no generalization between states, we can still consider it in the general function approximation setting as this setting generalizes the tabular setting. Thus we have a differential action-value estimate for each pair of state (number of free servers and priority of the customer at the head of the queue) and action (accept or reject). Figure 10.5 shows the solution found by differential semi-gradient Sarsa with parameters $\alpha = 0.01$, $\beta = 0.01$, and $\varepsilon = 0.1$. The initial action values and \bar{R} were zero.

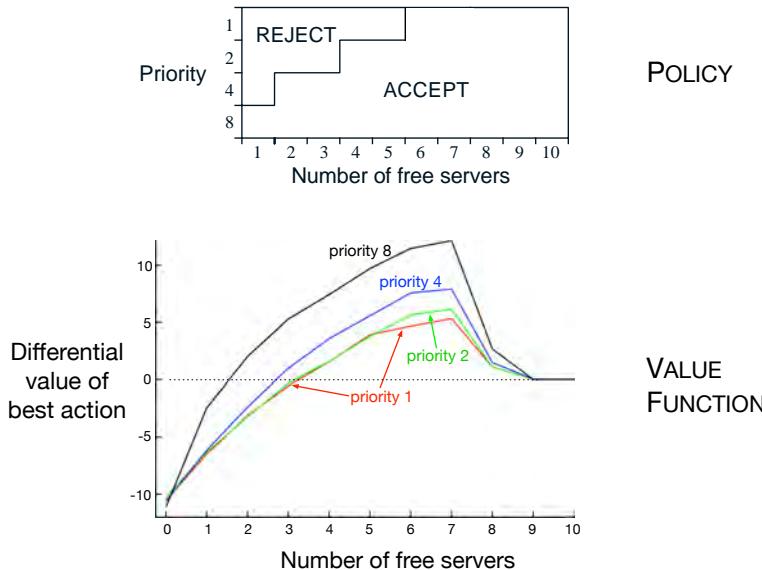


Figure 10.5: The policy and value function found by differential semi-gradient one-step Sarsa on the access-control queuing task after 2 million steps. The drop on the right of the graph is probably due to insufficient data; many of these states were never experienced. The value learned for \bar{R} was about 2.31. (Note that priority 1 here is the lowest priority.) ■

10.4 Deprecating the Discounted Setting

The continuing, discounted problem formulation has been very useful in the tabular case, in which the returns from each state can be separately identified and averaged. But in the approximate case it is questionable whether one should ever use this problem formulation.

To see why, consider an infinite sequence of returns with no beginning or end, and no clearly identified states. The states might be represented only by feature vectors, which may do little to distinguish the states from each other. As a special case, all of the feature vectors may be the same. Thus one really has only the reward sequence (and the actions), and performance has to be assessed purely from these. How could it be done? One way is by averaging the rewards over a long interval—this is the idea of the average-reward setting. How could discounting be used? Well, for each time step we could measure the discounted return. Some returns would be small and some big, so again we would have to average them over a sufficiently large time interval. In the continuing setting there are no starts and ends, and no special time steps, so there is nothing else that could be done. However, if you do this, it turns out that the average of the discounted returns is proportional to the average reward. In fact, for policy π , the average of the discounted returns is always $r(\pi)/(1 - \gamma)$, that is, it is essentially the average reward, $r(\pi)$. In particular, the *ordering* of all policies in the average discounted return setting would be exactly the same as in the average-reward setting. The discount rate γ thus has no effect on the problem formulation. It could in fact be *zero* and the ranking would be unchanged.

This surprising fact is proven in the box on the next page, but the basic idea can be seen via a symmetry argument. Each time step is exactly the same as every other. With discounting, every reward will appear exactly once in each position in some return. The t th reward will appear undiscounted in the $t - 1$ st return, discounted once in the $t - 2$ nd return, and discounted 999 times in the $t - 1000$ th return. The weight on the t th reward is thus $1 + \gamma + \gamma^2 + \gamma^3 + \dots = 1/(1 - \gamma)$. Because all states are the same, they are all weighted by this, and thus the average of the returns will be this times the average reward, or $r(\pi)/(1 - \gamma)$.

This example and the more general argument in the box show that if we optimized discounted value over the on-policy distribution, then the effect would be identical to optimizing *undiscounted* average reward; the actual value of γ would have no effect. This strongly suggests that discounting has no role to play in the definition of the control problem with function approximation. One can nevertheless go ahead and use discounting in solution methods. The discounting parameter γ changes from a problem parameter to a solution method parameter! Unfortunately, discounting algorithms with function approximation do not optimize discounted value over the on-policy distribution, and thus are not guaranteed to optimize average reward.

The root cause of the difficulties with the discounted control setting is that with function approximation we have lost the policy improvement theorem (Section 4.2). It is no longer true that if we change the policy to improve the discounted value of one state then we are guaranteed to have improved the overall policy in any useful sense. That guarantee was key to the theory of our reinforcement learning control methods. With

The Futility of Discounting in Continuing Problems

Perhaps discounting can be saved by choosing an objective that sums discounted values over the distribution with which states occur under the policy:

$$\begin{aligned}
 J(\pi) &= \sum_s \mu_\pi(s) v_\pi^\gamma(s) && \text{(where } v_\pi^\gamma \text{ is the discounted value function)} \\
 &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) [r + \gamma v_\pi^\gamma(s')] && \text{(Bellman Eq.)} \\
 &= r(\pi) + \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \gamma v_\pi^\gamma(s') && \text{(from (10.7))} \\
 &= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s' | s, a) && \text{(from (3.4))} \\
 &= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \mu_\pi(s') && \text{(from (10.8))} \\
 &= r(\pi) + \gamma J(\pi) \\
 &= r(\pi) + \gamma r(\pi) + \gamma^2 J(\pi) \\
 &= r(\pi) + \gamma r(\pi) + \gamma^2 r(\pi) + \gamma^3 r(\pi) + \dots \\
 &= \frac{1}{1 - \gamma} r(\pi).
 \end{aligned}$$

The proposed discounted objective orders policies identically to the undiscounted (average reward) objective. The discount rate γ does not influence the ordering!

function approximation we have lost it!

In fact, the lack of a policy improvement theorem is also a theoretical lacuna for the total-episodic and average-reward settings. Once we introduce function approximation we can no longer guarantee improvement for any setting. In Chapter 13 we introduce an alternative class of reinforcement learning algorithms based on parameterized policies, and there we have a theoretical guarantee called the “policy-gradient theorem” which plays a similar role as the policy improvement theorem. But for methods that learn action values we seem to be currently without a local improvement guarantee (possibly the approach taken by Perkins and Precup (2003) may provide a part of the answer). We do know that ϵ -greedification may sometimes result in an inferior policy, as policies may chatter among good policies rather than converge (Gordon, 1996a). This is an area with multiple open theoretical questions.

10.5 Differential Semi-gradient n -step Sarsa

In order to generalize to n -step bootstrapping, we need an n -step version of the TD error. We begin by generalizing the n -step return (7.4) to its differential form, with function approximation:

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_{t+n-1} + \cdots + R_{t+n} - \bar{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad (10.14)$$

where \bar{R} is an estimate of $r(\pi)$, $n \geq 1$, and $t + n < T$. If $t + n \geq T$, then we define $G_{t:t+n} \doteq G_t$ as usual. The n -step TD error is then

$$\delta_t \doteq G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}), \quad (10.15)$$

after which we can apply our usual semi-gradient Sarsa update (10.12). Pseudocode for the complete algorithm is given in the box.

Differential semi-gradient n -step Sarsa for estimating $\hat{q} \approx q_\pi$ or q_*

Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$, a policy π
 Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
 Initialize average-reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)
 Algorithm parameters: step sizes $\alpha, \beta > 0$, small $\varepsilon > 0$, a positive integer n
 All store and access operations (S_t , A_t , and R_t) can take their index mod $n + 1$

Initialize and store S_0 and A_0
 Loop for each step, $t = 0, 1, 2, \dots$:

- Take action A_t
- Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
- Select and store an action $A_{t+1} \sim \pi(\cdot | S_{t+1})$, or ε -greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
- $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
- If $\tau \geq 0$:
 - $\delta \leftarrow \sum_{i=\tau+1}^{\tau+n} (R_i - \bar{R}) + \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}) - \hat{q}(S_\tau, A_\tau, \mathbf{w})$
 - $\bar{R} \leftarrow \bar{R} + \beta \delta$
 - $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$

Exercise 10.9 In the differential semi-gradient n -step Sarsa algorithm, the step-size parameter on the average reward, β , needs to be quite small so that \bar{R} becomes a good long-term estimate of the average reward. Unfortunately, \bar{R} will then be biased by its initial value for many steps, which may make learning inefficient. Alternatively, one could use a sample average of the observed rewards for \bar{R} . That would initially adapt rapidly but in the long run would also adapt slowly. As the policy slowly changed, \bar{R} would also change; the potential for such long-term nonstationarity makes sample-average methods ill-suited. In fact, the step-size parameter on the average reward is a perfect place to use the unbiased constant-step-size trick from Exercise 2.7. Describe the specific changes needed to the boxed algorithm for differential semi-gradient n -step Sarsa to use this trick. \square

10.6 Summary

In this chapter we have extended the ideas of parameterized function approximation and semi-gradient descent, introduced in the previous chapter, to control. The extension is immediate for the episodic case, but for the continuing case we have to introduce a whole new problem formulation based on maximizing the *average reward setting* per time step. Surprisingly, the discounted formulation cannot be carried over to control in the presence of approximations. In the approximate case most policies cannot be represented by a value function. The arbitrary policies that remain need to be ranked, and the scalar average reward $r(\pi)$ provides an effective way to do this.

The average reward formulation involves new *differential* versions of value functions, Bellman equations, and TD errors, but all of these parallel the old ones, and the conceptual changes are small. There is also a new parallel set of differential algorithms for the average-reward case.

Bibliographical and Historical Remarks

- 10.1** Semi-gradient Sarsa with function approximation was first explored by Rummery and Niranjan (1994). Linear semi-gradient Sarsa with ε -greedy action selection does not converge in the usual sense, but does enter a bounded region near the best solution (Gordon, 1996a, 2001). Precup and Perkins (2003) showed convergence in a differentiable action selection setting. See also Perkins and Pendrith (2002) and Melo, Meyn, and Ribeiro (2008). The mountain-car example is based on a similar task studied by Moore (1990), but the exact form used here is from Sutton (1996).
- 10.2** Episodic n -step semi-gradient Sarsa is based on the forward Sarsa(λ) algorithm of van Seijen (2016). The empirical results shown here are new to the second edition of this text.
- 10.3** The average-reward formulation has been described for dynamic programming (e.g., Puterman, 1994) and from the point of view of reinforcement learning (Mahadevan, 1996; Tadepalli and Ok, 1994; Bertsekas and Tsitsiklis, 1996; Tsitsiklis and Van Roy, 1999). The algorithm described here is the on-policy analog of the “R-learning” algorithm introduced by Schwartz (1993). The name R-learning was probably meant to be the alphabetic successor to Q-learning, but we prefer to think of it as a reference to the learning of differential or *relative* values. The access-control queuing example was suggested by the work of Carlström and Nordström (1997).
- 10.4** The recognition of the limitations of discounting as a formulation of the reinforcement learning problem with function approximation became apparent to the authors shortly after the publication of the first edition of this text. Singh, Jaakkola, and Jordan (1994) may have been the first to observe it in print.

Chapter 11

*Off-policy Methods with Approximation

This book has treated on-policy and off-policy learning methods since Chapter 5 primarily as two alternative ways of handling the conflict between exploitation and exploration inherent in learning forms of generalized policy iteration. The two chapters preceding this have treated the *on*-policy case with function approximation, and in this chapter we treat the *off*-policy case with function approximation. The extension to function approximation turns out to be significantly different and harder for off-policy learning than it is for on-policy learning. The tabular off-policy methods developed in Chapters 6 and 7 readily extend to semi-gradient algorithms, but these algorithms do not converge as robustly as they do under on-policy training. In this chapter we explore the convergence problems, take a closer look at the theory of linear function approximation, introduce a notion of learnability, and then discuss new algorithms with stronger convergence guarantees for the off-policy case. In the end we will have improved methods, but the theoretical results will not be as strong, nor the empirical results as satisfying, as they are for on-policy learning. Along the way, we will gain a deeper understanding of approximation in reinforcement learning for on-policy learning as well as off-policy learning.

Recall that in off-policy learning we seek to learn a value function for a *target policy* π , given data due to a different *behavior policy* b . In the prediction case, both policies are static and given, and we seek to learn either state values $\hat{v} \approx v_\pi$ or action values $\hat{q} \approx q_\pi$. In the control case, action values are learned, and both policies typically change during learning— π being the greedy policy with respect to \hat{q} , and b being something more exploratory such as the ϵ -greedy policy with respect to \hat{q} .

The challenge of off-policy learning can be divided into two parts, one that arises in the tabular case and one that arises only with function approximation. The first part of the challenge has to do with the target of the update (not to be confused with the target policy), and the second part has to do with the distribution of the updates. The techniques related to importance sampling developed in Chapters 5 and 7 deal with the first part; these may increase variance but are needed in all successful algorithms,

tabular and approximate. The extension of these techniques to function approximation are quickly dealt with in the first section of this chapter.

Something more is needed for the second part of the challenge of off-policy learning with function approximation because the distribution of updates in the off-policy case is not according to the on-policy distribution. The on-policy distribution is important to the stability of semi-gradient methods. Two general approaches have been explored to deal with this. One is to use importance sampling methods again, this time to warp the update distribution back to the on-policy distribution, so that semi-gradient methods are guaranteed to converge (in the linear case). The other is to develop true gradient methods that do not rely on any special distribution for stability. We present methods based on both approaches. This is a cutting-edge research area, and it is not clear which of these approaches is most effective in practice.

11.1 Semi-gradient Methods

We begin by describing how the methods developed in earlier chapters for the off-policy case extend readily to function approximation as semi-gradient methods. These methods address the first part of the challenge of off-policy learning (changing the update targets) but not the second part (changing the update distribution). Accordingly, these methods may diverge in some cases, and in that sense are not sound, but still they are often successfully used. Remember that these methods *are* guaranteed stable and asymptotically unbiased for the tabular case, which corresponds to a special case of function approximation. So it may still be possible to combine them with feature selection methods in such a way that the combined system could be assured stable. In any event, these methods are simple and thus a good place to start.

In Chapter 7 we described a variety of tabular off-policy algorithms. To convert them to semi-gradient form, we simply replace the update to an array (V or Q) to an update to a weight vector (\mathbf{w}), using the approximate value function (\hat{v} or \hat{q}) and its gradient. Many of these algorithms use the per-step importance sampling ratio:

$$\rho_t \doteq \rho_{t:t} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}. \quad (11.1)$$

For example, the one-step, state-value algorithm is semi-gradient off-policy TD(0), which is just like the corresponding on-policy algorithm (page 203) except for the addition of ρ_t :

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \rho_t \delta_t \nabla \hat{v}(S_t, \mathbf{w}_t), \quad (11.2)$$

where δ_t is defined appropriately depending on whether the problem is episodic and discounted, or continuing and undiscounted using average reward:

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \text{ or} \quad (11.3)$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t). \quad (11.4)$$

For action values, the one-step algorithm is semi-gradient Expected Sarsa:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \text{ with} \quad (11.5)$$

$$\delta_t \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \text{ or} \quad (\text{episodic})$$

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \sum_a \pi(a|S_{t+1}) \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t). \quad (\text{continuing})$$

Note that this algorithm does not use importance sampling. In the tabular case it is clear that this is appropriate because the only sample action is A_t , and in learning its value we do not have to consider any other actions. With function approximation it is less clear because we might want to weight different state-action pairs differently once they all contribute to the same overall approximation. Proper resolution of this issue awaits a more thorough understanding of the theory of function approximation in reinforcement learning.

In the multi-step generalizations of these algorithms, both the state-value and action-value algorithms involve importance sampling. The n -step version of semi-gradient Sarsa is

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha \rho_{t+1} \cdots \rho_{t+n} [G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})] \nabla \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) \quad (11.6)$$

with

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \text{ or} \quad (\text{episodic})$$

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_t + \cdots + R_{t+n} - \bar{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad (\text{continuing})$$

where here we are being slightly informal in our treatment of the ends of episodes. In the first equation, the ρ_k s for $k \geq T$ (where T is the last time step of the episode) should be taken to be 1, and $G_{t:t+n}$ should be taken to be G_t if $t+n \geq T$.

Recall that we also presented in Chapter 7 an off-policy algorithm that does not involve importance sampling at all: the n -step tree-backup algorithm. Here is its semi-gradient version:

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha [G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})] \nabla \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \quad (11.7)$$

$$G_{t:t+n} \doteq \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) + \sum_{k=t}^{t+n-1} \delta_k \prod_{i=t+1}^k \gamma \pi(A_i|S_i), \quad (11.8)$$

with δ_t as defined at the top of this page for Expected Sarsa. We also defined in Chapter 7 an algorithm that unifies all action-value algorithms: n -step $Q(\sigma)$. We leave the semi-gradient form of that algorithm, and also of the n -step state-value algorithm, as exercises for the reader.

Exercise 11.1 Convert the equation of n -step off-policy TD (7.9) to semi-gradient form. Give accompanying definitions of the return for both the episodic and continuing cases. \square

**Exercise 11.2* Convert the equations of n -step $Q(\sigma)$ (7.11 and 7.17) to semi-gradient form. Give definitions that cover both the episodic and continuing cases. \square

11.2 Examples of Off-policy Divergence

In this section we begin to discuss the second part of the challenge of off-policy learning with function approximation—that the distribution of updates does not match the on-policy distribution. We describe some instructive counterexamples to off-policy learning—cases where semi-gradient and other simple algorithms are unstable and diverge.

To establish intuitions, it is best to consider first a very simple example. Suppose, perhaps as part of a larger MDP, there are two states whose estimated values are of the functional form w and $2w$, where the parameter vector \mathbf{w} consists of only a single component w . This occurs under linear function approximation if the feature vectors for the two states are each simple numbers (single-component vectors), in this case 1 and 2. In the first state, only one action is available, and it results deterministically in a transition to the second state with a reward of 0:



where the expressions inside the two circles indicate the two state's values.

Suppose initially $w = 10$. The transition will then be from a state of estimated value 10 to a state of estimated value 20. It will look like a good transition, and w will be increased to raise the first state's estimated value. If γ is nearly 1, then the TD error will be nearly 10, and, if $\alpha = 0.1$, then w will be increased to nearly 11 in trying to reduce the TD error. However, the second state's estimated value will also be increased, to nearly 22. If the transition occurs again, then it will be from a state of estimated value ≈ 11 to a state of estimated value ≈ 22 , for a TD error of ≈ 11 —larger, not smaller, than before. It will look even more like the first state is undervalued, and its value will be increased again, this time to ≈ 12.1 . This looks bad, and in fact with further updates w will diverge to infinity.

To see this definitively we have to look more carefully at the sequence of updates. The TD error on a transition between the two states is

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t) = 0 + \gamma 2w_t - w_t = (2\gamma - 1)w_t,$$

and the off-policy semi-gradient TD(0) update (from (11.2)) is

$$w_{t+1} = w_t + \alpha \rho_t \delta_t \nabla \hat{v}(S_t, w_t) = w_t + \alpha \cdot 1 \cdot (2\gamma - 1)w_t \cdot 1 = (1 + \alpha(2\gamma - 1))w_t.$$

Note that the importance sampling ratio, ρ_t , is 1 on this transition because there is only one action available from the first state, so its probabilities of being taken under the target and behavior policies must both be 1. In the final update above, the new parameter is the old parameter times a scalar constant, $1 + \alpha(2\gamma - 1)$. If this constant is greater than 1, then the system is unstable and w will go to positive or negative infinity depending on its initial value. Here this constant is greater than 1 whenever $\gamma > 0.5$. Note that stability does not depend on the specific step size, as long as $\alpha > 0$. Smaller or larger step sizes would affect the rate at which w goes to infinity, but not whether it goes there or not.

Key to this example is that the one transition occurs repeatedly without w being updated on other transitions. This is possible under off-policy training because the

behavior policy might select actions on those other transitions which the target policy never would. For these transitions, ρ_t would be zero and no update would be made. Under on-policy training, however, ρ_t is always one. Each time there is a transition from the w state to the $2w$ state, increasing w , there would also have to be a transition out of the $2w$ state. That transition would reduce w , unless it were to a state whose value was higher (because $\gamma < 1$) than $2w$, and then that state would have to be followed by a state of even higher value, or else again w would be reduced. Each state can support the one before only by creating a higher expectation. Eventually the piper must be paid. In the on-policy case the promise of future reward must be kept and the system is kept in check. But in the off-policy case, a promise can be made and then, after taking an action that the target policy never would, forgotten and forgiven.

This simple example communicates much of the reason why off-policy training can lead to divergence, but it is not completely convincing because it is not complete—it is just a fragment of a complete MDP. Can there really be a complete system with instability? A simple complete example of divergence is *Baird's counterexample*. Consider the episodic seven-state, two-action MDP shown in Figure 11.1. The dashed action takes the system to one of the six upper states with equal probability, whereas the solid action takes the system to the seventh state. The behavior policy b selects the dashed and solid actions with probabilities $\frac{6}{7}$ and $\frac{1}{7}$, so that the next-state distribution under it is uniform (the same for all nonterminal states), which is also the starting distribution for each episode. The target policy π always takes the solid action, and so the on-policy distribution (for π) is concentrated in the seventh state. The reward is zero on all transitions. The discount rate is $\gamma = 0.99$.

Consider estimating the state-value under the linear parameterization indicated by the expression shown in each state circle. For example, the estimated value of the leftmost state is $2w_1 + w_8$, where the subscript corresponds to the component of the

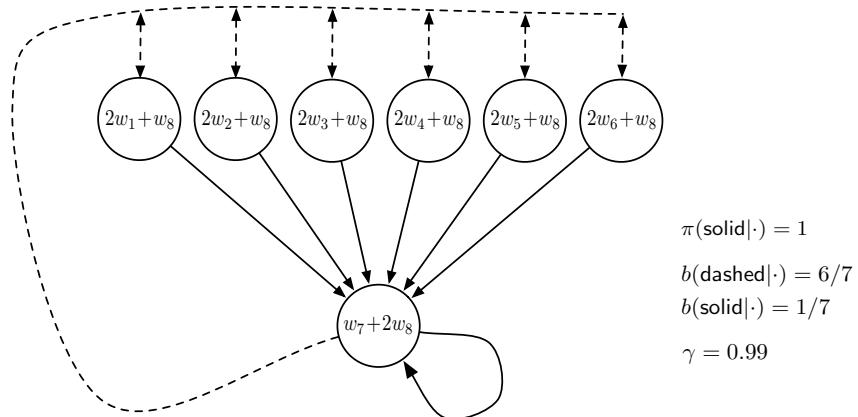


Figure 11.1: Baird's counterexample. The approximate state-value function for this Markov process is of the form shown by the linear expressions inside each state. The **solid** action usually results in the seventh state, and the **dashed** action usually results in one of the other six states, each with equal probability. The reward is always zero.

overall weight vector $\mathbf{w} \in \mathbb{R}^8$; this corresponds to a feature vector for the first state being $\mathbf{x}(1) = (2, 0, 0, 0, 0, 0, 0, 1)^\top$. The reward is zero on all transitions, so the true value function is $v_\pi(s) = 0$, for all s , which can be exactly approximated if $\mathbf{w} = \mathbf{0}$. In fact, there are many solutions, as there are more components to the weight vector (8) than there are nonterminal states (7). Moreover, the set of feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, is a linearly independent set. In all these ways this task seems a favorable case for linear function approximation.

If we apply semi-gradient TD(0) to this problem (11.2), then the weights diverge to infinity, as shown in Figure 11.2 (left). The instability occurs for any positive step size, no matter how small. In fact, it even occurs if an expected update is done as in dynamic programming (DP), as shown in Figure 11.2 (right). That is, if the weight vector, \mathbf{w}_k , is updated for all states at the same time in a semi-gradient way, using the DP (expectation-based) target:

$$\mathbf{w}_{k+1} \doteq \mathbf{w}_k + \frac{\alpha}{|\mathcal{S}|} \sum_s \left(\mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_k) \mid S_t = s] - \hat{v}(s, \mathbf{w}_k) \right) \nabla \hat{v}(s, \mathbf{w}_k). \quad (11.9)$$

In this case, there is no randomness and no asynchrony, just as in a classical DP update. The method is conventional except in its use of semi-gradient function approximation. Yet still the system is unstable.

If we alter just the distribution of DP updates in Baird’s counterexample, from the uniform distribution to the on-policy distribution (which generally requires asynchronous updating), then convergence is guaranteed to a solution with error bounded by (9.14). This example is striking because the TD and DP methods used are arguably the simplest

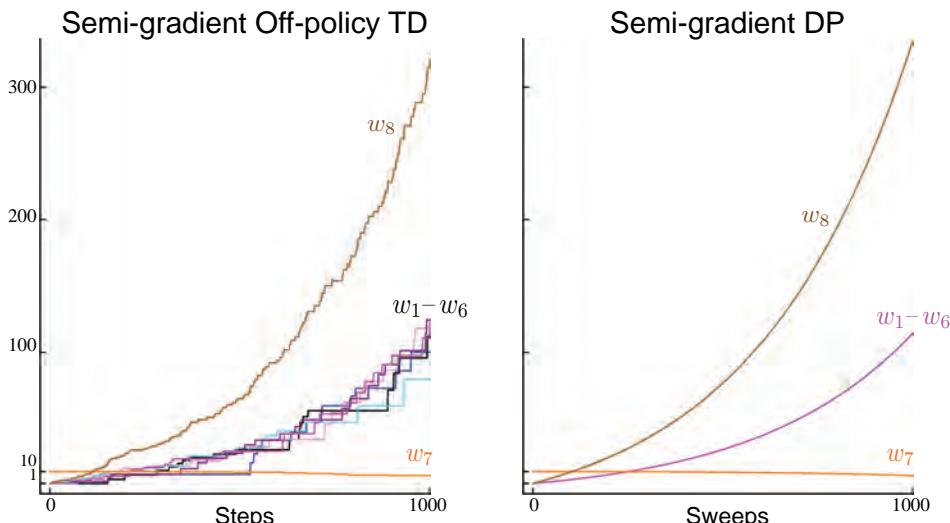


Figure 11.2: Demonstration of instability on Baird’s counterexample. Shown are the evolution of the components of the parameter vector \mathbf{w} of the two semi-gradient algorithms. The step size was $\alpha = 0.01$, and the initial weights were $\mathbf{w} = (1, 1, 1, 1, 1, 1, 10, 1)^\top$.

and best-understood bootstrapping methods, and the linear, semi-descent method used is arguably the simplest and best-understood kind of function approximation. The example shows that even the simplest combination of bootstrapping and function approximation can be unstable if the updates are not done according to the on-policy distribution.

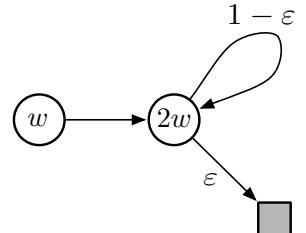
There are also counterexamples similar to Baird's showing divergence for Q-learning. This is cause for concern because otherwise Q-learning has the best convergence guarantees of all control methods. Considerable effort has gone into trying to find a remedy to this problem or to obtain some weaker, but still workable, guarantee. For example, it may be possible to guarantee convergence of Q-learning as long as the behavior policy is sufficiently close to the target policy, for example, when it is the ε -greedy policy. To the best of our knowledge, Q-learning has never been found to diverge in this case, but there has been no theoretical analysis. In the rest of this section we present several other ideas that have been explored.

Suppose that instead of taking just a step toward the expected one-step return on each iteration, as in Baird's counterexample, we actually change the value function all the way to the best, least-squares approximation. Would this solve the instability problem? Of course it would if the feature vectors, $\{\mathbf{x}(s) : s \in \mathcal{S}\}$, formed a linearly independent set, as they do in Baird's counterexample, because then exact approximation is possible on each iteration and the method reduces to standard tabular DP. But of course the point here is to consider the case when an exact solution is *not* possible. In this case stability is not guaranteed even when forming the best approximation at each iteration, as shown in the example.

Example 11.1: Tsitsiklis and Van Roy's Counterexample This example shows that linear function approximation would not work with DP even if the least-squares solution was found at each step. The counterexample is formed by extending the w -to- $2w$ example (from earlier in this section) with a terminal state, as shown to the right. As before, the estimated value of the first state is w , and the estimated value of the second state is $2w$. The reward is zero on all transitions, so the true values are zero at both states, which is exactly representable with $w = 0$. If we set w_{k+1} at each step so as to minimize the \overline{VE} between the estimated value and the expected one-step return, then we have

$$\begin{aligned} w_{k+1} &= \underset{w \in \mathbb{R}}{\operatorname{argmin}} \sum_{s \in \mathcal{S}} \left(\hat{v}(s, w) - \mathbb{E}_{\pi}[R_{t+1} + \gamma \hat{v}(S_{t+1}, w_k) \mid S_t = s] \right)^2 \\ &= \underset{w \in \mathbb{R}}{\operatorname{argmin}} (w - \gamma 2w_k)^2 + (2w - (1 - \varepsilon)\gamma 2w_k)^2 \\ &= \frac{6 - 4\varepsilon}{5} \gamma w_k. \end{aligned} \tag{11.10}$$

The sequence $\{w_k\}$ diverges when $\gamma > \frac{5}{6-4\varepsilon}$ and $w_0 \neq 0$. ■



Another way to try to prevent instability is to use special methods for function approximation. In particular, stability is guaranteed for function approximation methods that do not extrapolate from the observed targets. These methods, called *averagers*, include nearest neighbor methods and locally weighted regression, but not popular methods such as tile coding and artificial neural networks (ANNs).

Exercise 11.3 (programming) Apply one-step semi-gradient Q-learning to Baird’s counterexample and show empirically that its weights diverge. \square

11.3 The Deadly Triad

Our discussion so far can be summarized by saying that the danger of instability and divergence arises whenever we combine all of the following three elements, making up what we call *the deadly triad*:

Function approximation A powerful, scalable way of generalizing from a state space much larger than the memory and computational resources (e.g., linear function approximation or ANNs).

Bootstrapping Update targets that include existing estimates (as in dynamic programming or TD methods) rather than relying exclusively on actual rewards and complete returns (as in MC methods).

Off-policy training Training on a distribution of transitions other than that produced by the target policy. Sweeping through the state space and updating all states uniformly, as in dynamic programming, does not respect the target policy and is an example of off-policy training.

In particular, note that the danger is *not* due to control or to generalized policy iteration. Those cases are more complex to analyze, but the instability arises in the simpler prediction case whenever it includes all three elements of the deadly triad. The danger is also *not* due to learning or to uncertainties about the environment, because it occurs just as strongly in planning methods, such as dynamic programming, in which the environment is completely known.

If any two elements of the deadly triad are present, but not all three, then instability can be avoided. It is natural, then, to go through the three and see if there is any one that can be given up.

Of the three, *function approximation* most clearly cannot be given up. We need methods that scale to large problems and to great expressive power. We need at least linear function approximation with many features and parameters. State aggregation or nonparametric methods whose complexity grows with data are too weak or too expensive. Least-squares methods such as LSTD are of quadratic complexity and are therefore too expensive for large problems.

Doing without *bootstrapping* is possible, at the cost of computational and data efficiency. Perhaps most important are the losses in computational efficiency. Monte Carlo (non-bootstrapping) methods require memory to save everything that happens between making

each prediction and obtaining the final return, and all their computation is done once the final return is obtained. The cost of these computational issues is not apparent on serial von Neumann computers, but would be on specialized hardware. With bootstrapping and eligibility traces (Chapter 12), data can be dealt with when and where it is generated, then need never be used again. The savings in communication and memory made possible by bootstrapping are great.

The losses in data efficiency by giving up *bootstrapping* are also significant. We have seen this repeatedly, such as in Chapters 7 (Figure 7.2) and 9 (Figure 9.2), where some degree of bootstrapping performed much better than Monte Carlo methods on the random-walk prediction task, and in Chapter 10 where the same was seen on the Mountain-Car control task (Figure 10.4). Many other problems show much faster learning with bootstrapping (e.g., see Figure 12.14). Bootstrapping often results in faster learning because it allows learning to take advantage of the state property, the ability to recognize a state upon returning to it. On the other hand, bootstrapping can impair learning on problems where the state representation is poor and causes poor generalization (e.g., this seems to be the case on Tetris, see Şimşek, Algórtá, and Kothiyal, 2016). A poor state representation can also result in bias; this is the reason for the poorer bound on the asymptotic approximation quality of bootstrapping methods (Equation 9.14). On balance, the ability to bootstrap has to be considered extremely valuable. One may sometimes choose not to use it by selecting long n -step updates (or a large bootstrapping parameter, $\lambda \approx 1$; see Chapter 12) but often bootstrapping greatly increases efficiency. It is an ability that we would very much like to keep in our toolkit.

Finally, there is *off-policy learning*; can we give that up? On-policy methods are often adequate. For model-free reinforcement learning, one can simply use Sarsa rather than Q-learning. Off-policy methods free behavior from the target policy. This could be considered an appealing convenience but not a necessity. However, off-policy learning is *essential* to other anticipated use cases, cases that we have not yet mentioned in this book but may be important to the larger goal of creating a powerful intelligent agent.

In these use cases, the agent learns not just a single value function and single policy, but large numbers of them in parallel. There is extensive psychological evidence that people and animals learn to predict many different sensory events, not just rewards. We can be surprised by unusual events, and correct our predictions about them, even if they are of neutral valence (neither good nor bad). This kind of prediction presumably underlies predictive models of the world such as are used in planning. We predict what we will see after eye movements, how long it will take to walk home, the probability of making a jump shot in basketball, and the satisfaction we will get from taking on a new project. In all these cases, the events we would like to predict depend on our acting in a certain way. To learn them all, in parallel, requires learning from the one stream of experience. There are many target policies, and thus the one behavior policy cannot equal all of them. Yet parallel learning is conceptually possible because the behavior policy may overlap in part with many of the target policies. To take full advantage of this requires off-policy learning.

11.4 Linear Value-function Geometry

To better understand the stability challenge of off-policy learning, it is helpful to think about value function approximation more abstractly and independently of how learning is done. We can imagine the space of all possible state-value functions—all functions from states to real numbers $v : \mathcal{S} \rightarrow \mathbb{R}$. Most of these value functions do not correspond to any policy. More important for our purposes is that most are not representable by the function approximator, which by design has far fewer parameters than there are states.

Given an enumeration of the state space $\mathcal{S} = \{s_1, s_2, \dots, s_{|\mathcal{S}|}\}$, any value function v corresponds to a vector listing the value of each state in order $[v(s_1), v(s_2), \dots, v(s_{|\mathcal{S}|})]^\top$. This vector representation of a value function has as many components as there are states. In most cases where we want to use function approximation, this would be far too many components to represent the vector explicitly. Nevertheless, the idea of this vector is conceptually useful. In the following, we treat a value function and its vector representation interchangeably.

To develop intuitions, consider the case with three states $\mathcal{S} = \{s_1, s_2, s_3\}$ and two parameters $\mathbf{w} = (w_1, w_2)^\top$. We can then view all value functions/vectors as points in a three-dimensional space. The parameters provide an alternative coordinate system over a two-dimensional subspace. Any weight vector $\mathbf{w} = (w_1, w_2)^\top$ is a point in the two-dimensional subspace and thus also a complete value function $v_{\mathbf{w}}$ that assigns values to all three states. With general function approximation the relationship between the full space and the subspace of representable functions could be complex, but in the case of *linear* value-function approximation the subspace is a simple plane, as suggested by Figure 11.3.

Now consider a single fixed policy π . We assume that its true value function, v_π , is too complex to be represented exactly as an approximation. Thus v_π is not in the subspace; in the figure it is depicted as being above the planar subspace of representable functions.

If v_π cannot be represented exactly, what representable value function is closest to it? This turns out to be a subtle question with multiple answers. To begin, we need a measure of the distance between two value functions. Given two value functions v_1 and v_2 , we can talk about the vector difference between them, $v = v_1 - v_2$. If v is small, then the two value functions are close to each other. But how are we to measure the size of this difference vector? The conventional Euclidean norm is not appropriate because, as discussed in Section 9.2, some states are more important than others because they occur more frequently or because we are more interested in them (Section 9.11). As in Section 9.2, let us use the distribution $\mu : \mathcal{S} \rightarrow [0, 1]$ to specify the degree to which we care about different states being accurately valued (often taken to be the on-policy distribution). We can then define the distance between value functions using the norm

$$\|v\|_\mu^2 \doteq \sum_{s \in \mathcal{S}} \mu(s)v(s)^2. \quad (11.11)$$

Note that the $\overline{\text{VE}}$ from Section 9.2 can be written simply using this norm as $\overline{\text{VE}}(\mathbf{w}) = \|v_{\mathbf{w}} - v_\pi\|_\mu^2$. For any value function v , the operation of finding its closest value function in the subspace of representable value functions is a projection operation. We define a

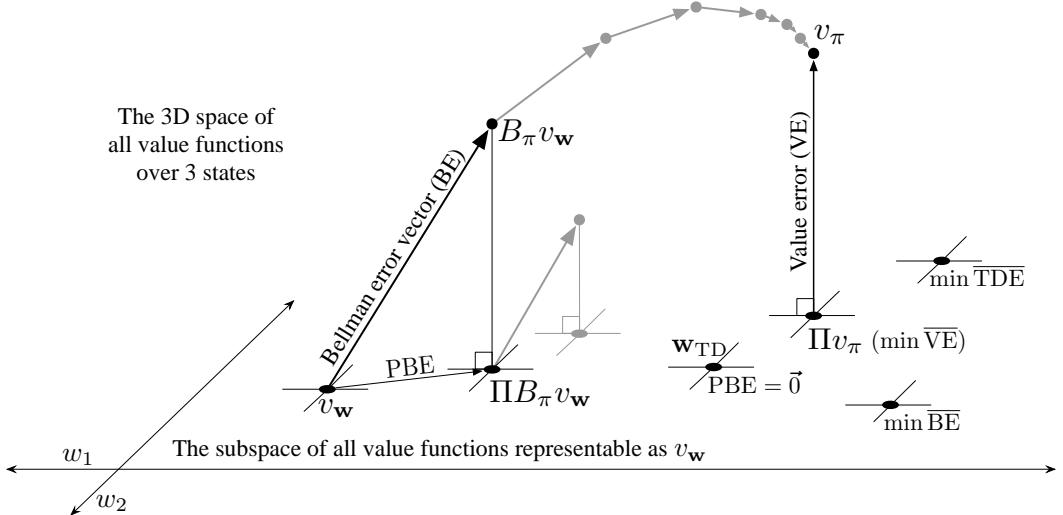


Figure 11.3: The geometry of linear value-function approximation. Shown is the three-dimensional space of all value functions over three states, while shown as a plane is the subspace of all value functions representable by a linear function approximator with parameter $\mathbf{w} = (w_1, w_2)$. The true value function v_π is in the larger space and can be projected down (into the subspace, using a projection operator Π) to its best approximation in the value error (VE) sense. The best approximators in the Bellman error (BE), projected Bellman error (PBE), and temporal difference error (TDE) senses are all potentially different and are shown in the lower right. (VE, BE, and PBE are all treated as the corresponding vectors in this figure.) The Bellman operator takes a value function in the plane to one outside, which can then be projected back. If you iteratively applied the Bellman operator outside the space (shown in gray above) you would reach the true value function, as in conventional dynamic programming. If instead you kept projecting back into the subspace at each step, as in the lower step shown in gray, then the fixed point would be the point of vector-zero PBE.

projection operator Π that takes an arbitrary value function to the representable function that is closest in our norm:

$$\Pi v \doteq v_w \text{ where } \mathbf{w} = \arg \min_{\mathbf{w} \in \mathbb{R}^d} \|v - v_w\|_\mu^2. \quad (11.12)$$

The representable value function that is closest to the true value function v_π is thus its projection, Πv_π , as suggested in Figure 11.3. This is the solution asymptotically found by Monte Carlo methods, albeit often very slowly. The projection operation is discussed more fully in the box on the next page.

TD methods find different solutions. To understand their rationale, recall that the Bellman equation for value function v_π is

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')], \quad \text{for all } s \in \mathcal{S}. \quad (11.16)$$

The projection matrix

For a linear function approximator, the projection operation is linear, which implies that it can be represented as an $|\mathcal{S}| \times |\mathcal{S}|$ matrix:

$$\Pi \doteq \mathbf{X} (\mathbf{X}^\top \mathbf{D} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{D}, \quad (11.13)$$

where, as in Section 9.4, \mathbf{D} denotes the $|\mathcal{S}| \times |\mathcal{S}|$ diagonal matrix with the $\mu(s)$ on the diagonal, and \mathbf{X} denotes the $|\mathcal{S}| \times d$ matrix whose rows are the feature vectors $\mathbf{x}(s)^\top$, one for each state s . If the inverse in (11.13) does not exist, then the pseudoinverse is substituted. Using these matrices, the squared norm of a vector can be written

$$\|v\|_\mu^2 = v^\top \mathbf{D} v, \quad (11.14)$$

and the approximate linear value function can be written

$$v_{\mathbf{w}} = \mathbf{X} \mathbf{w}. \quad (11.15)$$

The true value function v_π is the only value function that solves (11.16) exactly. If an approximate value function $v_{\mathbf{w}}$ were substituted for v_π , the difference between the right and left sides of the modified equation could be used as a measure of how far off $v_{\mathbf{w}}$ is from v_π . We call this the *Bellman error* at state s :

$$\bar{\delta}_{\mathbf{w}}(s) \doteq \left(\sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_{\mathbf{w}}(s')] \right) - v_{\mathbf{w}}(s) \quad (11.17)$$

$$= \mathbb{E}_\pi [R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t) \mid S_t = s, A_t \sim \pi], \quad (11.18)$$

which shows clearly the relationship of the Bellman error to the TD error (11.3). The Bellman error is the expectation of the TD error.

The vector of all the Bellman errors, at all states, $\bar{\delta}_{\mathbf{w}} \in \mathbb{R}^{|\mathcal{S}|}$, is called the *Bellman error vector* (shown as BE in Figure 11.3). The overall size of this vector, in the norm, is an overall measure of the error in the value function, called the *mean square Bellman error*:

$$\overline{\text{BE}}(\mathbf{w}) = \|\bar{\delta}_{\mathbf{w}}\|_\mu^2. \quad (11.19)$$

It is not possible in general to reduce the $\overline{\text{BE}}$ to zero (at which point $v_{\mathbf{w}} = v_\pi$), but for linear function approximation there is a unique value of \mathbf{w} for which the $\overline{\text{BE}}$ is minimized. This point in the representable-function subspace (labeled $\min \overline{\text{BE}}$ in Figure 11.3) is different in general from that which minimizes the $\overline{\text{VE}}$ (shown as Πv_π). Methods that seek to minimize the $\overline{\text{BE}}$ are discussed in the next two sections.

The Bellman error vector is shown in Figure 11.3 as the result of applying the *Bellman operator* $B_\pi : \mathbb{R}^{|\mathcal{S}|} \rightarrow \mathbb{R}^{|\mathcal{S}|}$ to the approximate value function. The Bellman operator is

defined by

$$(B_\pi v)(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v(s')], \quad (11.20)$$

for all $s \in \mathcal{S}$, $v : \mathcal{S} \rightarrow \mathbb{R}$. The Bellman error vector for v_w can be written $\bar{\delta}_w = B_\pi v_w - v_w$.

If the Bellman operator is applied to a value function in the representable subspace, then, in general, it will produce a new value function that is outside the subspace, as suggested in the figure. In dynamic programming (without function approximation), this operator is applied repeatedly to the points outside the representable space, as suggested by the gray arrows in the top of Figure 11.3. Eventually that process converges to the true value function v_π , the only fixed point for the Bellman operator, the only value function for which

$$v_\pi = B_\pi v_\pi, \quad (11.21)$$

which is just another way of writing the Bellman equation for π (11.16).

With function approximation, however, the intermediate value functions lying outside the subspace cannot be represented. The gray arrows in the upper part of Figure 11.3 cannot be followed because after the first update (dark line) the value function must be projected back into something representable. The next iteration then begins within the subspace; the value function is again taken outside of the subspace by the Bellman operator and then mapped back by the projection operator, as suggested by the lower gray arrow and line. Following these arrows is a DP-like process with approximation.

In this case we are interested in the projection of the Bellman error vector back into the representable space. This is the projected Bellman error vector $\Pi \bar{\delta}_w$, shown in Figure 11.3 as PBE. The size of this vector, in the norm, is another measure of error in the approximate value function. For any approximate value function v_w , we define the *mean square Projected Bellman error*, denoted $\overline{\text{PBE}}$, as

$$\overline{\text{PBE}}(w) = \|\Pi \bar{\delta}_w\|_\mu^2. \quad (11.22)$$

With linear function approximation there always exists an approximate value function (within the subspace) with zero $\overline{\text{PBE}}$; this is the TD fixed point, w_{TD} , introduced in Section 9.4. As we have seen, this point is not always stable under semi-gradient TD methods and off-policy training. As shown in the figure, this value function is generally different from those minimizing $\overline{\text{VE}}$ or $\overline{\text{BE}}$. Methods that are guaranteed to converge to it are discussed in Sections 11.7 and 11.8.

11.5 Gradient Descent in the Bellman Error

Armed with a better understanding of value function approximation and its various objectives, we return now to the challenge of stability in off-policy learning. We would like to apply the approach of stochastic gradient descent (SGD, Section 9.3), in which updates are made that in expectation are equal to the negative gradient of an objective

function. These methods always go downhill (in expectation) in the objective and because of this are typically stable with excellent convergence properties. Among the algorithms investigated so far in this book, only the Monte Carlo methods are true SGD methods. These methods converge robustly under both on-policy and off-policy training as well as for general nonlinear (differentiable) function approximators, though they are often slower than semi-gradient methods with bootstrapping, which are not SGD methods. Semi-gradient methods may diverge under off-policy training, as we have seen earlier in this chapter, and under contrived cases of nonlinear function approximation (Tsitsiklis and Van Roy, 1997). With a true SGD method such divergence would not be possible.

The appeal of SGD is so strong that great effort has gone into finding a practical way of harnessing it for reinforcement learning. The starting place of all such efforts is the choice of an error or objective function to optimize. In this and the next section we explore the origins and limits of the most popular proposed objective function, that based on the *Bellman error* introduced in the previous section. Although this has been a popular and influential approach, the conclusion that we reach here is that it is a misstep and yields no good learning algorithms. On the other hand, this approach fails in an interesting way that offers insight into what might constitute a good approach.

To begin, let us consider not the Bellman error, but something more immediate and naive. Temporal difference learning is driven by the TD error. Why not take the minimization of the expected square of the TD error as the objective? In the general function-approximation case, the one-step TD error with discounting is

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t).$$

A possible objective function then is what one might call the *mean square TD error*:

$$\begin{aligned}\overline{\text{TDE}}(\mathbf{w}) &= \sum_{s \in \mathcal{S}} \mu(s) \mathbb{E}[\delta_t^2 \mid S_t = s, A_t \sim \pi] \\ &= \sum_{s \in \mathcal{S}} \mu(s) \mathbb{E}[\rho_t \delta_t^2 \mid S_t = s, A_t \sim b] \\ &= \mathbb{E}_b[\rho_t \delta_t^2]. \quad (\text{if } \mu \text{ is the distribution encountered under } b)\end{aligned}$$

The last equation is of the form needed for SGD; it gives the objective as an expectation that can be sampled from experience (remember the experience is due to the behavior policy b). Thus, following the standard SGD approach, one can derive the per-step update based on a sample of this expected value:

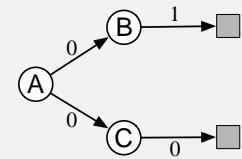
$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2} \alpha \nabla(\rho_t \delta_t^2) \\ &= \mathbf{w}_t - \alpha \rho_t \delta_t \nabla \delta_t \\ &= \mathbf{w}_t + \alpha \rho_t \delta_t (\nabla \hat{v}(S_t, \mathbf{w}_t) - \gamma \nabla \hat{v}(S_{t+1}, \mathbf{w}_t)),\end{aligned}\tag{11.23}$$

which you will recognize as the same as the semi-gradient TD algorithm (11.2) except for the additional final term. This term completes the gradient and makes this a true SGD algorithm with excellent convergence guarantees. Let us call this algorithm the *naive*

residual-gradient algorithm (after Baird, 1995). Although the naive residual-gradient algorithm converges robustly, it does not necessarily converge to a desirable place.

**Example 11.2: A-split example,
showing the naiveté of the naive residual-gradient algorithm**

Consider the three-state episodic MRP shown to the right. Episodes begin in state A and then ‘split’ stochastically, half the time going to B (and then invariably going on to terminate with a reward of 1) and half the time going to state C (and then invariably terminating with a reward of zero). Reward for the first transition, out of A, is always zero whichever way the episode goes. As this is an episodic problem, we can take γ to be 1. We also assume on-policy training, so that ρ_t is always 1, and tabular function approximation, so that the learning algorithms are free to give arbitrary, independent values to all three states. Thus, this should be an easy problem.



What should the values be? From A, half the time the return is 1, and half the time the return is 0; A should have value $\frac{1}{2}$. From B the return is always 1, so its value should be 1, and similarly from C the return is always 0, so its value should be 0. These are the true values and, as this is a tabular problem, all the methods presented previously converge to them exactly.

However, the naive residual-gradient algorithm finds different values for B and C. It converges with B having a value of $\frac{3}{4}$ and C having a value of $\frac{1}{4}$ (A converges correctly to $\frac{1}{2}$). These are in fact the values that minimize the $\overline{\text{TDE}}$.

Let us compute the $\overline{\text{TDE}}$ for these values. The first transition of each episode is either up from A’s $\frac{1}{2}$ to B’s $\frac{3}{4}$, a change of $\frac{1}{4}$, or down from A’s $\frac{1}{2}$ to C’s $\frac{1}{4}$, a change of $-\frac{1}{4}$. Because the reward is zero on these transitions, and $\gamma = 1$, these changes are the TD errors, and thus the squared TD error is always $\frac{1}{16}$ on the first transition. The second transition is similar; it is either up from B’s $\frac{3}{4}$ to a reward of 1 (and a terminal state of value 0), or down from C’s $\frac{1}{4}$ to a reward of 0 (again with a terminal state of value 0). Thus, the TD error is always $\pm\frac{1}{4}$, for a squared error of $\frac{1}{16}$ on the second step. Thus, for this set of values, the $\overline{\text{TDE}}$ on both steps is $\frac{1}{16}$.

Now let’s compute the $\overline{\text{TDE}}$ for the true values (B at 1, C at 0, and A at $\frac{1}{2}$). In this case the first transition is either from $\frac{1}{2}$ up to 1, at B, or from $\frac{1}{2}$ down to 0, at C; in either case the absolute error is $\frac{1}{2}$ and the squared error is $\frac{1}{4}$. The second transition has zero error because the starting value, either 1 or 0 depending on whether the transition is from B or C, always exactly matches the immediate reward and return. Thus the squared TD error is $\frac{1}{4}$ on the first transition and 0 on the second, for a mean reward over the two transitions of $\frac{1}{8}$. As $\frac{1}{8}$ is bigger than $\frac{1}{16}$, this solution is worse according to the $\overline{\text{TDE}}$. On this simple problem, the true values do not have the smallest $\overline{\text{TDE}}$.

A tabular representation is used in the A-split example, so the true state values can be exactly represented, yet the naive residual-gradient algorithm finds different values, and these values have lower $\overline{\text{TDE}}$ than do the true values. Minimizing the $\overline{\text{TDE}}$ is naive; by penalizing all TD errors it achieves something more like temporal smoothing than accurate prediction.

A better idea would seem to be minimizing the mean square Bellman error ($\overline{\text{BE}}$). If the exact values are learned, the Bellman error is zero everywhere. Thus, a Bellman-error-minimizing algorithm should have no trouble with the A-split example. We cannot expect to achieve zero Bellman error in general, as it would involve finding the true value function, which we presume is outside the space of representable value functions. But getting close to this ideal is a natural-seeming goal. As we have seen, the Bellman error is also closely related to the TD error. The Bellman error for a state is the expected TD error in that state. So let's repeat the derivation above with the expected TD error (all expectations here are implicitly conditional on S_t):

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla(\mathbb{E}_\pi[\delta_t]^2) \\ &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla(\mathbb{E}_b[\rho_t\delta_t]^2) \\ &= \mathbf{w}_t - \alpha\mathbb{E}_b[\rho_t\delta_t]\nabla\mathbb{E}_b[\rho_t\delta_t] \\ &= \mathbf{w}_t - \alpha\mathbb{E}_b[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))]\mathbb{E}_b[\rho_t\nabla\delta_t] \\ &= \mathbf{w}_t + \alpha\left[\mathbb{E}_b[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}))] - \hat{v}(S_t, \mathbf{w})\right]\left[\nabla\hat{v}(S_t, \mathbf{w}) - \gamma\mathbb{E}_b[\rho_t\nabla\hat{v}(S_{t+1}, \mathbf{w})]\right].\end{aligned}$$

This update and various ways of sampling it are referred to as the *residual-gradient algorithm*. If you simply used the sample values in all the expectations, then the equation above reduces almost exactly to (11.23), the naive residual-gradient algorithm.¹ But this is naive, because the equation above involves the next state, S_{t+1} , appearing in two expectations that are multiplied together. To get an unbiased sample of the product, two independent samples of the next state are required, but during normal interaction with an external environment only one is obtained. One expectation or the other can be sampled, but not both.

There are two ways to make the residual-gradient algorithm work. One is in the case of deterministic environments. If the transition to the next state is deterministic, then the two samples will necessarily be the same, and the naive algorithm is valid. The other way is to obtain *two* independent samples of the next state, S_{t+1} , from S_t , one for the first expectation and another for the second expectation. In real interaction with an environment, this would not seem possible, but when interacting with a simulated environment, it is. One simply rolls back to the previous state and obtains an alternate next state before proceeding forward from the first next state. In either of these cases the residual-gradient algorithm is guaranteed to converge to a minimum of the $\overline{\text{BE}}$ under the usual conditions on the step-size parameter. As a true SGD method, this convergence is

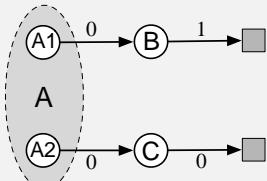
¹For state values there remains a small difference in the treatment of the importance sampling ratio ρ_t . In the analogous action-value case (which is the most important case for control algorithms), the residual-gradient algorithm would reduce exactly to the naive version.

robust, applying to both linear and nonlinear function approximators. In the linear case, convergence is always to the *unique* \mathbf{w} that minimizes the $\overline{\text{BE}}$.

However, there remain at least three ways in which the convergence of the residual-gradient method is unsatisfactory. The first of these is that empirically it is slow, much slower than semi-gradient methods. Indeed, proponents of this method have proposed increasing its speed by combining it with faster semi-gradient methods initially, then gradually switching over to residual gradient for the convergence guarantee (Baird and Moore, 1999). The second way in which the residual-gradient algorithm is unsatisfactory is that it still seems to converge to the wrong values. It does get the right values in all tabular cases, such as the A-split example, as for those an exact solution to the Bellman

Example 11.3: A-presplit example, a counterexample for the $\overline{\text{BE}}$

Consider the three-state episodic MRP shown to the right: Episodes start in either A_1 or A_2 , with equal probability. These two states look exactly the same to the function approximator, like a single state A whose feature representation is distinct from and unrelated to the feature representation of the other two states, B and C , which are also distinct from each other. Specifically, the parameter of the function approximator has three components, one giving the value of state B , one giving the value of state C , and one giving the value of both states A_1 and A_2 . Other than the selection of the initial state, the system is deterministic. If it starts in A_1 , then it transitions to B with a reward of 0 and then on to termination with a reward of 1. If it starts in A_2 , then it transitions to C , and then to termination, with both rewards zero.



To a learning algorithm, seeing only the features, the system looks identical to the A-split example. The system seems to always start in A , followed by either B or C with equal probability, and then terminating with a 1 or a 0 depending deterministically on the previous state. As in the A-split example, the true values of B and C are 1 and 0, and the best shared value of A_1 and A_2 is $\frac{1}{2}$, by symmetry.

Because this problem appears externally identical to the A-split example, we already know what values will be found by the algorithms. Semi-gradient TD converges to the ideal values just mentioned, while the naive residual-gradient algorithm converges to values of $\frac{3}{4}$ and $\frac{1}{4}$ for B and C respectively. All state transitions are deterministic, so the non-naive residual-gradient algorithm will also converge to these values (it is the same algorithm in this case). It follows then that this ‘naive’ solution must also be the one that minimizes the $\overline{\text{BE}}$, and so it is. On a deterministic problem, the Bellman errors and TD errors are all the same, so the $\overline{\text{BE}}$ is always the same as the $\overline{\text{TDE}}$. Optimizing the $\overline{\text{BE}}$ on this example gives rise to the same failure mode as with the naive residual-gradient algorithm on the A-split example.

equation is possible. But if we examine examples with genuine function approximation, then the residual-gradient algorithm, and indeed the \overline{BE} objective, seem to find the wrong value functions. One of the most telling such examples is the variation on the A-split example known as the *A-presplit* example, shown on the preceding page, in which the residual-gradient algorithm finds the same poor solution as its naive version. This example shows intuitively that minimizing the \overline{BE} (which the residual-gradient algorithm surely does) may not be a desirable goal.

The third way in which the convergence of the residual-gradient algorithm is not satisfactory is explained in the next section. Like the second way, the third way is also a problem with the \overline{BE} objective itself rather than with any particular algorithm for achieving it.

11.6 The Bellman Error is Not Learnable

The concept of learnability that we introduce in this section is different from that commonly used in machine learning. There, a hypothesis is said to be “learnable” if it is *efficiently* learnable, meaning that it can be learned within a polynomial rather than an exponential number of examples. Here we use the term in a more basic way, to mean learnable at all, with any amount of experience. It turns out many quantities of apparent interest in reinforcement learning cannot be learned even from an infinite amount of experiential data. These quantities are well defined and can be computed given knowledge of the internal structure of the environment, but cannot be computed or estimated from the observed sequence of feature vectors, actions, and rewards.² We say that they are not *learnable*. It will turn out that the Bellman error objective (\overline{BE}) introduced in the last two sections is not learnable in this sense. That the Bellman error objective cannot be learned from the observable data is probably the strongest reason not to seek it.

To make the concept of learnability clear, let’s start with some simple examples. Consider the two Markov reward processes³ (MRPs) diagrammed below:



Where two edges leave a state, both transitions are assumed to occur with equal probability, and the numbers indicate the reward received. All the states appear the same; they all produce the same single-component feature vector $x = 1$ and have approximated value w . Thus, the only varying part of the data trajectory is the reward sequence. The left MRP stays in the same state and emits an endless stream of 0s and 2s at random, each with 0.5 probability. The right MRP, on every step, either stays in its current state or

²They would of course be estimated if the *state* sequence were observed rather than only the corresponding feature vectors.

³All MRPs can be considered MDPs with a single action in all states; what we conclude about MRPs here applies as well to MDPs.

switches to the other, with equal probability. The reward is deterministic in this MRP, always a 0 from one state and always a 2 from the other, but because the each state is equally likely on each step, the observable data is again an endless stream of 0s and 2s at random, identical to that produced by the left MRP. (We can assume the right MRP starts in one of two states at random with equal probability.) Thus, even given an infinite amount of data, it would not be possible to tell which of these two MRPs was generating it. In particular, we could not tell if the MRP has one state or two, is stochastic or deterministic. These things are not learnable.

This pair of MRPs also illustrates that the \overline{VE} objective (9.1) is not learnable. If $\gamma = 0$, then the true values of the three states (in both MRPs), left to right, are 1, 0, and 2. Suppose $w = 1$. Then the \overline{VE} is 0 for the left MRP and 1 for the right MRP. Because the \overline{VE} is different in the two problems, yet the data generated has the same distribution, the \overline{VE} cannot be learned. The \overline{VE} is not a unique function of the data distribution. And if it cannot be learned, then how could the \overline{VE} possibly be useful as an objective for learning?

If an objective cannot be learned, it does indeed draw its utility into question. In the case of the \overline{VE} , however, there is a way out. Note that the same solution, $w = 1$, is optimal for both MRPs above (assuming μ is the same for the two indistinguishable states in the right MRP). Is this a coincidence, or could it be generally true that all MDPs with the same data distribution also have the same optimal parameter vector? If this is true—and we will show next that it is—then the \overline{VE} remains a usable objective. The \overline{VE} is not learnable, but the parameter that optimizes it is!

To understand this, it is useful to bring in another natural objective function, this time one that is clearly learnable. One error that is always observable is that between the value estimate at each time and the return from that time. The *mean square return error*, denoted \overline{RE} , is the expectation, under μ , of the square of this error. In the on-policy case the \overline{RE} can be written

$$\begin{aligned}\overline{RE}(w) &= \mathbb{E} \left[(G_t - \hat{v}(S_t, w))^2 \right] \\ &= \overline{VE}(w) + \mathbb{E} \left[(G_t - v_\pi(S_t))^2 \right].\end{aligned}\tag{11.24}$$

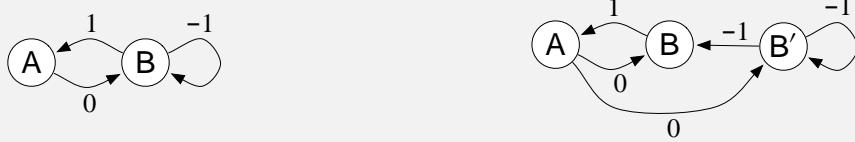
Thus, the two objectives are the same except for a variance term that does not depend on the parameter vector. The two objectives must therefore have the same optimal parameter value w^* . The overall relationships are summarized in the left side of Figure 11.4.

**Exercise 11.4* Prove (11.24). Hint: Write the \overline{RE} as an expectation over possible states s of the expectation of the squared error given that $S_t = s$. Then add and subtract the true value of state s from the error (before squaring), grouping the subtracted true value with the return and the added true value with the estimated value. Then, if you expand the square, the most complex term will end up being zero, leaving you with (11.24). \square

Now let us return to the \overline{BE} . The \overline{BE} is like the \overline{VE} in that it can be computed from knowledge of the MDP but is not learnable from data. But it is not like the \overline{VE} in that its minimum solution is not learnable. The box on the next page presents a counterexample—two MRPs that generate the same data distribution but whose minimizing parameter vector is different, proving that the optimal parameter vector is not a function of the

Example 11.4: Counterexample to the learnability of the Bellman error

To show the full range of possibilities we need a slightly more complex pair of Markov reward processes (MRPs) than those considered earlier. Consider the following two MRPs:



Where two edges leave a state, both transitions are assumed to occur with equal probability, and the numbers indicate the reward received. The MRP on the left has two states that are represented distinctly. The MRP on the right has three states, two of which, B and B', appear the same and must be given the same approximate value. Specifically, \mathbf{w} has two components and the value of state A is given by the first component and the value of B and B' is given by the second. The second MRP has been designed so that equal time is spent in all three states, so we can take $\mu(s) = \frac{1}{3}$, for all s .

Note that the observable data distribution is identical for the two MRPs. In both cases the agent will see single occurrences of A followed by a 0, then some number of apparent Bs, each followed by a -1 except the last, which is followed by a 1, then we start all over again with a single A and a 0, etc. All the statistical details are the same as well; in both MRPs, the probability of a string of k Bs is 2^{-k} .

Now suppose $\mathbf{w} = \mathbf{0}$. In the first MRP, this is an exact solution, and the $\overline{\text{BE}}$ is zero. In the second MRP, this solution produces a squared error in both B and B' of 1, such that $\overline{\text{BE}} = \mu(\mathbf{B})1 + \mu(\mathbf{B}')1 = \frac{2}{3}$. These two MRPs, which generate the same data distribution, have different $\overline{\text{BE}}$ s; the $\overline{\text{BE}}$ is not learnable.

Moreover (and unlike the earlier example for the $\overline{\text{VE}}$) the minimizing value of \mathbf{w} is different for the two MRPs. For the first MRP, $\mathbf{w} = \mathbf{0}$ minimizes the $\overline{\text{BE}}$ for any γ . For the second MRP, the minimizing \mathbf{w} is a complicated function of γ , but in the limit, as $\gamma \rightarrow 1$, it is $(-\frac{1}{2}, 0)^\top$. Thus the solution that minimizes $\overline{\text{BE}}$ cannot be estimated from data alone; knowledge of the MRP beyond what is revealed in the data is required. In this sense, it is impossible in principle to pursue the $\overline{\text{BE}}$ as an objective for learning.

It may be surprising that in the second MRP the $\overline{\text{BE}}$ -minimizing value of A is so far from zero. Recall that A has a dedicated weight and thus its value is unconstrained by function approximation. A is followed by a reward of 0 and transition to a state with a value of nearly 0, which suggests $v_{\mathbf{w}}(\mathbf{A})$ should be 0; why is its optimal value substantially negative rather than 0? The answer is that making $v_{\mathbf{w}}(\mathbf{A})$ negative reduces the error upon arriving in A from B. The reward on this deterministic transition is 1, which implies that B should have a value 1 more than A. Because B's value is approximately zero, A's value is driven toward -1. The $\overline{\text{BE}}$ -minimizing value of $\approx -\frac{1}{2}$ for A is a compromise between reducing the errors on leaving and on entering A.

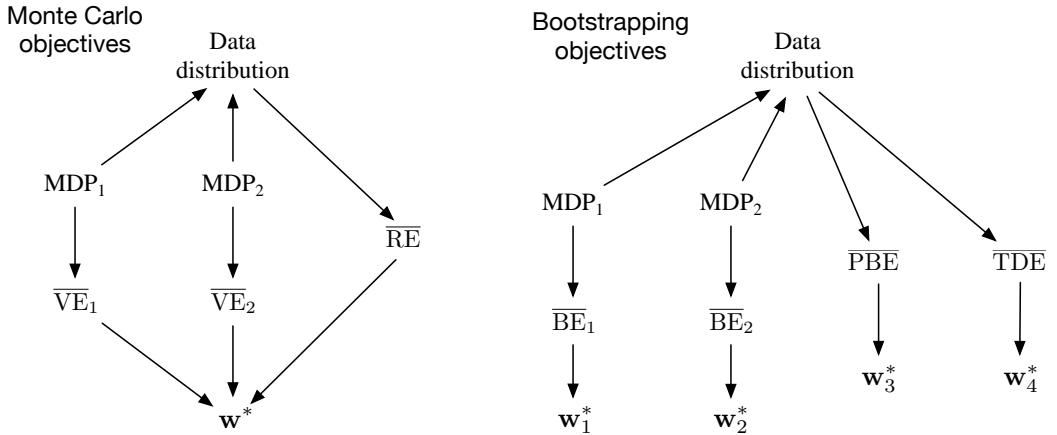


Figure 11.4: Causal relationships among the data distribution, MDPs, and various objectives. **Left, Monte Carlo objectives:** Two different MDPs can produce the same data distribution yet also produce different \overline{VE} s, proving that the \overline{VE} objective cannot be determined from data and is not learnable. However, all such \overline{VE} s must have the same optimal parameter vector, w^* ! Moreover, this same w^* can be determined from another objective, the \overline{RE} , which *is* uniquely determined from the data distribution. Thus w^* and the \overline{RE} are learnable even though the \overline{VE} s are not. **Right, Bootstrapping objectives:** Two different MDPs can produce the same data distribution yet also produce different \overline{BE} s *and* have different minimizing parameter vectors; these are not learnable from the data distribution. The \overline{PBE} and \overline{TDE} objectives and their (different) minima can be directly determined from data and thus are learnable.

data and thus cannot be learned from it. The other bootstrapping objectives that we have considered, the \overline{PBE} and \overline{TDE} , can be determined from data (are learnable) and determine optimal solutions that are in general different from each other and the \overline{BE} minimums. The general case is summarized in the right side of Figure 11.4.

Thus, the \overline{BE} is not learnable; it cannot be estimated from feature vectors and other observable data. This limits the \overline{BE} to model-based settings. There can be no algorithm that minimizes the \overline{BE} without access to the underlying MDP states beyond the feature vectors. The residual-gradient algorithm is only able to minimize \overline{BE} because it is allowed to double sample from the same state—not a state that has the same feature vector, but one that is guaranteed to be the same underlying state. We can see now that there is no way around this. Minimizing the \overline{BE} requires some such access to the nominal, underlying MDP. This is an important limitation of the \overline{BE} beyond that identified in the A-presplit example on page 273. All this directs more attention toward the \overline{PBE} .

11.7 Gradient-TD Methods

We now consider SGD methods for minimizing the $\overline{\text{PBE}}$. As true SGD methods, these *Gradient-TD methods* have robust convergence properties even under off-policy training and nonlinear function approximation. Remember that in the linear case there is always an exact solution, the TD fixed point \mathbf{w}_{TD} , at which the $\overline{\text{PBE}}$ is zero. This solution could be found by least-squares methods (Section 9.8), but only by methods of quadratic $O(d^2)$ complexity in the number of parameters. We seek instead an SGD method, which should be $O(d)$ and have robust convergence properties. Gradient-TD methods come close to achieving these goals, at the cost of a rough doubling of computational complexity.

To derive an SGD method for the $\overline{\text{PBE}}$ (assuming linear function approximation) we begin by expanding and rewriting the objective (11.22) in matrix terms:

$$\begin{aligned}\overline{\text{PBE}}(\mathbf{w}) &= \|\Pi \bar{\delta}_{\mathbf{w}}\|_{\mu}^2 \\ &= (\mathbf{I} \bar{\delta}_{\mathbf{w}})^{\top} \mathbf{D} \Pi \bar{\delta}_{\mathbf{w}} \\ &= \bar{\delta}_{\mathbf{w}}^{\top} \Pi^{\top} \mathbf{D} \Pi \bar{\delta}_{\mathbf{w}} \\ &= \bar{\delta}_{\mathbf{w}}^{\top} \mathbf{D} \mathbf{X} (\mathbf{X}^{\top} \mathbf{D} \mathbf{X})^{-1} \mathbf{X}^{\top} \mathbf{D} \bar{\delta}_{\mathbf{w}}\end{aligned}\tag{from (11.14)}$$

$$\begin{aligned}(\text{using (11.13) and the identity } \Pi^{\top} \mathbf{D} \Pi = \mathbf{D} \mathbf{X} (\mathbf{X}^{\top} \mathbf{D} \mathbf{X})^{-1} \mathbf{X}^{\top} \mathbf{D}) \\ = (\mathbf{X}^{\top} \mathbf{D} \bar{\delta}_{\mathbf{w}})^{\top} (\mathbf{X}^{\top} \mathbf{D} \mathbf{X})^{-1} (\mathbf{X}^{\top} \mathbf{D} \bar{\delta}_{\mathbf{w}}).\end{aligned}\tag{11.26}$$

The gradient with respect to \mathbf{w} is

$$\nabla \overline{\text{PBE}}(\mathbf{w}) = 2 \nabla [\mathbf{X}^{\top} \mathbf{D} \bar{\delta}_{\mathbf{w}}]^{\top} (\mathbf{X}^{\top} \mathbf{D} \mathbf{X})^{-1} (\mathbf{X}^{\top} \mathbf{D} \bar{\delta}_{\mathbf{w}}).$$

To turn this into an SGD method, we have to sample something on every time step that has this quantity as its expected value. Let us take μ to be the distribution of states visited under the behavior policy. All three of the factors above can then be written in terms of expectations under this distribution. For example, the last factor can be written

$$\mathbf{X}^{\top} \mathbf{D} \bar{\delta}_{\mathbf{w}} = \sum_s \mu(s) \mathbf{x}(s) \bar{\delta}_{\mathbf{w}}(s) = \mathbb{E}[\rho_t \delta_t \mathbf{x}_t],$$

which is just the expectation of the semi-gradient TD(0) update (11.2). The first factor is the transpose of the gradient of this update:

$$\begin{aligned}\nabla \mathbb{E}[\rho_t \delta_t \mathbf{x}_t]^{\top} &= \mathbb{E}[\rho_t \nabla \delta_t^{\top} \mathbf{x}_t^{\top}] \\ &= \mathbb{E}[\rho_t \nabla (R_{t+1} + \gamma \mathbf{w}^{\top} \mathbf{x}_{t+1} - \mathbf{w}^{\top} \mathbf{x}_t)^{\top} \mathbf{x}_t^{\top}] \\ &= \mathbb{E}[\rho_t (\gamma \mathbf{x}_{t+1} - \mathbf{x}_t)^{\top}].\end{aligned}\tag{using episodic δ_t }$$

Finally, the middle factor is the inverse of the expected outer-product matrix of the feature vectors:

$$\mathbf{X}^{\top} \mathbf{D} \mathbf{X} = \sum_s \mu(s) \mathbf{x}(s) \mathbf{x}(s)^{\top} = \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^{\top}].$$

Substituting these expectations for the three factors in our expression for the gradient of the $\overline{\text{PBE}}$, we get

$$\nabla \overline{\text{PBE}}(\mathbf{w}) = 2\mathbb{E}[\rho_t(\gamma \mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top] \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t]. \quad (11.27)$$

It might not be obvious that we have made any progress by writing the gradient in this form. It is a product of three expressions and the first and last are not independent. They both depend on the next feature vector \mathbf{x}_{t+1} ; we cannot simply sample both of these expectations and then multiply the samples. This would give us a biased estimate of the gradient just as in the residual-gradient algorithm.

Another idea would be to estimate the three expectations separately and then combine them to produce an unbiased estimate of the gradient. This would work, but would require a lot of computational resources, particularly to store the first two expectations, which are $d \times d$ matrices, and to compute the inverse of the second. This idea can be improved. If two of the three expectations are estimated and stored, then the third could be sampled and used in conjunction with the two stored quantities. For example, you could store estimates of the second two quantities (using the increment inverse-updating techniques in Section 9.8) and then sample the first expression. Unfortunately, the overall algorithm would still be of quadratic complexity (of order $O(d^2)$).

The idea of storing some estimates separately and then combining them with samples is a good one and is also used in Gradient-TD methods. Gradient-TD methods estimate and store *the product* of the second two factors in (11.27). These factors are a $d \times d$ matrix and a d -vector, so their product is just a d -vector, like \mathbf{w} itself. We denote this second learned vector as \mathbf{v} :

$$\mathbf{v} \approx \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t]. \quad (11.28)$$

This form is familiar to students of linear supervised learning. It is the solution to a linear least-squares problem that tries to approximate $\rho_t \delta_t$ from the features. The standard SGD method for incrementally finding the vector \mathbf{v} that minimizes the expected squared error $(\mathbf{v}^\top \mathbf{x}_t - \rho_t \delta_t)^2$ is known as the Least Mean Square (LMS) rule (here augmented with an importance sampling ratio):

$$\mathbf{v}_{t+1} \doteq \mathbf{v}_t + \beta \rho_t (\delta_t - \mathbf{v}_t^\top \mathbf{x}_t) \mathbf{x}_t,$$

where $\beta > 0$ is another step-size parameter. We can use this method to effectively achieve (11.28) with $O(d)$ storage and per-step computation.

Given a stored estimate \mathbf{v}_t approximating (11.28), we can update our main parameter vector \mathbf{w}_t using SGD methods based on (11.27). The simplest such rule is

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2} \alpha \nabla \overline{\text{PBE}}(\mathbf{w}_t) && \text{(the general SGD rule)} \\ &= \mathbf{w}_t - \frac{1}{2} \alpha 2\mathbb{E}[\rho_t(\gamma \mathbf{x}_{t+1} - \mathbf{x}_t)\mathbf{x}_t^\top] \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] && \text{(from (11.27))} \\ &= \mathbf{w}_t + \alpha \mathbb{E}[\rho_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})\mathbf{x}_t^\top] \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] && \text{(11.29)} \\ &\approx \mathbf{w}_t + \alpha \mathbb{E}[\rho_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})\mathbf{x}_t^\top] \mathbf{v}_t && \text{(based on (11.28))} \\ &\approx \mathbf{w}_t + \alpha \rho_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1}) \mathbf{x}_t^\top \mathbf{v}_t. && \text{(sampling)} \end{aligned}$$

This algorithm is called *GTD2*. Note that if the final inner product ($\mathbf{x}_t^\top \mathbf{v}_t$) is done first, then the entire algorithm is of $O(d)$ complexity.

A slightly better algorithm can be derived by doing a few more analytic steps before substituting in \mathbf{v}_t . Continuing from (11.29):

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha \mathbb{E}[\rho_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}) \mathbf{x}_t^\top] \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha (\mathbb{E}[\rho_t \mathbf{x}_t \mathbf{x}_t^\top] - \gamma \mathbb{E}[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top]) \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha (\mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top] - \gamma \mathbb{E}[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top]) \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha (\mathbb{E}[\rho_t \delta_t \mathbf{x}_t] - \gamma \mathbb{E}[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top] \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[\rho_t \delta_t \mathbf{x}_t]) \\
&\approx \mathbf{w}_t + \alpha (\mathbb{E}[\rho_t \delta_t \mathbf{x}_t] - \gamma \mathbb{E}[\rho_t \mathbf{x}_{t+1} \mathbf{x}_t^\top] \mathbf{v}_t) \quad (\text{based on (11.28)}) \\
&\approx \mathbf{w}_t + \alpha \rho_t (\delta_t \mathbf{x}_t - \gamma \mathbf{x}_{t+1} \mathbf{x}_t^\top \mathbf{v}_t), \quad (\text{sampling})
\end{aligned}$$

which again is $O(d)$ if the final product ($\mathbf{x}_t^\top \mathbf{v}_t$) is done first. This algorithm is known as either *TD(0)* with gradient correction (*TDC*) or, alternatively, as *GTD(0)*.

Figure 11.5 shows a sample and the expected behavior of TDC on Baird's counterexample. As intended, the $\overline{\text{PBE}}$ falls to zero, but note that the individual components of the parameter vector do not approach zero. In fact, these values are still far from

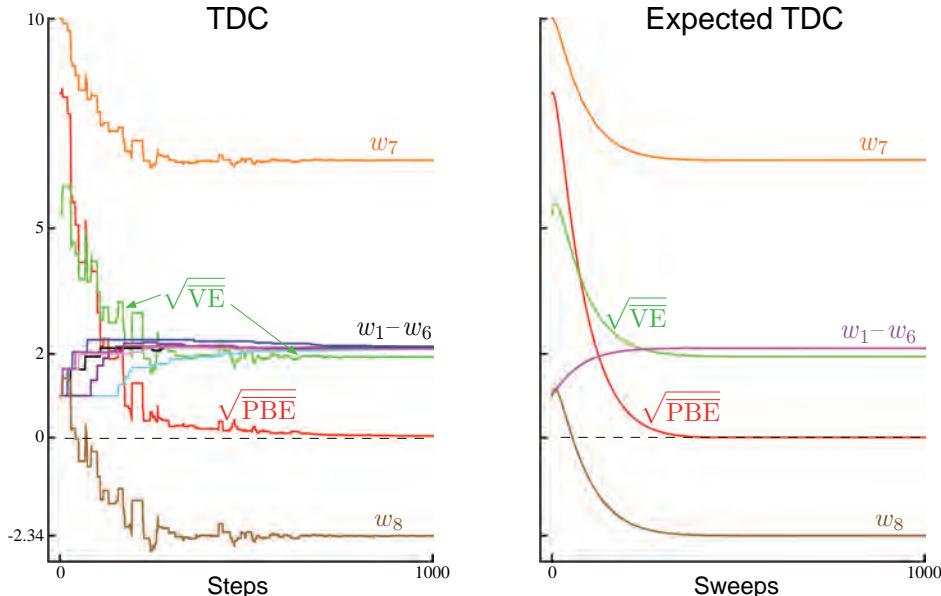


Figure 11.5: The behavior of the TDC algorithm on Baird's counterexample. On the left is shown a typical single run, and on the right is shown the expected behavior of this algorithm if the updates are done synchronously (analogous to (11.9), except for the two TDC parameter vectors). The step sizes were $\alpha = 0.005$ and $\beta = 0.05$.

an optimal solution, $\hat{v}(s) = 0$, for all s , for which \mathbf{w} would have to be proportional to $(1, 1, 1, 1, 1, 1, 4, -2)^\top$. After 1000 iterations we are still far from an optimal solution, as we can see from the $\overline{\text{VE}}$, which remains almost 2. The system is actually converging to an optimal solution, but progress is extremely slow because the $\overline{\text{PBE}}$ is already so close to zero.

GTD2 and TDC both involve two learning processes, a primary one for \mathbf{w} and a secondary one for \mathbf{v} . The logic of the primary learning process relies on the secondary learning process having finished, at least approximately, whereas the secondary learning process proceeds without being influenced by the first. We call this sort of asymmetrical dependence a *cascade*. In cascades we often assume that the secondary learning process is proceeding faster and thus is always at its asymptotic value, ready and accurate to assist the primary learning process. The convergence proofs for these methods often make this assumption explicitly. These are called *two-time-scale* proofs. The fast time scale is that of the secondary learning process, and the slower time scale is that of the primary learning process. If α is the step size of the primary learning process, and β is the step size of the secondary learning process, then these convergence proofs will typically require that in the limit $\beta \rightarrow 0$ and $\frac{\alpha}{\beta} \rightarrow 0$.

Gradient-TD methods are currently the most well understood and widely used stable off-policy methods. There are extensions to action values and control (GQ, Maei et al., 2010), to eligibility traces (GTD(λ) and GQ(λ), Maei, 2011; Maei and Sutton, 2010), and to nonlinear function approximation (Maei et al., 2009). There have also been proposed hybrid algorithms midway between semi-gradient TD and gradient TD (Hackman, 2012; White and White, 2016). Hybrid-TD algorithms behave like Gradient-TD algorithms in states where the target and behavior policies are very different, and behave like semi-gradient algorithms in states where the target and behavior policies are the same. Finally, the Gradient-TD idea has been combined with the ideas of proximal methods and control variates to produce more efficient methods (Mahadevan et al., 2014; Du et al., 2017).

11.8 Emphatic-TD Methods

We turn now to the second major strategy that has been extensively explored for obtaining a cheap and efficient off-policy learning method with function approximation. Recall that linear semi-gradient TD methods are efficient and stable when trained under the on-policy distribution, and that we showed in Section 9.4 that this has to do with the positive definiteness of the matrix \mathbf{A} (9.11)⁴ and the match between the on-policy state distribution μ_π and the state-transition probabilities $p(s|s, a)$ under the target policy. In off-policy learning, we reweight the state transitions using importance sampling so that they become appropriate for learning about the target policy, but the state distribution is still that of the behavior policy. There is a mismatch. A natural idea is to somehow reweight the states, emphasizing some and de-emphasizing others, so as to return the distribution of updates to the on-policy distribution. There would then be a match, and stability and convergence would follow from existing results. This is the idea of

⁴In the off-policy case, the matrix \mathbf{A} is generally defined as $\mathbb{E}_{s \sim b}[\mathbf{x}(s)\mathbb{E}[\mathbf{x}(S_{t+1})^\top | S_t = s, A_t \sim \pi]]$.

Emphatic-TD methods, first introduced for on-policy training in Section 9.11.

Actually, the notion of “the on-policy distribution” is not quite right, as there are many on-policy distributions, and any one of these is sufficient to guarantee stability. Consider an undiscounted episodic problem. The way episodes terminate is fully determined by the transition probabilities, but there may be several different ways the episodes might begin. However the episodes start, if all state transitions are due to the target policy, then the state distribution that results is an on-policy distribution. You might start close to the terminal state and visit only a few states with high probability before ending the episode. Or you might start far away and pass through many states before terminating. Both are on-policy distributions, and training on both with a linear semi-gradient method would be guaranteed to be stable. However the process starts, an on-policy distribution results as long as all states encountered are updated up until termination.

If there is discounting, it can be treated as partial or probabilistic termination for these purposes. If $\gamma = 0.9$, then we can consider that with probability 0.1 the process terminates on every time step and then immediately restarts in the state that is transitioned to. A discounted problem is one that is continually terminating and restarting with probability $1 - \gamma$ on every step. This way of thinking about discounting is an example of a more general notion of *pseudo termination*—termination that does not affect the sequence of state transitions, but does affect the learning process and the quantities being learned. This kind of pseudo termination is important to off-policy learning because the restarting is optional—remember we can start any way we want to—and the termination relieves the need to keep including encountered states within the on-policy distribution. That is, if we don’t consider the new states as restarts, then discounting quickly gives us a limited on-policy distribution.

The one-step Emphatic-TD algorithm for learning episodic state values is defined by:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t),$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha M_t \rho_t \delta_t \nabla \hat{v}(S_t, \mathbf{w}_t),$$

$$M_t = \gamma \rho_{t-1} M_{t-1} + I_t,$$

with I_t , the *interest*, being arbitrary and M_t , the *emphasis*, being initialized to $M_{-1} = 0$. How does this algorithm perform on Baird’s counterexample? Figure 11.6 shows the trajectory in expectation of the components of the parameter vector (for the case in which $I_t = 1$, for all t). There are some oscillations but eventually everything converges and the $\bar{V}\mathbb{E}$ goes to zero. These trajectories are obtained by iteratively computing the expectation of the parameter vector trajectory without any of the variance due to sampling of transitions and rewards. We do not show the results of applying the Emphatic-TD algorithm directly because its variance on Baird’s counterexample is so high that it is nigh impossible to get consistent results in computational experiments. The algorithm converges to the optimal solution in theory on this problem, but in practice it does not. We turn to the topic of reducing the variance of all these algorithms in the next section.

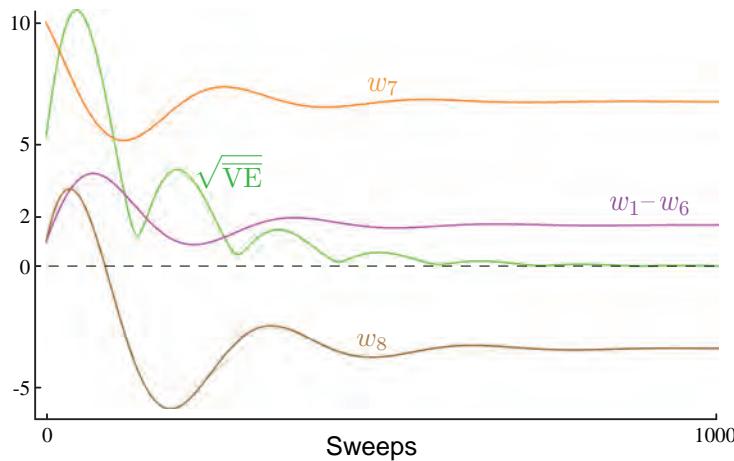


Figure 11.6: The behavior of the one-step Emphatic-TD algorithm in expectation on Baird’s counterexample. The step size was $\alpha = 0.03$.

11.9 Reducing Variance

Off-policy learning is inherently of greater variance than on-policy learning. This is not surprising; if you receive data less closely related to a policy, you should expect to learn less about the policy’s values. In the extreme, one may be able to learn nothing. You can’t expect to learn how to drive by cooking dinner, for example. Only if the target and behavior policies are related, if they visit similar states and take similar actions, should one be able to make significant progress in off-policy training.

On the other hand, any policy has many neighbors, many similar policies with considerable overlap in states visited and actions chosen, and yet which are not identical. The raison d’être of off-policy learning is to enable generalization to this vast number of related-but-not-identical policies. The problem remains of how to make the best use of the experience. Now that we have some methods that are stable in expected value (if the step sizes are set right), attention naturally turns to reducing the variance of the estimates. There are many possible ideas, and we can just touch on a few of them in this introductory text.

Why is controlling variance especially critical in off-policy methods based on importance sampling? As we have seen, importance sampling often involves products of policy ratios. The ratios are always one in expectation (5.13), but their actual values may be very high or as low as zero. Successive ratios are uncorrelated, so their products are also always one in expected value, but they can be of very high variance. Recall that these ratios multiply the step size in SGD methods, so high variance means taking steps that vary greatly in their sizes. This is problematic for SGD because of the occasional very large steps. They must not be so large as to take the parameter to a part of the space with a very different gradient. SGD methods rely on averaging over multiple steps to get a good sense of the gradient, and if they make large moves from single samples they become unreliable. If the step-size parameter is set small enough to prevent this, then the expected step

can end up being very small, resulting in very slow learning. The notions of momentum (Derthick, 1984), of Polyak-Ruppert averaging (Polyak, 1990; Ruppert, 1988; Polyak and Juditsky, 1992), or further extensions of these ideas may significantly help. Methods for adaptively setting separate step sizes for different components of the parameter vector are also pertinent (e.g., Jacobs, 1988; Sutton, 1992b, c), as are the “importance weight aware” updates of Karampatziakis and Langford (2010).

In Chapter 5 we saw how weighted importance sampling is significantly better behaved, with lower variance updates, than ordinary importance sampling. However, adapting weighted importance sampling to function approximation is challenging and can probably only be done approximately with $O(d)$ complexity (Mahmood and Sutton, 2015).

The Tree Backup algorithm (Section 7.5) shows that it is possible to perform some off-policy learning without using importance sampling. This idea has been extended to the off-policy case to produce stable and more efficient methods by Munos, Stepleton, Harutyunyan, and Bellemare (2016) and by Mahmood, Yu and Sutton (2017).

Another, complementary strategy is to allow the target policy to be determined in part by the behavior policy, in such a way that it never can be so different from it to create large importance sampling ratios. For example, the target policy can be defined by reference to the behavior policy, as in the “recognizers” proposed by Precup et al. (2006).

11.10 Summary

Off-policy learning is a tempting challenge, testing our ingenuity in designing stable and efficient learning algorithms. Tabular Q-learning makes off-policy learning seem easy, and it has natural generalizations to Expected Sarsa and to the Tree Backup algorithm. But as we have seen in this chapter, the extension of these ideas to significant function approximation, even linear function approximation, involves new challenges and forces us to deepen our understanding of reinforcement learning algorithms.

Why go to such lengths? One reason to seek off-policy algorithms is to give flexibility in dealing with the tradeoff between exploration and exploitation. Another is to free behavior from learning, and avoid the tyranny of the target policy. TD learning appears to hold out the possibility of learning about multiple things in parallel, of using one stream of experience to solve many tasks simultaneously. We can certainly do this in special cases, just not in every case that we would like to or as efficiently as we would like to.

In this chapter we divided the challenge of off-policy learning into two parts. The first part, correcting the targets of learning for the behavior policy, is straightforwardly dealt with using the techniques devised earlier for the tabular case, albeit at the cost of increasing the variance of the updates and thereby slowing learning. High variance will probably always remain a challenge for off-policy learning.

The second part of the challenge of off-policy learning emerges as the instability of semi-gradient TD methods that involve bootstrapping. We seek powerful function approximation, off-policy learning, and the efficiency and flexibility of bootstrapping

TD methods, but it is challenging to combine all three aspects of this *deadly triad* in one algorithm without introducing the potential for instability. There have been several attempts. The most popular has been to seek to perform true stochastic gradient descent (SGD) in the Bellman error (a.k.a. the Bellman residual). However, our analysis concludes that this is not an appealing goal in many cases, and that anyway it is impossible to achieve with a learning algorithm—the gradient of the $\overline{\text{BE}}$ is not learnable from experience that reveals only feature vectors and not underlying states. Another approach, Gradient-TD methods, performs SGD in the *projected* Bellman error. The gradient of the $\overline{\text{PBE}}$ is learnable with $O(d)$ complexity, but at the cost of a second parameter vector with a second step size. The newest family of methods, Emphatic-TD methods, refine an old idea for reweighting updates, emphasizing some and de-emphasizing others. In this way they restore the special properties that make on-policy learning stable with computationally simple semi-gradient methods.

The whole area of off-policy learning is relatively new and unsettled. Which methods are best or even adequate is not yet clear. Are the complexities of the new methods introduced at the end of this chapter really necessary? Which of them can be combined effectively with variance reduction methods? The potential for off-policy learning remains tantalizing, the best way to achieve it still a mystery.

Bibliographical and Historical Remarks

- 11.1** The first semi-gradient method was linear TD(λ) (Sutton, 1988). The name “semi-gradient” is more recent (Sutton, 2015a). Semi-gradient off-policy TD(0) with general importance-sampling ratio may not have been explicitly stated until Sutton, Mahmood, and White (2016), but the action-value forms were introduced by Precup, Sutton, and Singh (2000), who also did eligibility trace forms of these algorithms (see Chapter 12). Their continuing, undiscounted forms have not been significantly explored. The n -step forms given here are new.
- 11.2** The earliest w -to- $2w$ example was given by Tsitsiklis and Van Roy (1996), who also introduced the specific counterexample in the box on page 263. Baird’s counterexample is due to Baird (1995), though the version we present here is slightly modified. Averaging methods for function approximation were developed by Gordon (1995, 1996b). Other examples of instability with off-policy DP methods and more complex methods of function approximation are given by Boyan and Moore (1995). Bradtko (1993) gives an example in which Q-learning using linear function approximation in a linear quadratic regulation problem converges to a destabilizing policy.
- 11.3** The deadly triad was first identified by Sutton (1995b) and thoroughly analyzed by Tsitsiklis and Van Roy (1997). The name “deadly triad” is due to Sutton (2015a).
- 11.4** This kind of linear analysis was pioneered by Tsitsiklis and Van Roy (1996; 1997), including the dynamic programming operator. Diagrams like Figure 11.3 were

introduced by Lagoudakis and Parr (2003).

What we have called the Bellman operator, and denoted B_π , is more commonly denoted T^π and called a “dynamic programming operator,” while a generalized form, denoted $T^{(\lambda)}$, is called the “TD(λ) operator” (Tsitsiklis and Van Roy, 1996, 1997).

- 11.5** The \overline{BE} was first proposed as an objective function for dynamic programming by Schweitzer and Seidmann (1985). Baird (1995, 1999) extended it to TD learning based on stochastic gradient descent. In the literature, \overline{BE} minimization is often referred to as Bellman residual minimization.

The earliest A-split example is due to Dayan (1992). The two forms given here were introduced by Sutton et al. (2009a).

- 11.6** The contents of this section are new to this text.

- 11.7** Gradient-TD methods were introduced by Sutton, Szepesvari, and Maei (2009b). The methods highlighted in this section were introduced by Sutton et al. (2009a) and Mahmood et al. (2014). A major extension to proximal TD methods was developed by Mahadevan et al. (2014). The most sensitive empirical investigations to date of Gradient-TD and related methods are given by Geist and Scherrer (2014), Dann, Neumann, and Peters (2014), White (2015), and Ghiassian, Patterson, White, Sutton, and White (2018). Recent developments in the theory of Gradient-TD methods are presented by Yu (2017).

- 11.8** Emphatic-TD methods were introduced by Sutton, Mahmood, and White (2016). Full convergence proofs and other theory were later established by Yu (2015; 2016; Yu, Mahmood, and Sutton, 2017), Hallak, Tamar, and Mannor (2015), and Hallak, Tamar, Munos, and Mannor (2016).

Chapter 12

Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning. For example, in the popular $\text{TD}(\lambda)$ algorithm, the λ refers to the use of an eligibility trace. Almost any temporal-difference (TD) method, such as Q-learning or Sarsa, can be combined with eligibility traces to obtain a more general method that may learn more efficiently.

Eligibility traces unify and generalize TD and Monte Carlo methods. When TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end ($\lambda=1$) and one-step TD methods at the other ($\lambda=0$). In between are intermediate methods that are often better than either extreme method. Eligibility traces also provide a way of implementing Monte Carlo methods online and on continuing problems without episodes.

Of course, we have already seen one way of unifying TD and Monte Carlo methods: the n -step TD methods of Chapter 7. What eligibility traces offer beyond these is an elegant algorithmic mechanism with significant computational advantages. The mechanism is a short-term memory vector, the *eligibility trace* $\mathbf{z}_t \in \mathbb{R}^d$, that parallels the long-term weight vector $\mathbf{w}_t \in \mathbb{R}^d$. The rough idea is that when a component of \mathbf{w}_t participates in producing an estimated value, then the corresponding component of \mathbf{z}_t is bumped up and then begins to fade away. Learning will then occur in that component of \mathbf{w}_t if a nonzero TD error occurs before the trace falls back to zero. The trace-decay parameter $\lambda \in [0, 1]$ determines the rate at which the trace falls.

The primary computational advantage of eligibility traces over n -step methods is that only a single trace vector is required rather than a store of the last n feature vectors. Learning also occurs continually and uniformly in time rather than being delayed and then catching up at the end of the episode. In addition learning can occur and affect behavior immediately after a state is encountered rather than being delayed n steps.

Eligibility traces illustrate that a learning algorithm can sometimes be implemented in a different way to obtain computational advantages. Many algorithms are most naturally formulated and understood as an update of a state's value based on events that follow that state over multiple future time steps. For example, Monte Carlo methods (Chapter 5) update a state based on all the future rewards, and n -step TD methods (Chapter 7)

update based on the next n rewards and state n steps in the future. Such formulations, based on looking forward from the updated state, are called *forward views*. Forward views are always somewhat complex to implement because the update depends on later things that are not available at the time. However, as we show in this chapter it is often possible to achieve nearly the same updates—and sometimes *exactly* the same updates—with an algorithm that uses the current TD error, looking backward to recently visited states using an eligibility trace. These alternate ways of looking at and implementing learning algorithms are called *backward views*. Backward views, transformations between forward views and backward views, and equivalences between them, date back to the introduction of temporal difference learning but have become much more powerful and sophisticated since 2014. Here we present the basics of the modern view.

As usual, first we fully develop the ideas for state values and prediction, then extend them to action values and control. We develop them first for the on-policy case then extend them to off-policy learning. Our treatment pays special attention to the case of linear function approximation, for which the results with eligibility traces are stronger. All these results apply also to the tabular and state aggregation cases because these are special cases of linear function approximation.

12.1 The λ -return

In Chapter 7 we defined an n -step return as the sum of the first n rewards plus the estimated value of the state reached in n steps, each appropriately discounted (7.1). The general form of that equation, for any parameterized function approximator, is

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T-n, \quad (12.1)$$

where $\hat{v}(s, \mathbf{w})$ is the approximate value of state s given weight vector \mathbf{w} (Chapter 9), and T is the time of episode termination, if any. We noted in Chapter 7 that each n -step return, for $n \geq 1$, is a valid update target for a tabular learning update, just as it is for an approximate SGD learning update such as (9.7).

Now we note that a valid update can be done not just toward any n -step return, but toward any *average* of n -step returns for different n s. For example, an update can be done toward a target that is half of a two-step return and half of a four-step return: $\frac{1}{2}G_{t:t+2} + \frac{1}{2}G_{t:t+4}$. Any set of n -step returns can be averaged in this way, even an infinite set, as long as the weights on the component returns are positive and sum to 1. The composite return possesses an error reduction property similar to that of individual n -step returns (7.3) and thus can be used to construct updates with guaranteed convergence properties. Averaging produces a substantial new range of algorithms. For example, one could average one-step and infinite-step returns to obtain another way of interrelating TD and Monte Carlo methods. In principle, one could even average experience-based updates with DP updates to get a simple combination of experience-based and model-based methods (cf. Chapter 8).

An update that averages simpler component updates is called a *compound update*. The backup diagram for a compound update consists of the backup diagrams for each of the component updates with a horizontal line above them and the weighting fractions below.

For example, the compound update for the case mentioned at the start of this section, mixing half of a two-step return and half of a four-step return, has the diagram shown to the right. A compound update can only be done when the longest of its component updates is complete. The update at the right, for example, could only be done at time $t+4$ for the estimate formed at time t . In general one would like to limit the length of the longest component update because of the corresponding delay in the updates.

The TD(λ) algorithm can be understood as one particular way of averaging n -step updates. This average contains all the n -step updates, each weighted proportionally to λ^{n-1} (where $\lambda \in [0, 1]$), and is normalized by a factor of $1 - \lambda$ to ensure that the weights sum to 1 (Figure 12.1). The resulting update is toward a return, called the λ -return, defined in its state-based form by

$$G_t^\lambda \doteq (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}. \quad (12.2)$$

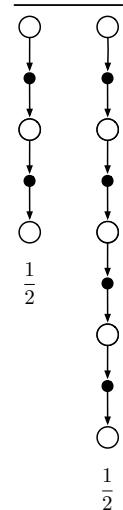


Figure 12.2 further illustrates the weighting on the sequence of n -step returns in the λ -return. The one-step return is given the largest weight, $1 - \lambda$; the two-step return is given the next largest weight, $(1 - \lambda)\lambda$; the three-step return is given the weight $(1 - \lambda)\lambda^2$; and so on. The weight fades by λ with each additional step. After a terminal state has been reached, all subsequent n -step returns are equal to the conventional return, G_t . If

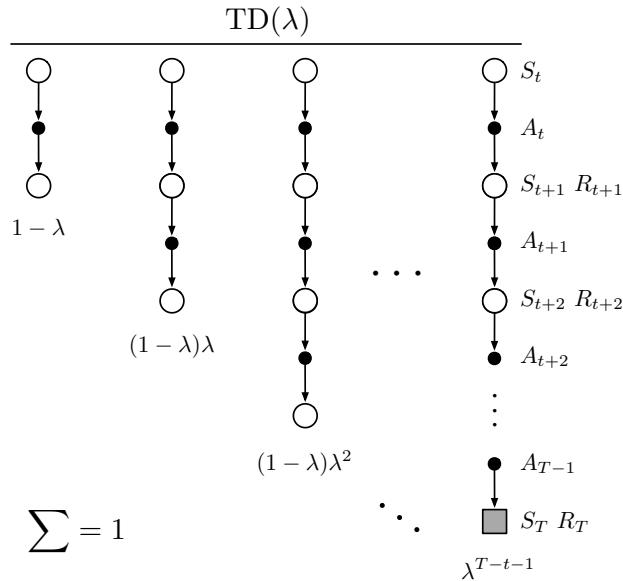


Figure 12.1: The backup diagram for TD(λ). If $\lambda = 0$, then the overall update reduces to its first component, the one-step TD update, whereas if $\lambda = 1$, then the overall update reduces to its last component, the Monte Carlo update.

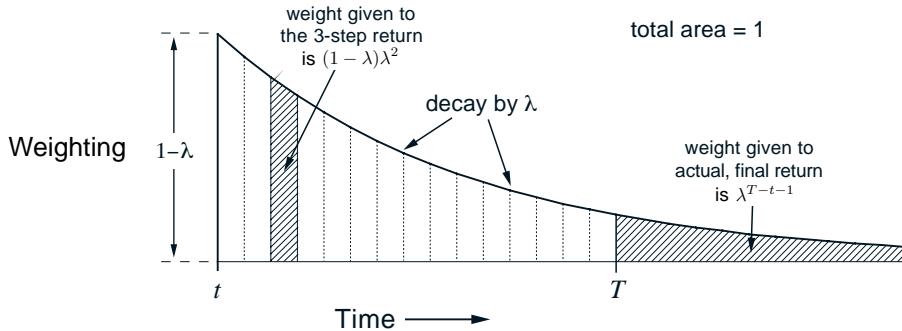


Figure 12.2: Weighting given in the λ -return to each of the n -step returns.

we want, we can separate these post-termination terms from the main sum, yielding

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t, \quad (12.3)$$

as indicated in the figures. This equation makes it clearer what happens when $\lambda = 1$. In this case the main sum goes to zero, and the remaining term reduces to the conventional return. Thus, for $\lambda = 1$, updating according to the λ -return is a Monte Carlo algorithm. On the other hand, if $\lambda = 0$, then the λ -return reduces to $G_{t:t+1}$, the one-step return. Thus, for $\lambda = 0$, updating according to the λ -return is a one-step TD method.

Exercise 12.1 Just as the return can be written recursively in terms of the first reward and itself one-step later (3.9), so can the λ -return. Derive the analogous recursive relationship from (12.2) and (12.1). \square

Exercise 12.2 The parameter λ characterizes how fast the exponential weighting in Figure 12.2 falls off, and thus how far into the future the λ -return algorithm looks in determining its update. But a rate factor such as λ is sometimes an awkward way of characterizing the speed of the decay. For some purposes it is better to specify a time constant, or half-life. What is the equation relating λ and the half-life, τ_λ , the time by which the weighting sequence will have fallen to half of its initial value? \square

We are now ready to define our first learning algorithm based on the λ -return: the *off-line λ -return algorithm*. As an off-line algorithm, it makes no changes to the weight vector during the episode. Then, at the end of the episode, a whole sequence of off-line updates are made according to our usual semi-gradient rule, using the λ -return as the target:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[G_t^\lambda - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t), \quad t = 0, \dots, T-1. \quad (12.4)$$

The λ -return gives us an alternative way of moving smoothly between Monte Carlo and one-step TD methods that can be compared with the n -step bootstrapping way developed in Chapter 7. There we assessed effectiveness on a 19-state random walk task (Example 7.1, page 144). Figure 12.3 shows the performance of the off-line λ -return algorithm on this task alongside that of the n -step methods (repeated from Figure 7.2). The experiment was just as described earlier except that for the λ -return algorithm we varied λ instead of n . The performance measure used is the estimated root-mean-square error between the correct and estimated values of each state measured at the end of the episode, averaged over the first 10 episodes and the 19 states. Note that overall performance of the off-line λ -return algorithms is comparable to that of the n -step algorithms. In both cases we get best performance with an intermediate value of the bootstrapping parameter, n for n -step methods and λ for the off-line λ -return algorithm.

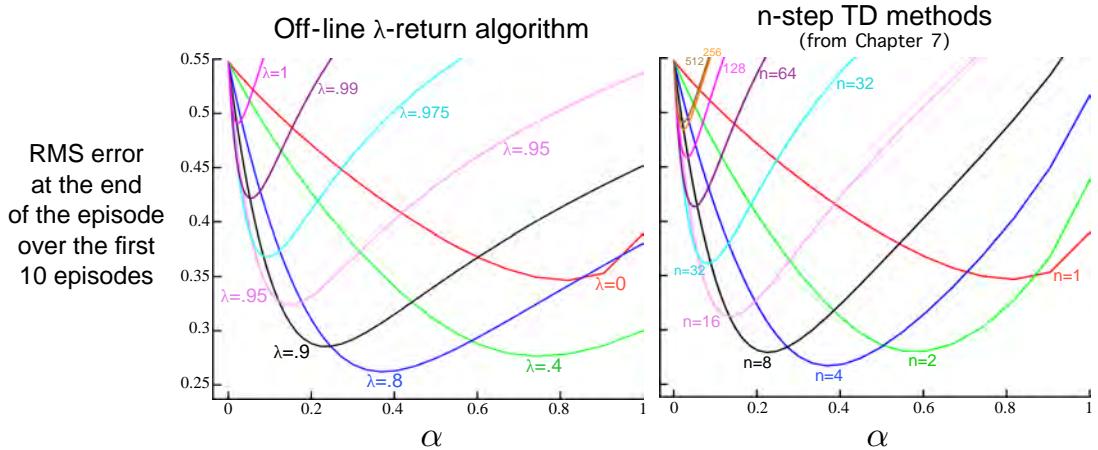


Figure 12.3: 19-state Random walk results (Example 7.1): Performance of the off-line λ -return algorithm alongside that of the n -step TD methods. In both case, intermediate values of the bootstrapping parameter (λ or n) performed best. The results with the off-line λ -return algorithm are slightly better at the best values of α and λ , and at high α .

The approach that we have been taking so far is what we call the theoretical, or *forward*, view of a learning algorithm. For each state visited, we look forward in time to all the future rewards and decide how best to combine them. We might imagine ourselves riding the stream of states, looking forward from each state to determine its update, as suggested by Figure 12.4. After looking forward from and updating one state, we move on to the next and never have to work with the preceding state again. Future states, on the other hand, are viewed and processed repeatedly, once from each vantage point preceding them.

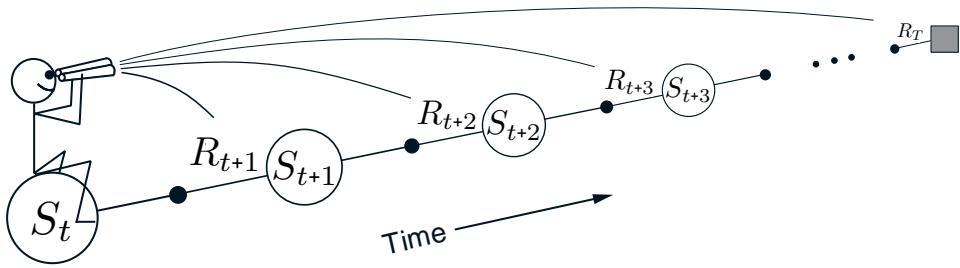


Figure 12.4: The forward view. We decide how to update each state by looking forward to future rewards and states.

12.2 TD(λ)

TD(λ) is one of the oldest and most widely used algorithms in reinforcement learning. It was the first algorithm for which a formal relationship was shown between a more theoretical forward view and a more computationally-congenial backward view using eligibility traces. Here we will show empirically that it approximates the off-line λ -return algorithm presented in the previous section.

TD(λ) improves over the off-line λ -return algorithm in three ways. First it updates the weight vector on every step of an episode rather than only at the end, and thus its estimates may be better sooner. Second, its computations are equally distributed in time rather than all at the end of the episode. And third, it can be applied to continuing problems rather than just to episodic problems. In this section we present the semi-gradient version of TD(λ) with function approximation.

With function approximation, the eligibility trace is a vector $\mathbf{z}_t \in \mathbb{R}^d$ with the same number of components as the weight vector \mathbf{w}_t . Whereas the weight vector is a long-term memory, accumulating over the lifetime of the system, the eligibility trace is a short-term memory, typically lasting less time than the length of an episode. Eligibility traces assist in the learning process; their only consequence is that they affect the weight vector, and then the weight vector determines the estimated value.

In TD(λ), the eligibility trace vector is initialized to zero at the beginning of the episode, is incremented on each time step by the value gradient, and then fades away by $\gamma\lambda$:

$$\begin{aligned}\mathbf{z}_{-1} &\doteq \mathbf{0}, \\ \mathbf{z}_t &\doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t, \mathbf{w}_t), \quad 0 \leq t \leq T,\end{aligned}\tag{12.5}$$

where γ is the discount rate and λ is the parameter introduced in the previous section, which we henceforth call the trace-decay parameter. The eligibility trace keeps track of which components of the weight vector have contributed, positively or negatively, to recent state valuations, where “recent” is defined in terms of $\gamma\lambda$. (Recall that in linear function approximation, $\nabla\hat{v}(S_t, \mathbf{w}_t)$ is the feature vector, \mathbf{x}_t , in which case the eligibility trace vector is just a sum of past, fading, input vectors.) The trace is said to indicate the eligibility of each component of the weight vector for undergoing learning changes

should a reinforcing event occur. The reinforcing events we are concerned with are the moment-by-moment one-step TD errors. The TD error for state-value prediction is

$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t). \quad (12.6)$$

In $TD(\lambda)$, the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t. \quad (12.7)$$

Semi-gradient $TD(\lambda)$ for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated

Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$

Initialize value-function weights \mathbf{w} arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:

 Initialize S

$\mathbf{z} \leftarrow \mathbf{0}$ (a d -dimensional vector)

 Loop for each step of episode:

 Choose $A \sim \pi(\cdot | S)$

 Take action A , observe R, S'

$\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + \nabla \hat{v}(S, \mathbf{w})$

$\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$

$S \leftarrow S'$

 until S' is terminal

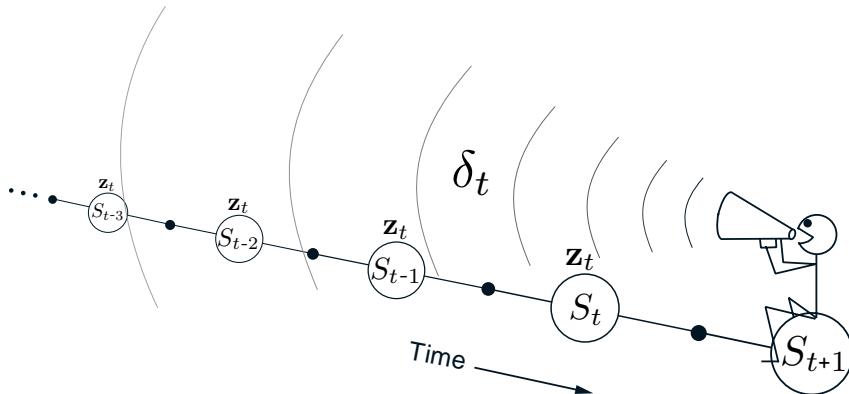


Figure 12.5: The backward or mechanistic view of $TD(\lambda)$. Each update depends on the current TD error combined with the current eligibility traces of past events.

$\text{TD}(\lambda)$ is oriented backward in time. At each moment we look at the current TD error and assign it backward to each prior state according to how much that state contributed to the current eligibility trace at that time. We might imagine ourselves riding along the stream of states, computing TD errors, and shouting them back to the previously visited states, as suggested by Figure 12.5. Where the TD error and traces come together, we get the update given by (12.7), changing the values of those past states for when they occur again in the future.

To better understand the backward view of $\text{TD}(\lambda)$, consider what happens at various values of λ . If $\lambda = 0$, then by (12.5) the trace at t is exactly the value gradient corresponding to S_t . Thus the $\text{TD}(\lambda)$ update (12.7) reduces to the one-step semi-gradient TD update treated in Chapter 9 (and, in the tabular case, to the simple TD rule (6.2)). This is why that algorithm was called $\text{TD}(0)$. In terms of Figure 12.5, $\text{TD}(0)$ is the case in which only the one state preceding the current one is updated by the TD error (other states may have their value estimates changed by generalization due to function approximation). For larger values of λ , but still $\lambda < 1$, more of the preceding states are updated, but each more temporally distant state is updated less because the corresponding eligibility trace is smaller, as suggested by the figure. We say that the earlier states are given less *credit* for the TD error.

If $\lambda = 1$, then the credit given to earlier states falls only by γ per step. This turns out to be just the right thing to do to achieve Monte Carlo behavior. For example, remember that the TD error, δ_t , includes an undiscounted term of R_{t+1} . In passing this back k steps it needs to be discounted, like any reward in a return, by γ^k , which is just what the falling eligibility trace achieves. If $\lambda = 1$ and $\gamma = 1$, then the eligibility traces do not decay at all with time. In this case the method behaves like a Monte Carlo method for an undiscounted, episodic task. If $\lambda = 1$, the algorithm is also known as $\text{TD}(1)$.

$\text{TD}(1)$ is a way of implementing Monte Carlo algorithms that is more general than those presented earlier and that significantly increases their range of applicability. Whereas the earlier Monte Carlo methods were limited to episodic tasks, $\text{TD}(1)$ can be applied to discounted continuing tasks as well. Moreover, $\text{TD}(1)$ can be performed incrementally and online. One disadvantage of Monte Carlo methods is that they learn nothing from an episode until it is over. For example, if a Monte Carlo control method takes an action that produces a very poor reward but does not end the episode, then the agent's tendency to repeat the action will be undiminished during the episode. Online $\text{TD}(1)$, on the other hand, learns in an n -step TD way from the incomplete ongoing episode, where the n steps are all the way up to the current step. If something unusually good or bad happens during an episode, control methods based on $\text{TD}(1)$ can learn immediately and alter their behavior on that same episode.

It is revealing to revisit the 19-state random walk example (Example 7.1) to see how well $\text{TD}(\lambda)$ does in approximating the off-line λ -return algorithm. The results for both algorithms are shown in Figure 12.6. For each λ value, if α is selected optimally for it (or smaller), then the two algorithms perform virtually identically. If α is chosen larger than is optimal, however, then the λ -return algorithm is only a little worse whereas $\text{TD}(\lambda)$ is much worse and may even be unstable. This is not catastrophic for $\text{TD}(\lambda)$ on this problem, as these higher parameter values are not what one would want to use anyway, but for other problems it can be a significant weakness.

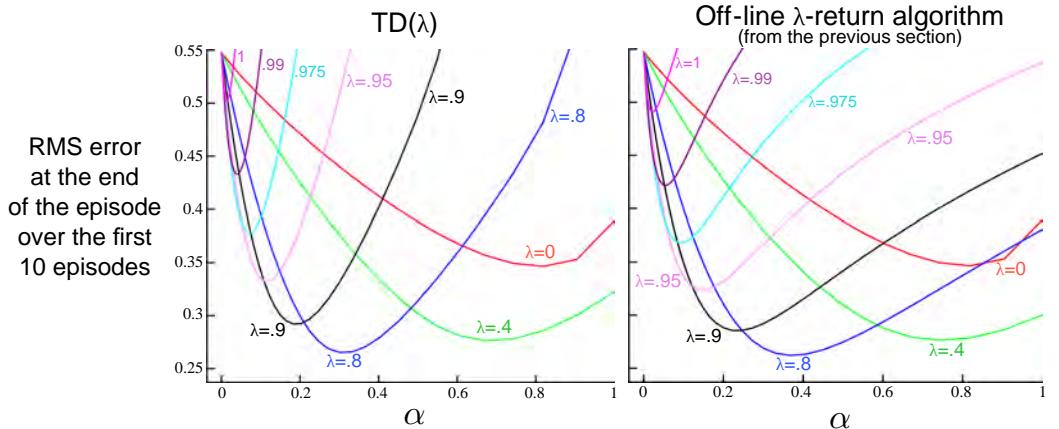


Figure 12.6: 19-state Random walk results (Example 7.1): Performance of $\text{TD}(\lambda)$ alongside that of the off-line λ -return algorithm. The two algorithms performed virtually identically at low (less than optimal) α values, but $\text{TD}(\lambda)$ was worse at high α values.

Linear $\text{TD}(\lambda)$ has been proved to converge in the on-policy case if the step-size parameter is reduced over time according to the usual conditions (2.7). Just as discussed in Section 9.4, convergence is not to the minimum-error weight vector, but to a nearby weight vector that depends on λ . The bound on solution quality presented in that section (9.14) can now be generalized to apply for any λ . For the continuing discounted case,

$$\overline{\text{VE}}(\mathbf{w}_\infty) \leq \frac{1-\gamma\lambda}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}). \quad (12.8)$$

That is, the asymptotic error is no more than $\frac{1-\gamma\lambda}{1-\gamma}$ times the smallest possible error. As λ approaches 1, the bound approaches the minimum error (and it is loosest at $\lambda=0$). In practice, however, $\lambda=1$ is often the poorest choice, as will be illustrated later in Figure 12.14.

Exercise 12.3 Some insight into how $\text{TD}(\lambda)$ can closely approximate the off-line λ -return algorithm can be gained by seeing that the latter's error term (in brackets in (12.4)) can be written as the sum of TD errors (12.6) for a single fixed \mathbf{w} . Show this, following the pattern of (6.6), and using the recursive relationship for the λ -return you obtained in Exercise 12.1. \square

Exercise 12.4 Use your result from the preceding exercise to show that, if the weight updates over an episode were computed on each step but not actually used to change the weights (\mathbf{w} remained fixed), then the sum of $\text{TD}(\lambda)$'s weight updates would be the same as the sum of the off-line λ -return algorithm's updates. \square

12.3 *n*-step Truncated λ -return Methods

The off-line λ -return algorithm is an important ideal, but it is of limited utility because it uses the λ -return (12.2), which is not known until the end of the episode. In the

continuing case, the λ -return is technically never known, as it depends on n -step returns for arbitrarily large n , and thus on rewards arbitrarily far in the future. However, the dependence becomes weaker for longer-delayed rewards, falling by $\gamma\lambda$ for each step of delay. A natural approximation, then, would be to truncate the sequence after some number of steps. Our existing notion of n -step returns provides a natural way to do this in which the missing rewards are replaced with estimated values.

In general, we define the *truncated λ -return* for time t , given data only up to some later horizon, h , as

$$G_{t:h}^\lambda \doteq (1 - \lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}, \quad 0 \leq t < h \leq T. \quad (12.9)$$

If you compare this equation with the λ -return (12.3), it is clear that the horizon h is playing the same role as was previously played by T , the time of termination. Whereas in the λ -return there is a residual weight given to the conventional return G_t , here it is given to the longest available n -step return, $G_{t:h}$ (Figure 12.2).

The truncated λ -return immediately gives rise to a family of n -step λ -return algorithms similar to the n -step methods of Chapter 7. In all of these algorithms, updates are delayed by n steps and only take into account the first n rewards, but now all the k -step returns are included for $1 \leq k \leq n$ (whereas the earlier n -step algorithms used only the n -step return), weighted geometrically as in Figure 12.2. In the state-value case, this family of algorithms is known as Truncated TD(λ), or TTD(λ). The compound backup diagram, shown in Figure 12.7, is similar to that for TD(λ) (Figure 12.1) except that the longest component update is at most n steps rather than always going all the way to the

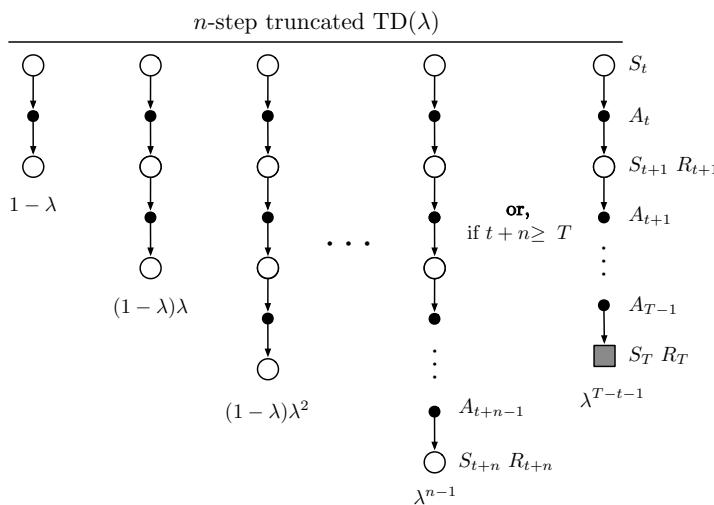


Figure 12.7: The backup diagram for Truncated TD(λ).

end of the episode. $\text{TTD}(\lambda)$ is defined by (cf. (9.15)):

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha [G_{t:t+n}^\lambda - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t < T.$$

This algorithm can be implemented efficiently so that per-step computation does not scale with n (though of course memory must). Much as in n -step TD methods, no updates are made on the first $n - 1$ time steps of each episode, and $n - 1$ additional updates are made upon termination. Efficient implementation relies on the fact that the k -step λ -return can be written exactly as

$$G_{t:t+k}^\lambda = \hat{v}(S_t, \mathbf{w}_{t-1}) + \sum_{i=t}^{t+k-1} (\gamma \lambda)^{i-t} \delta'_i, \quad (12.10)$$

where

$$\delta'_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_{t-1}).$$

Exercise 12.5 Several times in this book (often in exercises) we have established that returns can be written as sums of TD errors if the value function is held constant. Why is (12.10) another instance of this? Prove (12.10). \square

12.4 Redoing Updates: Online λ -return Algorithm

Choosing the truncation parameter n in Truncated TD(λ) involves a tradeoff. n should be large so that the method closely approximates the off-line λ -return algorithm, but it should also be small so that the updates can be made sooner and can influence behavior sooner. Can we get the best of both? Well, yes, in principle we can, albeit at the cost of computational complexity.

The idea is that, on each time step as you gather a new increment of data, you go back and redo all the updates since the beginning of the current episode. The new updates will be better than the ones you previously made because now they can take into account the time step's new data. That is, the updates are always towards an n -step truncated λ -return target, but they always use the latest horizon. In each pass over that episode you can use a slightly longer horizon and obtain slightly better results. Recall that the truncated λ -return is defined in (12.9) as

$$G_{t:h}^\lambda \doteq (1 - \lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}.$$

Let us step through how this target could ideally be used if computational complexity was not an issue. The episode begins with an estimate at time 0 using the weights \mathbf{w}_0 from the end of the previous episode. Learning begins when the data horizon is extended to time step 1. The target for the estimate at step 0, given the data up to horizon 1, could only be the one-step return $G_{0:1}$, which includes R_1 and bootstraps from the estimate $\hat{v}(S_1, \mathbf{w}_0)$. Note that this is exactly what $G_{0:1}^\lambda$ is, with the sum in the first term of the

equation degenerating to zero. Using this update target, we construct \mathbf{w}_1 . Then, after advancing the data horizon to step 2, what do we do? We have new data in the form of R_2 and S_2 , as well as the new \mathbf{w}_1 , so now we can construct a better update target $G_{0:2}^\lambda$ for the first update from S_0 as well as a better update target $G_{1:2}^\lambda$ for the second update from S_1 . Using these improved targets, we redo the updates at S_1 and S_2 , starting again from \mathbf{w}_0 , to produce \mathbf{w}_2 . Now we advance the horizon to step 3 and repeat, going all the way back to produce three new targets, redoing all updates starting from the original \mathbf{w}_0 to produce \mathbf{w}_3 , and so on. Each time the horizon is advanced, all the updates are redone starting from \mathbf{w}_0 using the weight vector from the preceding horizon.

This conceptual algorithm involves multiple passes over the episode, one at each horizon, each generating a different sequence of weight vectors. To describe it clearly we have to distinguish between the weight vectors computed at the different horizons. Let us use \mathbf{w}_t^h to denote the weights used to generate the value at time t in the sequence up to horizon h . The first weight vector \mathbf{w}_0^h in each sequence is that inherited from the previous episode (so they are the same for all h), and the last weight vector \mathbf{w}_h^h in each sequence defines the ultimate weight-vector sequence of the algorithm. At the final horizon $h = T$ we obtain the final weights \mathbf{w}_T^T which will be passed on to form the initial weights of the next episode. With these conventions, the three first sequences described in the previous paragraph can be given explicitly:

$$h = 1 : \quad \mathbf{w}_1^1 \doteq \mathbf{w}_0^1 + \alpha [G_{0:1}^\lambda - \hat{v}(S_0, \mathbf{w}_0^1)] \nabla \hat{v}(S_0, \mathbf{w}_0^1),$$

$$\begin{aligned} h = 2 : \quad \mathbf{w}_1^2 &\doteq \mathbf{w}_0^2 + \alpha [G_{0:2}^\lambda - \hat{v}(S_0, \mathbf{w}_0^2)] \nabla \hat{v}(S_0, \mathbf{w}_0^2), \\ &\mathbf{w}_2^2 \doteq \mathbf{w}_1^2 + \alpha [G_{1:2}^\lambda - \hat{v}(S_1, \mathbf{w}_1^2)] \nabla \hat{v}(S_1, \mathbf{w}_1^2), \end{aligned}$$

$$\begin{aligned} h = 3 : \quad \mathbf{w}_1^3 &\doteq \mathbf{w}_0^3 + \alpha [G_{0:3}^\lambda - \hat{v}(S_0, \mathbf{w}_0^3)] \nabla \hat{v}(S_0, \mathbf{w}_0^3), \\ &\mathbf{w}_2^3 \doteq \mathbf{w}_1^3 + \alpha [G_{1:3}^\lambda - \hat{v}(S_1, \mathbf{w}_1^3)] \nabla \hat{v}(S_1, \mathbf{w}_1^3), \\ &\mathbf{w}_3^3 \doteq \mathbf{w}_2^3 + \alpha [G_{2:3}^\lambda - \hat{v}(S_2, \mathbf{w}_2^3)] \nabla \hat{v}(S_2, \mathbf{w}_2^3). \end{aligned}$$

The general form for the update is

$$\mathbf{w}_{t+1}^h \doteq \mathbf{w}_t^h + \alpha [G_{t:h}^\lambda - \hat{v}(S_t, \mathbf{w}_t^h)] \nabla \hat{v}(S_t, \mathbf{w}_t^h), \quad 0 \leq t < h \leq T.$$

This update, together with $\mathbf{w}_t \doteq \mathbf{w}_t^t$ defines the *online λ -return algorithm*.

The online λ -return algorithm is fully online, determining a new weight vector \mathbf{w}_t at each step t during an episode, using only information available at time t . Its main drawback is that it is computationally complex, passing over the portion of the episode experienced so far on every step. Note that it is strictly more complex than the off-line λ -return algorithm, which passes through all the steps at the time of termination but does not make any updates during the episode. In return, the online algorithm can be expected to perform better than the off-line one, not only during the episode when it makes an update while the off-line algorithm makes none, but also at the end of the episode because the weight vector used in bootstrapping (in $G_{t:h}^\lambda$) has had a larger number of informative

updates. This effect can be seen if one looks carefully at Figure 12.8, which compares the two algorithms on the 19-state random walk task.

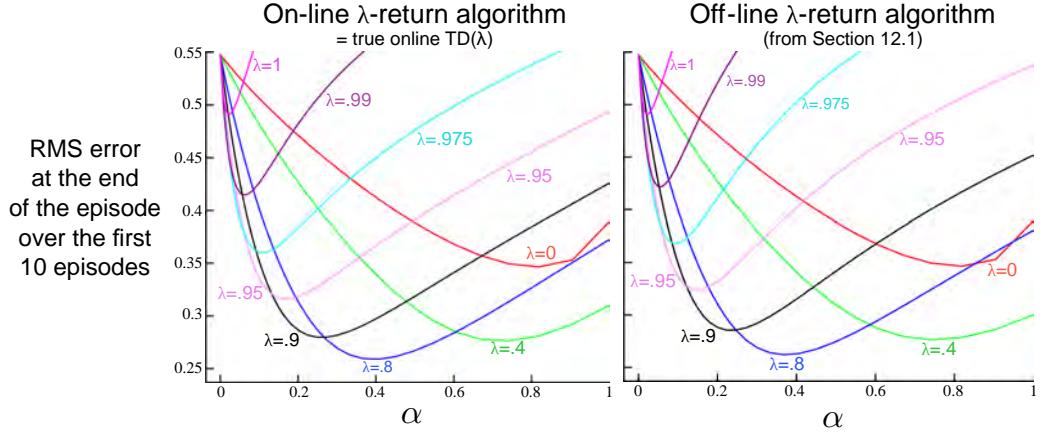


Figure 12.8: 19-state Random walk results (Example 7.1): Performance of online and off-line λ -return algorithms. The performance measure here is the VE at the end of the episode, which should be the best case for the off-line algorithm. Nevertheless, the online algorithm performs subtly better. For comparison, the $\lambda=0$ line is the same for both methods.

12.5 True Online TD(λ)

The online λ -return algorithm just presented is currently the best performing temporal-difference algorithm. It is an ideal which online TD(λ) only approximates. As presented, however, the online λ -return algorithm is very complex. Is there a way to invert this forward-view algorithm to produce an efficient backward-view algorithm using eligibility traces? It turns out that there is indeed an exact computationally congenial implementation of the online λ -return algorithm for the case of linear function approximation. This implementation is known as the true online TD(λ) algorithm because it is “truer” to the ideal of the online λ -return algorithm than the TD(λ) algorithm is.

The derivation of true online TD(λ) is a little too complex to present here (see the next section and the appendix to the paper by van Seijen et al., 2016) but its strategy is simple. The sequence of weight vectors produced by the online λ -return algorithm can be arranged in a triangle:

$$\begin{array}{ccccccc}
 \mathbf{w}_0^0 & & & & & & \\
 \mathbf{w}_0^1 & \mathbf{w}_1^1 & & & & & \\
 \mathbf{w}_0^2 & \mathbf{w}_1^2 & \mathbf{w}_2^2 & & & & \\
 \mathbf{w}_0^3 & \mathbf{w}_1^3 & \mathbf{w}_2^3 & \mathbf{w}_3^3 & & & \\
 \vdots & \vdots & \vdots & \vdots & \ddots & & \\
 \mathbf{w}_0^T & \mathbf{w}_1^T & \mathbf{w}_2^T & \mathbf{w}_3^T & \cdots & \mathbf{w}_T^T &
 \end{array}$$

One row of this triangle is produced on each time step. It turns out that the weight vectors on the diagonal, the \mathbf{w}_t^t , are the only ones really needed. The first, \mathbf{w}_0^0 , is the initial weight

vector of the episode, the last, \mathbf{w}_T^T , is the final weight vector, and each weight vector along the way, \mathbf{w}_t^t , plays a role in bootstrapping in the n -step returns of the updates. In the final algorithm the diagonal weight vectors are renamed without a superscript, $\mathbf{w}_t \doteq \mathbf{w}_t^t$. The strategy then is to find a compact, efficient way of computing each \mathbf{w}_t^t from the one before. If this is done, for the linear case in which $\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s)$, then we arrive at the true online TD(λ) algorithm:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t + \alpha (\mathbf{w}_t^\top \mathbf{x}_t - \mathbf{w}_{t-1}^\top \mathbf{x}_t) (\mathbf{z}_t - \mathbf{x}_t),$$

where we have used the shorthand $\mathbf{x}_t \doteq \mathbf{x}(S_t)$, δ_t is defined as in TD(λ) (12.6), and \mathbf{z}_t is defined by

$$\mathbf{z}_t \doteq \gamma \lambda \mathbf{z}_{t-1} + (1 - \alpha \gamma \lambda \mathbf{z}_{t-1}^\top \mathbf{x}_t) \mathbf{x}_t. \quad (12.11)$$

This algorithm has been proven to produce exactly the same sequence of weight vectors, $\mathbf{w}_t, 0 \leq t \leq T$, as the online λ -return algorithm (van Seijen et al. 2016). Thus the results on the random walk task on the left of Figure 12.8 are also its results on that task. Now, however, the algorithm is much less expensive. The memory requirements of true online TD(λ) are identical to those of conventional TD(λ), while the per-step computation is increased by about 50% (there is one more inner product in the eligibility-trace update). Overall, the per-step computational complexity remains of $O(d)$, the same as TD(λ). Pseudocode for the complete algorithm is given in the box.

True online TD(λ) for estimating $\mathbf{w}^\top \mathbf{x} \approx v_\pi$

Input: the policy π to be evaluated

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \rightarrow \mathbb{R}^d$ such that $\mathbf{x}(\text{terminal}, \cdot) = \mathbf{0}$

Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:

 Initialize state and obtain initial feature vector \mathbf{x}

$\mathbf{z} \leftarrow \mathbf{0}$	(a d -dimensional vector)
$V_{old} \leftarrow 0$	(a temporary scalar variable)

 Loop for each step of episode:

Choose $A \sim \pi$	
Take action A , observe R , \mathbf{x}' (feature vector of the next state)	
$V \leftarrow \mathbf{w}^\top \mathbf{x}$	
$V' \leftarrow \mathbf{w}^\top \mathbf{x}'$	
$\delta \leftarrow R + \gamma V' - V$	
$\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + (1 - \alpha \gamma \lambda \mathbf{z}^\top \mathbf{x}) \mathbf{x}$	
$\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + V - V_{old})\mathbf{z} - \alpha(V - V_{old})\mathbf{x}$	
$V_{old} \leftarrow V'$	
$\mathbf{x} \leftarrow \mathbf{x}'$	

 until $\mathbf{x}' = \mathbf{0}$ (signaling arrival at a terminal state)

The eligibility trace (12.11) used in true online TD(λ) is called a *dutch trace* to distinguish it from the trace (12.5) used in TD(λ), which is called an *accumulating trace*.

Earlier work often used a third kind of trace called the *replacing trace*, defined only for the tabular case or for binary feature vectors such as those produced by tile coding. The replacing trace is defined on a component-by-component basis depending on whether the component of the feature vector was 1 or 0:

$$z_{i,t} \doteq \begin{cases} 1 & \text{if } x_{i,t} = 1 \\ \gamma \lambda z_{i,t-1} & \text{otherwise.} \end{cases} \quad (12.12)$$

Nowadays, we see replacing traces as crude approximations to dutch traces, which largely supersede them. Dutch traces usually perform better than replacing traces and have a clearer theoretical basis. Accumulating traces remain of interest for nonlinear function approximations where dutch traces are not available.

12.6 *Dutch Traces in Monte Carlo Learning

Although eligibility traces are closely associated historically with TD learning, in fact they have nothing to do with it. In fact, eligibility traces arise even in Monte Carlo learning, as we show in this section. We show that the linear MC algorithm (Chapter 9), taken as a forward view, can be used to derive an equivalent yet computationally cheaper backward-view algorithm using dutch traces. This is the only equivalence of forward- and backward-views that we explicitly demonstrate in this book. It gives some of the flavor of the proof of equivalence of true online TD(λ) and the online λ -return algorithm, but is much simpler.

The linear version of the gradient Monte Carlo prediction algorithm (page 202) makes the following sequence of updates, one for each time step of the episode:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [G - \mathbf{w}_t^\top \mathbf{x}_t] \mathbf{x}_t, \quad 0 \leq t < T. \quad (12.13)$$

To simplify the example, we assume here that the return G is a single reward received at the end of the episode (this is why G is not subscripted by time) and that there is no discounting. In this case the update is also known as the Least Mean Square (LMS) rule. As a Monte Carlo algorithm, all the updates depend on the final reward/return, so none can be made until the end of the episode. The MC algorithm is an off-line algorithm and we do not seek to improve this aspect of it. Rather we seek merely an implementation of this algorithm with computational advantages. We will still update the weight vector only at the end of the episode, but we will do some computation during each step of the episode and less at its end. This will give a more equal distribution of computation— $O(d)$ per step—and also remove the need to store the feature vectors at each step for use later at the end of each episode. Instead, we will introduce an additional vector memory, the eligibility trace, keeping in it a summary of all the feature vectors seen so far. This will be sufficient to efficiently recreate exactly the same overall update as the sequence of MC

updates (12.13), by the end of the episode:

$$\begin{aligned}\mathbf{w}_T &= \mathbf{w}_{T-1} + \alpha (G - \mathbf{w}_{T-1}^\top \mathbf{x}_{T-1}) \mathbf{x}_{T-1} \\ &= \mathbf{w}_{T-1} + \alpha \mathbf{x}_{T-1} (-\mathbf{x}_{T-1}^\top \mathbf{w}_{T-1}) + \alpha G \mathbf{x}_{T-1} \\ &= (\mathbf{I} - \alpha \mathbf{x}_{T-1} \mathbf{x}_{T-1}^\top) \mathbf{w}_{T-1} + \alpha G \mathbf{x}_{T-1} \\ &= \mathbf{F}_{T-1} \mathbf{w}_{T-1} + \alpha G \mathbf{x}_{T-1}\end{aligned}$$

where $\mathbf{F}_t \doteq \mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^\top$ is a *forgetting*, or *fading*, matrix. Now, recursing,

$$\begin{aligned}&= \mathbf{F}_{T-1} (\mathbf{F}_{T-2} \mathbf{w}_{T-2} + \alpha G \mathbf{x}_{T-2}) + \alpha G \mathbf{x}_{T-1} \\ &= \mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{w}_{T-2} + \alpha G (\mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}) \\ &= \mathbf{F}_{T-1} \mathbf{F}_{T-2} (\mathbf{F}_{T-3} \mathbf{w}_{T-3} + \alpha G \mathbf{x}_{T-3}) + \alpha G (\mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}) \\ &= \mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{F}_{T-3} \mathbf{w}_{T-3} + \alpha G (\mathbf{F}_{T-1} \mathbf{F}_{T-2} \mathbf{x}_{T-3} + \mathbf{F}_{T-1} \mathbf{x}_{T-2} + \mathbf{x}_{T-1}) \\ &\quad \vdots \\ &= \underbrace{\mathbf{F}_{T-1} \mathbf{F}_{T-2} \cdots \mathbf{F}_0 \mathbf{w}_0}_{\mathbf{a}_{T-1}} + \underbrace{\alpha G \sum_{k=0}^{T-1} \mathbf{F}_{T-1} \mathbf{F}_{T-2} \cdots \mathbf{F}_{k+1} \mathbf{x}_k}_{\mathbf{z}_{T-1}} \\ &= \mathbf{a}_{T-1} + \alpha G \mathbf{z}_{T-1},\end{aligned}\tag{12.14}$$

where \mathbf{a}_{T-1} and \mathbf{z}_{T-1} are the values at time $T-1$ of two auxiliary memory vectors that can be updated incrementally without knowledge of G and with $O(d)$ complexity per time step. The \mathbf{z}_t vector is in fact a dutch-style eligibility trace. It is initialized to $\mathbf{z}_0 = \mathbf{x}_0$ and then updated according to

$$\begin{aligned}\mathbf{z}_t &\doteq \sum_{k=0}^t \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_{k+1} \mathbf{x}_k, \quad 1 \leq t < T \\ &= \sum_{k=0}^{t-1} \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_{k+1} \mathbf{x}_k + \mathbf{x}_t \\ &= \mathbf{F}_t \sum_{k=0}^{t-1} \mathbf{F}_{t-1} \mathbf{F}_{t-2} \cdots \mathbf{F}_{k+1} \mathbf{x}_k + \mathbf{x}_t \\ &= \mathbf{F}_t \mathbf{z}_{t-1} + \mathbf{x}_t \\ &= (\mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^\top) \mathbf{z}_{t-1} + \mathbf{x}_t \\ &= \mathbf{z}_{t-1} - \alpha \mathbf{x}_t \mathbf{x}_t^\top \mathbf{z}_{t-1} + \mathbf{x}_t \\ &= \mathbf{z}_{t-1} - \alpha (\mathbf{z}_{t-1}^\top \mathbf{x}_t) \mathbf{x}_t + \mathbf{x}_t \\ &= \mathbf{z}_{t-1} + (1 - \alpha \mathbf{z}_{t-1}^\top \mathbf{x}_t) \mathbf{x}_t,\end{aligned}$$

which is the dutch trace for the case of $\gamma \lambda = 1$ (cf. Eq. 12.11). The \mathbf{a}_t auxiliary vector is initialized to $\mathbf{a}_0 = \mathbf{w}_0$ and then updated according to

$$\mathbf{a}_t \doteq \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_0 \mathbf{w}_0 = \mathbf{F}_t \mathbf{a}_{t-1} = \mathbf{a}_{t-1} - \alpha \mathbf{x}_t \mathbf{x}_t^\top \mathbf{a}_{t-1}, \quad 1 \leq t < T.$$

The auxiliary vectors, \mathbf{a}_t and \mathbf{z}_t , are updated on each time step $t < T$ and then, at time T when G is observed, they are used in (12.14) to compute \mathbf{w}_T . In this way we achieve exactly the same final result as the MC/LMS algorithm that has poor computational properties (12.13), but now with an incremental algorithm whose time and memory complexity per step is $O(d)$. This is surprising and intriguing because the notion of an eligibility trace (and the dutch trace in particular) has arisen in a setting without temporal-difference (TD) learning (in contrast to van Seijen and Sutton, 2014). It seems eligibility traces are not specific to TD learning at all; they are more fundamental than that. The need for eligibility traces seems to arise whenever one tries to learn long-term predictions in an efficient manner.

12.7 Sarsa(λ)

Very few changes in the ideas already presented in this chapter are required in order to extend eligibility-traces to action-value methods. To learn approximate action values, $\hat{q}(s, a, \mathbf{w})$, rather than approximate state values, $\hat{v}(s, \mathbf{w})$, we need to use the action-value form of the n -step return, from Chapter 10:

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad t+n < T,$$

with $G_{t:t+n} \doteq G_t$ if $t+n \geq T$. Using this, we can form the action-value form of the λ -return, which is otherwise identical to the state-value form (12.3). The action-value form of the off-line λ -return algorithm (12.4) simply uses \hat{q} rather than \hat{v} :

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[G_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad t = 0, \dots, T-1, \quad (12.15)$$

where $G_t^\lambda \doteq G_{t:\infty}^\lambda$. The compound backup diagram for this forward view is shown in Figure 12.9. Notice the similarity to the diagram of the TD(λ) algorithm (Figure 12.1). The first update looks ahead one full step, to the next state-action pair, the second looks ahead two steps, to the second state-action pair, and so on. A final update is based on the complete return. The weighting of each n -step update in the λ -return is just as in TD(λ) and the λ -return algorithm (12.3).

The temporal-difference method for action values, known as *Sarsa*(λ), approximates this forward view. It has the same update rule as given earlier for TD(λ):

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t,$$

except, naturally, using the action-value form of the TD error:

$$\delta_t \doteq R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \quad (12.16)$$

and the action-value form of the eligibility trace:

$$\begin{aligned} \mathbf{z}_{-1} &\doteq \mathbf{0}, \\ \mathbf{z}_t &\doteq \gamma \lambda \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad 0 \leq t \leq T. \end{aligned}$$

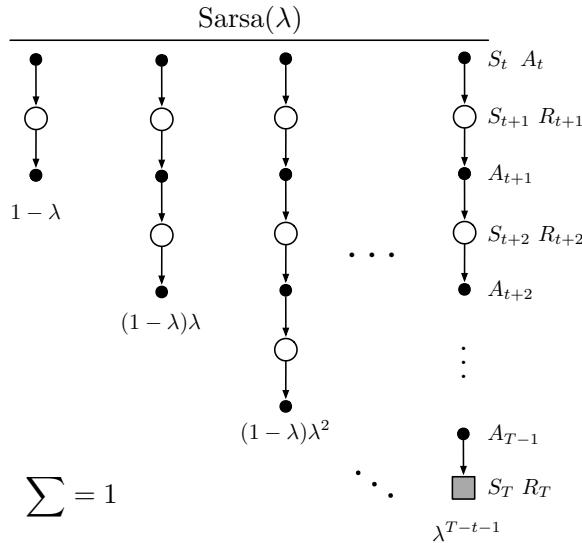
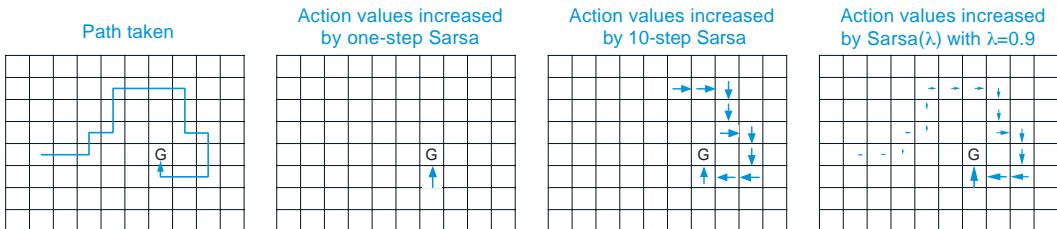


Figure 12.9: Sarsa(λ)'s backup diagram. Compare with Figure 12.1.

Complete pseudocode for Sarsa(λ) with linear function approximation, binary features, and either accumulating or replacing traces is given in the box on the next page. This pseudocode highlights a few optimizations possible in the special case of binary features (features are either active (=1) or inactive (=0)).

Example 12.1: Traces in Gridworld The use of eligibility traces can substantially increase the efficiency of control algorithms over one-step methods and even over n -step methods. The reason for this is illustrated by the gridworld example below.



The first panel shows the path taken by an agent in a single episode. The initial estimated values were zero, and all rewards were zero except for a positive reward at the goal location marked by G. The arrows in the other panels show, for various algorithms, which action-values would be increased, and by how much, upon reaching the goal. A one-step method would increment only the last action value, whereas an n -step method would equally increment the last n actions' values (assuming $\gamma = 1$), and an eligibility trace method would update all the action values up to the beginning of the episode, to different degrees, fading with recency. The fading strategy is often the best. ■

**Sarsa(λ) with binary features and linear function approximation
for estimating $\mathbf{w}^\top \mathbf{x} \approx q_\pi$ or q_***

Input: a function $\mathcal{F}(s, a)$ returning the set of (indices of) active features for s, a
 Input: a policy π
 Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$, small $\varepsilon > 0$
 Initialize: $\mathbf{w} = (w_1, \dots, w_d)^\top \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$), $\mathbf{z} = (z_1, \dots, z_d)^\top \in \mathbb{R}^d$

Loop for each episode:

- Initialize S
- Choose $A \sim \pi(\cdot | S)$ or ε -greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
- $\mathbf{z} \leftarrow \mathbf{0}$
- Loop for each step of episode:
 - Take action A , observe R, S'
 - $\delta \leftarrow R$
 - Loop for i in $\mathcal{F}(S, A)$:
 - $\delta \leftarrow \delta - w_i$
 - $z_i \leftarrow z_i + 1$ (accumulating traces)
 - or $z_i \leftarrow 1$ (replacing traces)
 - If S' is terminal then:
 - $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$
 - Go to next episode
 - Choose $A' \sim \pi(\cdot | S')$ or ε -greedy according to $\hat{q}(S', \cdot, \mathbf{w})$
 - Loop for i in $\mathcal{F}(S', A')$: $\delta \leftarrow \delta + \gamma w_i$
 - $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$
 - $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z}$
 - $S \leftarrow S'; A \leftarrow A'$

Exercise 12.6 Modify the pseudocode for Sarsa(λ) to use dutch traces (12.11) without the other distinctive features of a true online algorithm. Assume linear function approximation and binary features. \square

Example 12.2: Sarsa(λ) on Mountain Car Figure 12.10 (left) on the next page shows results with Sarsa(λ) on the Mountain Car task introduced in Example 10.1. The function approximation, action selection, and environmental details were exactly as in Chapter 10, and thus it is appropriate to numerically compare these results with the Chapter 10 results for n -step Sarsa (right side of the figure). The earlier results varied the update length n whereas here for Sarsa(λ) we vary the trace parameter λ , which plays a similar role. The fading-trace bootstrapping strategy of Sarsa(λ) appears to result in more efficient learning on this problem. \blacksquare

There is also an action-value version of our ideal TD method, the online λ -return algorithm (Section 12.4) and its efficient implementation as true online TD(λ) (Section 12.5). Everything in Section 12.4 goes through without change other than to use the action-value form of the n -step return given at the beginning of the current section. The analyses in Sections 12.5 and 12.6 also carry through for action values, the only change being the use

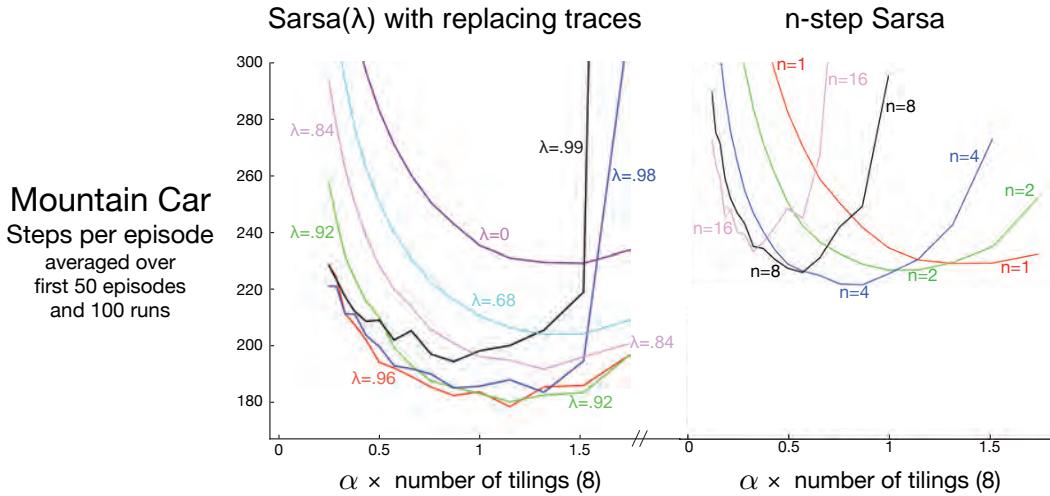


Figure 12.10: Early performance on the Mountain Car task of Sarsa(λ) with replacing traces and n -step Sarsa (copied from Figure 10.4) as a function of the step size, α .

of state-action feature vectors $\mathbf{x}_t = \mathbf{x}(S_t, A_t)$ instead of state feature vectors $\mathbf{x}_t = \mathbf{x}(S_t)$. Pseudocode for the resulting efficient algorithm, called *true online Sarsa(λ)* is given in the box on the next page. The figure below compares the performance of various versions of Sarsa(λ) on the Mountain Car example.

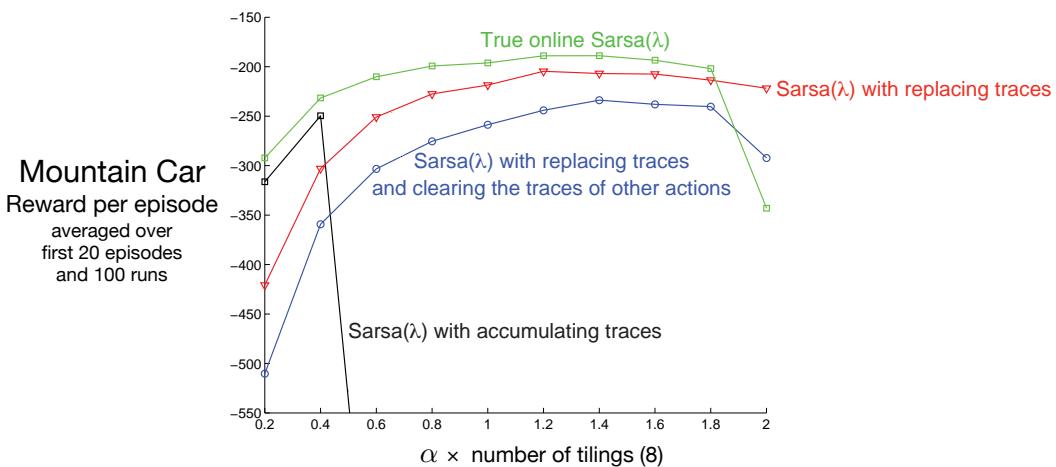


Figure 12.11: Summary comparison of Sarsa(λ) algorithms on the Mountain Car task. True online Sarsa(λ) performed better than regular Sarsa(λ) with both accumulating and replacing traces. Also included is a version of Sarsa(λ) with replacing traces in which, on each time step, the traces for the state and the actions not selected were set to zero.

True online Sarsa(λ) for estimating $\mathbf{w}^\top \mathbf{x} \approx q_\pi$ or q_*

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \rightarrow \mathbb{R}^d$ such that $\mathbf{x}(\text{terminal}, \cdot) = \mathbf{0}$
 Input: a policy π (if estimating q_π)
 Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$, small $\varepsilon > 0$
 Initialize: $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
 Initialize S
 Choose $A \sim \pi(\cdot | S)$ or ε -greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
 $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
 $\mathbf{z} \leftarrow \mathbf{0}$
 $Q_{old} \leftarrow 0$
 Loop for each step of episode:
 Take action A , observe R, S'
 Choose $A' \sim \pi(\cdot | S')$ or ε -greedy according to $\hat{q}(S', \cdot, \mathbf{w})$
 $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
 $Q \leftarrow \mathbf{w}^\top \mathbf{x}$
 $Q' \leftarrow \mathbf{w}^\top \mathbf{x}'$
 $\delta \leftarrow R + \gamma Q' - Q$
 $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + (1 - \alpha \gamma \lambda \mathbf{z}^\top \mathbf{x}) \mathbf{x}$
 $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
 $Q_{old} \leftarrow Q'$
 $\mathbf{x} \leftarrow \mathbf{x}'$
 $A \leftarrow A'$
 until S' is terminal

Finally, there is also a truncated version of Sarsa(λ), called *forward Sarsa(λ)* (van Seijen, 2016), which appears to be a particularly effective model-free control method for use in conjunction with multi-layer artificial neural networks.

12.8 Variable λ and γ

We are starting now to reach the end of our development of fundamental TD learning algorithms. To present the final algorithms in their most general forms, it is useful to generalize the degree of bootstrapping and discounting beyond constant parameters to functions potentially dependent on the state and action. That is, each time step will have a different λ and γ , denoted λ_t and γ_t . We change notation now so that $\lambda : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is now a function from states and actions to the unit interval such that $\lambda_t \doteq \lambda(S_t, A_t)$, and similarly, $\gamma : \mathcal{S} \rightarrow [0, 1]$ is a function from states to the unit interval such that $\gamma_t \doteq \gamma(S_t)$.

Introducing the function γ , the *termination function*, is particularly significant because it changes the return, the fundamental random variable whose expectation we seek to

estimate. Now the return is defined more generally as

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma_{t+1} G_{t+1} \\ &= R_{t+1} + \gamma_{t+1} R_{t+2} + \gamma_{t+1} \gamma_{t+2} R_{t+3} + \gamma_{t+1} \gamma_{t+2} \gamma_{t+3} R_{t+4} + \dots \\ &= \sum_{k=t}^{\infty} \left(\prod_{i=t+1}^k \gamma_i \right) R_{k+1}, \end{aligned} \quad (12.17)$$

where, to assure the sums are finite, we require that $\prod_{k=t}^{\infty} \gamma_k = 0$ with probability one for all t . One convenient aspect of this definition is that it enables the episodic setting and its algorithms to be presented in terms of a single stream of experience, without special terminal states, start distributions, or termination times. An erstwhile terminal state becomes a state at which $\gamma(s) = 0$ and which transitions to the start distribution. In that way (and by choosing $\gamma(\cdot)$ as a constant in all other states) we can recover the classical episodic setting as a special case. State dependent termination includes other prediction cases such as *pseudo termination*, in which we seek to predict a quantity without altering the flow of the Markov process. Discounted returns can be thought of as such a quantity, in which case state-dependent termination unifies the episodic and discounted-continuing cases. (The undiscounted-continuing case still needs some special treatment.)

The generalization to variable bootstrapping is not a change in the problem, like discounting, but a change in the solution strategy. The generalization affects the λ -returns for states and actions. The new state-based λ -return can be written recursively as

$$G_t^{\lambda s} \doteq R_{t+1} + \gamma_{t+1} ((1 - \lambda_{t+1}) \hat{v}(S_{t+1}, \mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda s}), \quad (12.18)$$

where now we have added the “ s ” to the superscript λ to remind us that this is a return that bootstraps from state values, distinguishing it from returns that bootstrap from action values, which we present below with “ a ” in the superscript. This equation says that the λ -return is the first reward, undiscounted and unaffected by bootstrapping, plus possibly a second term to the extent that we are not discounting at the next state (that is, according to γ_{t+1} ; recall that this is zero if the next state is terminal). To the extent that we aren’t terminating at the next state, we have a second term which is itself divided into two cases depending on the degree of bootstrapping in the state. To the extent we are bootstrapping, this term is the estimated value at the state, whereas, to the extent that we are not bootstrapping, the term is the λ -return for the next time step. The action-based λ -return is either the Sarsa form

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1} ((1 - \lambda_{t+1}) \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda a}), \quad (12.19)$$

or the Expected Sarsa form,

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1} ((1 - \lambda_{t+1}) \bar{V}_t(S_{t+1}) + \lambda_{t+1} G_{t+1}^{\lambda a}), \quad (12.20)$$

where (7.8) is generalized to function approximation as

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) \hat{q}(s, a, \mathbf{w}_t). \quad (12.21)$$

Exercise 12.7 Generalize the three recursive equations above to their truncated versions, defining $G_{t:h}^{\lambda s}$ and $G_{t:h}^{\lambda a}$. \square

12.9 Off-policy Traces with Control Variates

The final step is to incorporate importance sampling. For methods using non-truncated λ -returns, there is not a practical option in which the importance-sampling weighting is applied to the target return (as there is for n -step methods as explained in Section 7.3). Instead, we move directly to the bootstrapping generalization of per-decision importance sampling with control variates (Section 7.4).

In the state case, our final definition of the λ -return generalizes (12.18), after the model of (7.13), to

$$G_t^{\lambda s} \doteq \rho_t \left(R_{t+1} + \gamma_{t+1} ((1 - \lambda_{t+1}) \hat{v}(S_{t+1}, \mathbf{w}_t) + \lambda_{t+1} G_{t+1}^{\lambda s}) \right) + (1 - \rho_t) \hat{v}(S_t, \mathbf{w}_t), \quad (12.22)$$

where $\rho_t = \frac{\pi(A_t | S_t)}{b(A_t | S_t)}$ is the usual single-step importance sampling ratio. Much like the other returns we have seen in this book, this final λ -return can be approximated simply in terms of sums of the state-based TD error,

$$\delta_t^s \doteq R_{t+1} + \gamma_{t+1} \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \quad (12.23)$$

as

$$G_t^{\lambda s} \approx \hat{v}(S_t, \mathbf{w}_t) + \rho_t \sum_{k=t}^{\infty} \delta_k^s \prod_{i=t+1}^k \gamma_i \lambda_i \rho_i, \quad (12.24)$$

with the approximation becoming exact if the approximate value function does not change.

Exercise 12.8 Prove that (12.24) becomes exact if the value function does not change. To save writing, consider the case of $t = 0$, and use the notation $V_k \doteq \hat{v}(S_k, \mathbf{w})$. \square

Exercise 12.9 The truncated version of the general off-policy return is denoted $G_{t:h}^{\lambda s}$. Guess the correct equation, based on (12.24). \square

The above form of the λ -return (12.24) is convenient to use in a forward-view update,

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha (G_t^{\lambda s} - \hat{v}(S_t, \mathbf{w}_t)) \nabla \hat{v}(S_t, \mathbf{w}_t) \\ &\approx \mathbf{w}_t + \alpha \rho_t \left(\sum_{k=t}^{\infty} \delta_k^s \prod_{i=t+1}^k \gamma_i \lambda_i \rho_i \right) \nabla \hat{v}(S_t, \mathbf{w}_t), \end{aligned}$$

which to the experienced eye looks like an eligibility-based TD update—the product is like an eligibility trace and it is multiplied by TD errors. But this is just one time step of a forward view. The relationship that we are looking for is that the forward-view update, summed over time, is approximately equal to a backward-view update, summed over time (this relationship is only approximate because again we ignore changes in the value

function). The sum of the forward-view update over time is

$$\begin{aligned}
\sum_{t=0}^{\infty} (\mathbf{w}_{t+1} - \mathbf{w}_t) &\approx \sum_{t=0}^{\infty} \sum_{k=t}^{\infty} \alpha \rho_t \delta_k^s \nabla \hat{v}(S_t, \mathbf{w}_t) \prod_{i=t+1}^k \gamma_i \lambda_i \rho_i \\
&= \sum_{k=0}^{\infty} \sum_{t=0}^k \alpha \rho_t \nabla \hat{v}(S_t, \mathbf{w}_t) \delta_k^s \prod_{i=t+1}^k \gamma_i \lambda_i \rho_i \\
&\quad (\text{using the summation rule: } \sum_{t=x}^y \sum_{k=t}^y = \sum_{k=x}^y \sum_{t=x}^k) \\
&= \sum_{k=0}^{\infty} \alpha \delta_k^s \sum_{t=0}^k \rho_t \nabla \hat{v}(S_t, \mathbf{w}_t) \prod_{i=t+1}^k \gamma_i \lambda_i \rho_i,
\end{aligned}$$

which would be in the form of the sum of a backward-view TD update if the entire expression from the second sum on could be written and updated incrementally as an eligibility trace, which we now show can be done. That is, we show that if this expression was the trace at time k , then we could update it from its value at time $k-1$ by:

$$\begin{aligned}
\mathbf{z}_k &= \sum_{t=0}^k \rho_t \nabla \hat{v}(S_t, \mathbf{w}_t) \prod_{i=t+1}^k \gamma_i \lambda_i \rho_i \\
&= \sum_{t=0}^{k-1} \rho_t \nabla \hat{v}(S_t, \mathbf{w}_t) \prod_{i=t+1}^k \gamma_i \lambda_i \rho_i + \rho_k \nabla \hat{v}(S_k, \mathbf{w}_k) \\
&= \gamma_k \lambda_k \rho_k \underbrace{\sum_{t=0}^{k-1} \rho_t \nabla \hat{v}(S_t, \mathbf{w}_t) \prod_{i=t+1}^{k-1} \gamma_i \lambda_i \rho_i}_{\mathbf{z}_{k-1}} + \rho_k \nabla \hat{v}(S_k, \mathbf{w}_k) \\
&= \rho_k (\gamma_k \lambda_k \mathbf{z}_{k-1} + \nabla \hat{v}(S_k, \mathbf{w}_k)),
\end{aligned}$$

which, changing the index from k to t , is the general accumulating trace update for state values:

$$\mathbf{z}_t \doteq \rho_t (\gamma_t \lambda_t \mathbf{z}_{t-1} + \nabla \hat{v}(S_t, \mathbf{w}_t)), \quad (12.25)$$

This eligibility trace, together with the usual semi-gradient parameter-update rule for $\text{TD}(\lambda)$ (12.7), forms a general $\text{TD}(\lambda)$ algorithm that can be applied to either on-policy or off-policy data. In the on-policy case, the algorithm is exactly $\text{TD}(\lambda)$ because ρ_t is always 1 and (12.25) becomes the usual accumulating trace (12.5) (extended to variable λ and γ). In the off-policy case, the algorithm often works well but, as a semi-gradient method, is not guaranteed to be stable. In the next few sections we will consider extensions of it that do guarantee stability.

A very similar series of steps can be followed to derive the off-policy eligibility traces for action-value methods and corresponding general Sarsa(λ) algorithms. One could start with either recursive form for the general action-based λ -return, (12.19) or (12.20), but the latter (the Expected Sarsa form) works out to be simpler. We extend (12.20) to the

off-policy case after the model of (7.14) to produce

$$\begin{aligned} G_t^{\lambda a} &\doteq R_{t+1} + \gamma_{t+1} \left((1 - \lambda_{t+1}) \bar{V}_t(S_{t+1}) + \lambda_{t+1} [\rho_{t+1} G_{t+1}^{\lambda a} + \bar{V}_t(S_{t+1}) - \rho_{t+1} \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)] \right) \\ &= R_{t+1} + \gamma_{t+1} \left(\bar{V}_t(S_{t+1}) + \lambda_{t+1} \rho_{t+1} [G_{t+1}^{\lambda a} - \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)] \right) \end{aligned} \quad (12.26)$$

where $\bar{V}_t(S_{t+1})$ is as given by (12.21). Again the λ -return can be written approximately as the sum of TD errors,

$$G_t^{\lambda a} \approx \hat{q}(S_t, A_t, \mathbf{w}_t) + \sum_{k=t}^{\infty} \delta_k^a \prod_{i=t+1}^k \gamma_i \lambda_i \rho_i, \quad (12.27)$$

using the expectation form of the action-based TD error:

$$\delta_t^a = R_{t+1} + \gamma_{t+1} \bar{V}_t(S_{t+1}) - \hat{q}(S_t, A_t, \mathbf{w}_t). \quad (12.28)$$

As before, the approximation becomes exact if the approximate value function does not change.

Exercise 12.10 Prove that (12.27) becomes exact if the value function does not change. To save writing, consider the case of $t = 0$, and use the notation $Q_k = \hat{q}(S_k, A_k, \mathbf{w})$. Hint: Start by writing out δ_0^a and $G_0^{\lambda a}$, then $G_0^{\lambda a} - Q_0$. \square

Exercise 12.11 The truncated version of the general off-policy return is denoted $G_{t:h}^{\lambda a}$. Guess the correct equation for it, based on (12.27). \square

Using steps entirely analogous to those for the state case, one can write a forward-view update based on (12.27), transform the sum of the updates using the summation rule, and finally derive the following form for the eligibility trace for action values:

$$\mathbf{z}_t \doteq \gamma_t \lambda_t \rho_t \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t). \quad (12.29)$$

This eligibility trace, together with the expectation-based TD error (12.28) and the usual semi-gradient parameter-update rule (12.7), forms an elegant, efficient Expected Sarsa(λ) algorithm that can be applied to either on-policy or off-policy data. It is probably the best algorithm of this type at the current time (though of course it is not guaranteed to be stable until combined in some way with one of the methods presented in the following sections). In the on-policy case with constant λ and γ , and the usual state-action TD error (12.16), the algorithm would be identical to the Sarsa(λ) algorithm presented in Section 12.7.

Exercise 12.12 Show in detail the steps outlined above for deriving (12.29) from (12.27). Start with the update (12.15), substitute $G_t^{\lambda a}$ from (12.26) for G_t^λ , then follow similar steps as led to (12.25). \square

At $\lambda = 1$, these algorithms become closely related to corresponding Monte Carlo algorithms. One might expect that an exact equivalence would hold for episodic problems and off-line updating, but in fact the relationship is subtler and slightly weaker than that. Under these most favorable conditions still there is not an episode by episode equivalence of updates, only of their expectations. This should not be surprising as these methods

make irrevocable updates as a trajectory unfolds, whereas true Monte Carlo methods would make no update for a trajectory if any action within it has zero probability under the target policy. In particular, all of these methods, even at $\lambda = 1$, still bootstrap in the sense that their targets depend on the current value estimates—it's just that the dependence cancels out in expected value. Whether this is a good or bad property in practice is another question. Recently, methods have been proposed that do achieve an exact equivalence (Sutton, Mahmood, Precup and van Hasselt, 2014). These methods require an additional vector of “provisional weights” that keep track of updates which have been made but may need to be retracted (or emphasized) depending on the actions taken later. The state and state-action versions of these methods are called $PTD(\lambda)$ and $PQ(\lambda)$ respectively, where the ‘P’ stands for Provisional.

The practical consequences of all these new off-policy methods have not yet been established. Undoubtedly, issues of high variance will arise as they do in all off-policy methods using importance sampling (Section 11.9).

If $\lambda < 1$, then all these off-policy algorithms involve bootstrapping and the deadly triad applies (Section 11.3), meaning that they can be guaranteed stable only for the tabular case, for state aggregation, and for other limited forms of function approximation. For linear and more-general forms of function approximation the parameter vector may diverge to infinity as in the examples in Chapter 11. As we discussed there, the challenge of off-policy learning has two parts. Off-policy eligibility traces deal effectively with the first part of the challenge, correcting for the expected value of the targets, but not at all with the second part of the challenge, having to do with the distribution of updates. Algorithmic strategies for meeting the second part of the challenge of off-policy learning with eligibility traces are summarized in Section 12.11.

Exercise 12.13 What are the dutch-trace and replacing-trace versions of off-policy eligibility traces for state-value and action-value methods? \square

12.10 Watkins’s $Q(\lambda)$ to Tree-Backup(λ)

Several methods have been proposed over the years to extend Q-learning to eligibility traces. The original is *Watkins’s $Q(\lambda)$* , which decays its eligibility traces in the usual way as long as a greedy action was taken, then cuts the traces to zero after the first non-greedy action. The backup diagram for Watkins’s $Q(\lambda)$ is shown in Figure 12.12. In Chapter 6, we unified Q-learning and Expected Sarsa in the off-policy version of the latter, which includes Q-learning as a special case, and generalizes it to arbitrary target policies, and in the previous section of this chapter we completed our treatment of Expected Sarsa by generalizing it to off-policy eligibility traces. In Chapter 7, however, we distinguished n -step Expected Sarsa from n -step Tree Backup, where the latter retained the property of not using importance sampling. It remains then to present the eligibility trace version of Tree Backup, which we will call *Tree-Backup(λ)*, or $TB(\lambda)$ for short. This is arguably the true successor to Q-learning because it retains its appealing absence of importance sampling even though it can be applied to off-policy data.

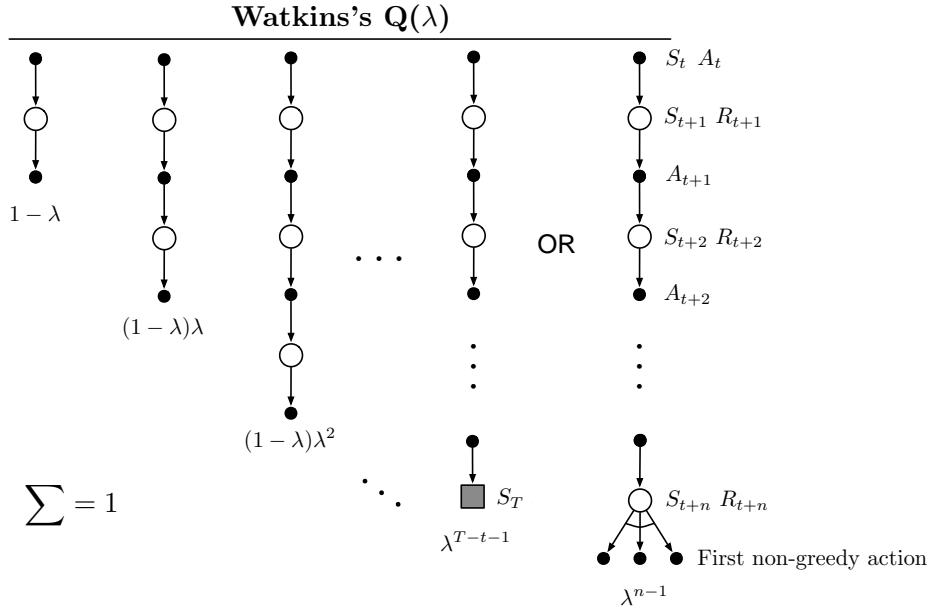


Figure 12.12: The backup diagram for Watkins's $Q(\lambda)$. The series of component updates ends either with the end of the episode or with the first nongreedy action, whichever comes first.

The concept of TB(λ) is straightforward. As shown in its backup diagram in Figure 12.13, the tree-backup updates of each length (from Section 7.5) are weighted in the usual way dependent on the bootstrapping parameter λ . To get the detailed equations, with the right indices on the general bootstrapping and discounting parameters, it is best to start with a recursive form (12.20) for the λ -return using action values, and then expand the bootstrapping case of the target after the model of (7.16):

$$\begin{aligned} G_t^{\lambda a} &\doteq R_{t+1} + \gamma_{t+1} \left((1 - \lambda_{t+1}) \bar{V}_t(S_{t+1}) + \lambda_{t+1} \left[\sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) \hat{q}(S_{t+1}, a, \mathbf{w}_t) + \pi(A_{t+1}|S_{t+1}) G_{t+1}^{\lambda a} \right] \right) \\ &= R_{t+1} + \gamma_{t+1} \left(\bar{V}_t(S_{t+1}) + \lambda_{t+1} \pi(A_{t+1}|S_{t+1}) \left(G_{t+1}^{\lambda a} - \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) \right) \right) \end{aligned}$$

As per the usual pattern, it can also be written approximately (ignoring changes in the approximate value function) as a sum of TD errors,

$$G_t^{\lambda a} \approx \hat{q}(S_t, A_t, \mathbf{w}_t) + \sum_{k=t}^{\infty} \delta_k^a \prod_{i=t+1}^k \gamma_i \lambda_i \pi(A_i|S_i),$$

using the expectation form of the action-based TD error (12.28).

Following the same steps as in the previous section, we arrive at a special eligibility trace update involving the target-policy probabilities of the selected actions,

$$\mathbf{z}_t \doteq \gamma_t \lambda_t \pi(A_t|S_t) \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t).$$

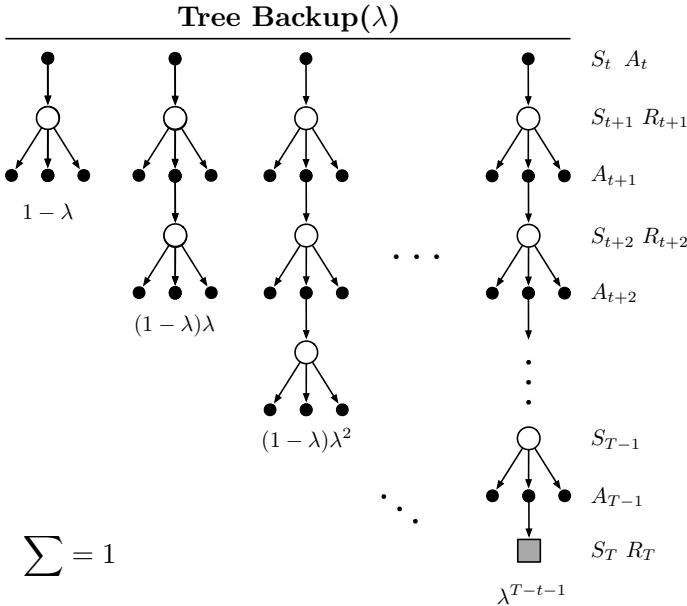


Figure 12.13: The backup diagram for the λ version of the Tree Backup algorithm.

This, together with the usual parameter-update rule (12.7), defines the $\text{TB}(\lambda)$ algorithm. Like all semi-gradient algorithms, $\text{TB}(\lambda)$ is not guaranteed to be stable when used with off-policy data and with a powerful function approximator. To obtain those assurances, $\text{TB}(\lambda)$ would have to be combined with one of the methods presented in the next section.

*Exercise 12.14 How might Double Expected Sarsa be extended to eligibility traces? □

12.11 Stable Off-policy Methods with Traces

Several methods using eligibility traces have been proposed that achieve guarantees of stability under off-policy training, and here we present four of the most important using this book's standard notation, including general bootstrapping and discounting functions. All are based on either the Gradient-TD or the Emphatic-TD ideas presented in Sections 11.7 and 11.8. All the algorithms assume linear function approximation, though extensions to nonlinear function approximation can also be found in the literature.

$GTD(\lambda)$ is the eligibility-trace algorithm analogous to TDC, the better of the two state-value Gradient-TD prediction algorithms discussed in Section 11.7. Its goal is to learn a parameter \mathbf{w}_t such that $\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}_t^\top \mathbf{x}(s) \approx v_\pi(s)$, even from data that is due to following another policy b . Its update is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t^s \mathbf{z}_t - \alpha \gamma_{t+1} (1 - \lambda_{t+1}) (\mathbf{z}_t^\top \mathbf{v}_t) \mathbf{x}_{t+1},$$

with δ_t^s , \mathbf{z}_t , and ρ_t defined in the usual ways for state values (12.23) (12.25) (11.1), and

$$\mathbf{v}_{t+1} \doteq \mathbf{v}_t + \beta \delta_t^s \mathbf{z}_t - \beta (\mathbf{v}_t^\top \mathbf{x}_t) \mathbf{x}_t, \quad (12.30)$$

where, as in Section 11.7, $\mathbf{v} \in \mathbb{R}^d$ is a vector of the same dimension as \mathbf{w} , initialized to $\mathbf{v}_0 = \mathbf{0}$, and $\beta > 0$ is a second step-size parameter.

$GQ(\lambda)$ is the Gradient-TD algorithm for action values with eligibility traces. Its goal is to learn a parameter \mathbf{w}_t such that $\hat{q}(s, a, \mathbf{w}_t) \doteq \mathbf{w}_t^\top \mathbf{x}(s, a) \approx q_\pi(s, a)$ from off-policy data. If the target policy is ε -greedy, or otherwise biased toward the greedy policy for \hat{q} , then $GQ(\lambda)$ can be used as a control algorithm. Its update is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t^a \mathbf{z}_t - \alpha \gamma_{t+1} (1 - \lambda_{t+1}) (\mathbf{z}_t^\top \mathbf{v}_t) \bar{\mathbf{x}}_{t+1},$$

where $\bar{\mathbf{x}}_t$ is the average feature vector for S_t under the target policy,

$$\bar{\mathbf{x}}_t \doteq \sum_a \pi(a|S_t) \mathbf{x}(S_t, a),$$

δ_t^a is the expectation form of the TD error, which can be written

$$\delta_t^a \doteq R_{t+1} + \gamma_{t+1} \mathbf{w}_t^\top \bar{\mathbf{x}}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t,$$

\mathbf{z}_t is defined in the usual way for action values (12.29), and the rest is as in $GTD(\lambda)$, including the update for \mathbf{v}_t (12.30).

$HTD(\lambda)$ is a hybrid state-value algorithm combining aspects of $GTD(\lambda)$ and $TD(\lambda)$. Its most appealing feature is that it is a strict generalization of $TD(\lambda)$ to off-policy learning, meaning that if the behavior policy happens to be the same as the target policy, then $HTD(\lambda)$ becomes the same as $TD(\lambda)$, which is not true for $GTD(\lambda)$. This is appealing because $TD(\lambda)$ is often faster than $GTD(\lambda)$ when both algorithms converge, and $TD(\lambda)$ requires setting only a single step size. $HTD(\lambda)$ is defined by

$$\begin{aligned} \mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha \delta_t^s \mathbf{z}_t + \alpha ((\mathbf{z}_t - \mathbf{z}_t^b)^\top \mathbf{v}_t) (\mathbf{x}_t - \gamma_{t+1} \mathbf{x}_{t+1}), \\ \mathbf{v}_{t+1} &\doteq \mathbf{v}_t + \beta \delta_t^s \mathbf{z}_t - \beta (\mathbf{z}_t^b)^\top \mathbf{v}_t (\mathbf{x}_t - \gamma_{t+1} \mathbf{x}_{t+1}), \quad \text{with } \mathbf{v}_0 \doteq \mathbf{0}, \\ \mathbf{z}_t &\doteq \rho_t (\gamma_t \lambda_t \mathbf{z}_{t-1} + \mathbf{x}_t), \quad \text{with } \mathbf{z}_{-1} \doteq \mathbf{0}, \\ \mathbf{z}_t^b &\doteq \gamma_t \lambda_t \mathbf{z}_{t-1}^b + \mathbf{x}_t, \quad \text{with } \mathbf{z}_{-1}^b \doteq \mathbf{0}, \end{aligned}$$

where $\beta > 0$ again is a second step-size parameter. In addition to the second set of weights, \mathbf{v}_t , $HTD(\lambda)$ also has a second set of eligibility traces, \mathbf{z}_t^b . These are conventional accumulating eligibility traces for the behavior policy and become equal to \mathbf{z}_t if all the ρ_t are 1, which causes the last term in the \mathbf{w}_t update to be zero and the overall update to reduce to $TD(\lambda)$.

Emphatic TD(λ) is the extension of the one-step Emphatic-TD algorithm (Sections 9.11 and 11.8) to eligibility traces. The resultant algorithm retains strong off-policy convergence guarantees while enabling any degree of bootstrapping, albeit at the cost of

high variance and potentially slow convergence. Emphatic TD(λ) is defined by

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t \\ \delta_t &\doteq R_{t+1} + \gamma_{t+1} \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t \\ \mathbf{z}_t &\doteq \rho_t (\gamma_t \lambda_t \mathbf{z}_{t-1} + M_t \mathbf{x}_t), \quad \text{with } \mathbf{z}_{-1} \doteq \mathbf{0}, \\ M_t &\doteq \lambda_t I_t + (1 - \lambda_t) F_t \\ F_t &\doteq \rho_{t-1} \gamma_t F_{t-1} + I_t, \quad \text{with } F_0 \doteq i(S_0),\end{aligned}$$

where $M_t \geq 0$ is the general form of *emphasis*, $F_t \geq 0$ is termed the *followon trace*, and $I_t \geq 0$ is the *interest*, as described in Section 11.8. Note that M_t , like δ_t , is not really an additional memory variable. It can be removed from the algorithm by substituting its definition into the eligibility-trace equation. Pseudocode and software for the true online version of Emphatic-TD(λ) are available on the web (Sutton, 2015b).

In the on-policy case ($\rho_t = 1$, for all t), Emphatic-TD(λ) is similar to conventional TD(λ), but still significantly different. In fact, whereas Emphatic-TD(λ) is guaranteed to converge for all state-dependent λ functions, TD(λ) is not. TD(λ) is guaranteed convergent only for all constant λ . See Yu's counterexample (Ghiassian, Rafiee, and Sutton, 2016).

12.12 Implementation Issues

It might at first appear that tabular methods using eligibility traces are much more complex than one-step methods. A naive implementation would require every state (or state-action pair) to update both its value estimate and its eligibility trace on every time step. This would not be a problem for implementations on single-instruction, multiple-data, parallel computers or in plausible artificial neural network (ANN) implementations, but it is a problem for implementations on conventional serial computers. Fortunately, for typical values of λ and γ the eligibility traces of almost all states are almost always nearly zero; only those states that have recently been visited will have traces significantly greater than zero and only these few states need to be updated to closely approximate these algorithms.

In practice, then, implementations on conventional computers may keep track of and update only the few traces that are significantly greater than zero. Using this trick, the computational expense of using traces in tabular methods is typically just a few times that of a one-step method. The exact multiple of course depends on λ and γ and on the expense of the other computations. Note that the tabular case is in some sense the worst case for the computational complexity of eligibility traces. When function approximation is used, the computational advantages of not using traces generally decrease. For example, if ANNs and backpropagation are used, then eligibility traces generally cause only a doubling of the required memory and computation per step. Truncated λ -return methods (Section 12.3) can be computationally efficient on conventional computers though they always require some additional memory.

12.13 Conclusions

Eligibility traces in conjunction with TD errors provide an efficient, incremental way of shifting and choosing between Monte Carlo and TD methods. The n -step methods of Chapter 7 also enabled this, but eligibility trace methods are more general, often faster to learn, and offer different computational complexity tradeoffs. This chapter has offered an introduction to the elegant, emerging theoretical understanding of eligibility traces for on- and off-policy learning and for variable bootstrapping and discounting. One aspect of this elegant theory is true online methods, which exactly reproduce the behavior of expensive ideal methods while retaining the computational congeniality of conventional TD methods. Another aspect is the possibility of derivations that automatically convert from intuitive forward-view methods to more efficient incremental backward-view algorithms. We illustrated this general idea in a derivation that started with a classical, expensive Monte Carlo algorithm and ended with a cheap incremental non-TD implementation using the same novel eligibility trace used in true online TD methods.

As we mentioned in Chapter 5, Monte Carlo methods may have advantages in non-Markov tasks because they do not bootstrap. Because eligibility traces make TD methods more like Monte Carlo methods, they also can have advantages in these cases. If one wants to use TD methods because of their other advantages, but the task is at least partially non-Markov, then the use of an eligibility trace method is indicated. Eligibility traces are the first line of defense against both long-delayed rewards and non-Markov tasks.

By adjusting λ , we can place eligibility trace methods anywhere along a continuum from Monte Carlo to one-step TD methods. Where shall we place them? We do not yet have a good theoretical answer to this question, but a clear empirical answer appears to be emerging. On tasks with many steps per episode, or many steps within the half-life of discounting, it appears significantly better to use eligibility traces than not to (e.g., see Figure 12.14). On the other hand, if the traces are so long as to produce a pure Monte Carlo method, or nearly so, then performance degrades sharply. An intermediate mixture appears to be the best choice. Eligibility traces should be used to bring us toward Monte Carlo methods, but not all the way there. In the future it may be possible to more finely vary the trade-off between TD and Monte Carlo methods by using variable λ , but at present it is not clear how this can be done reliably and usefully.

Methods using eligibility traces require more computation than one-step methods, but in return they offer significantly faster learning, particularly when rewards are delayed by many steps. Thus it often makes sense to use eligibility traces when data are scarce and cannot be repeatedly processed, as is often the case in online applications. On the other hand, in off-line applications in which data can be generated cheaply, perhaps from an inexpensive simulation, then it often does not pay to use eligibility traces. In these cases the objective is not to get more out of a limited amount of data, but simply to process as much data as possible as quickly as possible. In these cases the speedup per datum due to traces is typically not worth their computational cost, and one-step methods are favored.

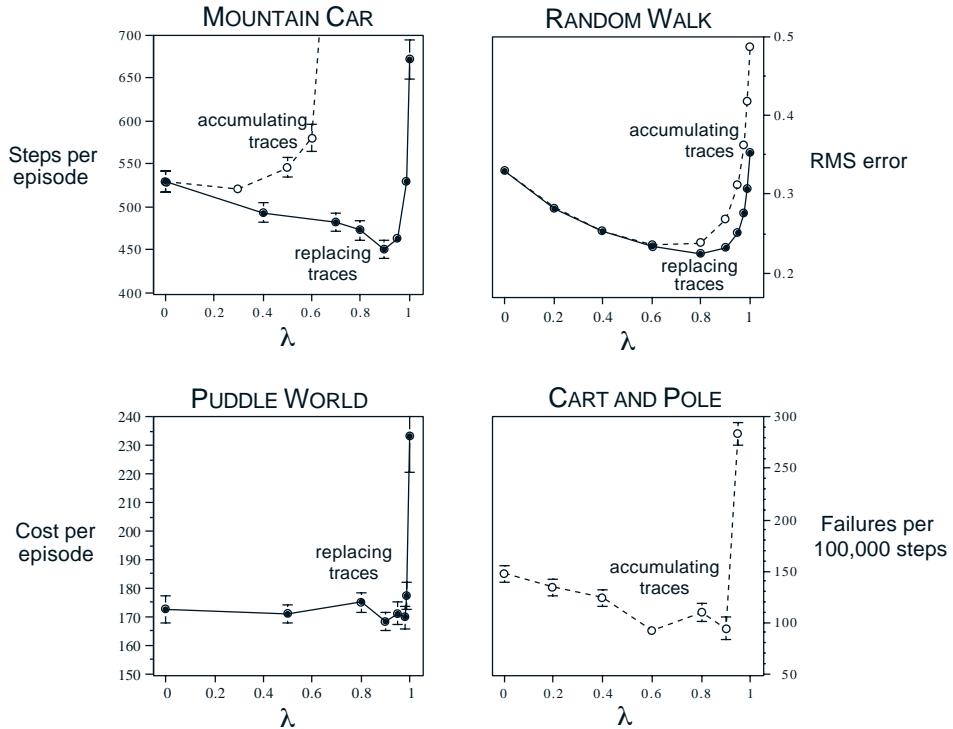


Figure 12.14: The effect of λ on reinforcement learning performance in four different test problems. In all cases, performance is generally best (a *lower* number in the graph) at an intermediate value of λ . The two left panels are applications to simple continuous-state control tasks using the Sarsa(λ) algorithm and tile coding, with either replacing or accumulating traces (Sutton, 1996). The upper-right panel is for policy evaluation on a random walk task using TD(λ) (Singh and Sutton, 1996). The lower right panel is unpublished data for the pole-balancing task (Example 3.4) from an earlier study (Sutton, 1984).

Bibliographical and Historical Remarks

Eligibility traces came into reinforcement learning via the fecund ideas of Klopf (1972). Our use of eligibility traces is based on Klopf’s work (Sutton, 1978a, 1978b, 1978c; Barto and Sutton, 1981a, 1981b; Sutton and Barto, 1981a; Barto, Sutton, and Anderson, 1983; Sutton, 1984). We may have been the first to use the term “eligibility trace” (Sutton and Barto, 1981a). The idea that stimuli produce after effects in the nervous system that are important for learning is very old (see Chapter 14). Some of the earliest uses of eligibility traces were in the actor–critic methods discussed in Chapter 13 (Barto, Sutton, and Anderson, 1983; Sutton, 1984).

- 12.1** Compound updates were called “complex backups” in the first edition of this book.

The λ -return and its error-reduction properties were introduced by Watkins (1989) and further developed by Jaakkola, Jordan, and Singh (1994). The random walk results in this and subsequent sections are new to this text, as are the terms “forward view” and “backward view.” The notion of a λ -return algorithm was introduced in the first edition of this text. The more refined treatment presented here was developed in conjunction with Harm van Seijen (e.g., van Seijen and Sutton, 2014).

- 12.2** TD(λ) with accumulating traces was introduced by Sutton (1988, 1984). Convergence in the mean was proved by Dayan (1992), and with probability 1 by many researchers, including Peng (1993), Dayan and Sejnowski (1994), Tsitsiklis (1994), and Gurvits, Lin, and Hanson (1994). The bound on the error of the asymptotic λ -dependent solution of linear TD(λ) is due to Tsitsiklis and Van Roy (1997).

- 12.3** Truncated TD methods were developed by Cichosz (1995) and van Seijen (2016).

- 12.4** The idea of redoing updates was extensively developed by van Seijen, originally under the name “best-match learning” (van Seijen, 2011; van Seijen, Whiteson, van Hasselt, and Weiring, 2011).

- 12.5** True online TD(λ) is primarily due to Harm van Seijen (van Seijen and Sutton, 2014; van Seijen et al., 2016) though some of its key ideas were discovered independently by Hado van Hasselt (personal communication). The name “dutch traces” is in recognition of the contributions of both scientists. Replacing traces are due to Singh and Sutton (1996).

- 12.6** The material in this section is from van Hasselt and Sutton (2015).

- 12.7** Sarsa(λ) with accumulating traces was first explored as a control method by Rummery and Niranjan (1994; Rummery, 1995). True Online Sarsa(λ) was

introduced by van Seijen and Sutton (2014). The algorithm on page 307 was adapted from van Seijen et al. (2016). The Mountain Car results were made for this text, except for Figure 12.11 which is adapted from van Seijen and Sutton (2014).

- 12.8** Perhaps the first published discussion of variable λ was by Watkins (1989), who pointed out that the cutting off of the update sequence (Figure 12.12) in his $Q(\lambda)$ when a nongreedy action was selected could be implemented by temporarily setting λ to 0.

Variable λ was introduced in the first edition of this text. The roots of variable γ are in the work on options (Sutton, Precup, and Singh, 1999) and its precursors (Sutton, 1995a), becoming explicit in the $GQ(\lambda)$ paper (Maei and Sutton, 2010), which also introduced some of these recursive forms for the λ -returns.

A different notion of variable λ has been developed by Yu (2012).

- 12.9** Off-policy eligibility traces were introduced by Precup et al. (2000, 2001), then further developed by Bertsekas and Yu (2009), Maei (2011; Maei and Sutton, 2010), Yu (2012), and by Sutton, Mahmood, Precup, and van Hasselt (2014). The last reference in particular gives a powerful forward view for off-policy TD methods with general state-dependent λ and γ . The presentation here seems to be new.

This section ends with an elegant Expected Sarsa(λ) algorithm. Although it is a natural algorithm, to our knowledge it has not previously been described or tested in the literature.

- 12.10** Watkins's $Q(\lambda)$ is due to Watkins (1989). The tabular, episodic, off-line version has been proven convergent by Munos, Stepleton, Harutyunyan, and Bellemare (2016). Alternative $Q(\lambda)$ algorithms were proposed by Peng and Williams (1994, 1996) and by Sutton, Mahmood, Precup, and van Hasselt (2014). Tree Backup(λ) is due to Precup, Sutton, and Singh (2000).

- 12.11** GTD(λ) is due to Maei (2011). GQ(λ) is due to Maei and Sutton (2010). HTD(λ) is due to White and White (2016) based on the one-step HTD algorithm introduced by Hackman (2012). The latest developments in the theory of Gradient-TD methods are by Yu (2017). Emphatic TD(λ) was introduced by Sutton, Mahmood, and White (2016), who proved its stability. Yu (2015, 2016) proved its convergence, and the algorithm was developed further by Hallak et al. (2015, 2016).

Chapter 13

Policy Gradient Methods

In this chapter we consider something new. So far in this book almost all the methods have been *action-value methods*; they learned the values of actions and then selected actions based on their estimated action values¹; their policies would not even exist without the action-value estimates. In this chapter we consider methods that instead learn a *parameterized policy* that can select actions without consulting a value function. A value function may still be used to *learn* the policy parameter, but is not required for action selection. We use the notation $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ for the policy's parameter vector. Thus we write $\pi(a|s, \boldsymbol{\theta}) = \Pr\{A_t=a | S_t=s, \boldsymbol{\theta}_t=\boldsymbol{\theta}\}$ for the probability that action a is taken at time t given that the environment is in state s at time t with parameter $\boldsymbol{\theta}$. If a method uses a learned value function as well, then the value function's weight vector is denoted $\mathbf{w} \in \mathbb{R}^d$ as usual, as in $\hat{v}(s, \mathbf{w})$.

In this chapter we consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter. These methods seek to *maximize* performance, so their updates approximate gradient *ascent* in J :

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}, \quad (13.1)$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$. All methods that follow this general schema we call *policy gradient methods*, whether or not they also learn an approximate value function. Methods that learn approximations to both policy and value functions are often called *actor-critic methods*, where ‘actor’ is a reference to the learned policy, and ‘critic’ refers to the learned value function, usually a state-value function. First we treat the episodic case, in which performance is defined as the value of the start state under the parameterized policy, before going on to consider the continuing case, in

¹The lone exception is the gradient bandit algorithms of Section 2.8. In fact, that section goes through many of the same steps, in the single-state bandit case, as we go through here for full MDPs. Reviewing that section would be good preparation for fully understanding this chapter.

which performance is defined as the average reward rate, as in Section 10.3. In the end, we are able to express the algorithms for both cases in very similar terms.

13.1 Policy Approximation and its Advantages

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s, \boldsymbol{\theta})$ is differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s, \boldsymbol{\theta})$ (the column vector of partial derivatives of $\pi(a|s, \boldsymbol{\theta})$ with respect to the components of $\boldsymbol{\theta}$) exists and is finite for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. In practice, to ensure exploration we generally require that the policy never becomes deterministic (i.e., that $\pi(a|s, \boldsymbol{\theta}) \in (0, 1)$, for all $s, a, \boldsymbol{\theta}$). In this section we introduce the most common parameterization for discrete action spaces and point out the advantages it offers over action-value methods. Policy-based methods also offer useful ways of dealing with continuous action spaces, as we describe later in Section 13.7.

If the action space is discrete and not too large, then a natural and common kind of parameterization is to form parameterized numerical preferences $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ for each state-action pair. The actions with the highest preferences in each state are given the highest probabilities of being selected, for example, according to an exponential soft-max distribution:

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s, a, \boldsymbol{\theta})}}{\sum_b e^{h(s, b, \boldsymbol{\theta})}}, \quad (13.2)$$

where $e \approx 2.71828$ is the base of the natural logarithm. Note that the denominator here is just what is required so that the action probabilities in each state sum to one. We call this kind of policy parameterization *soft-max in action preferences*.

The action preferences themselves can be parameterized arbitrarily. For example, they might be computed by a deep artificial neural network (ANN), where $\boldsymbol{\theta}$ is the vector of all the connection weights of the network (as in the AlphaGo system described in Section 16.6). Or the preferences could simply be linear in features,

$$h(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s, a), \quad (13.3)$$

using feature vectors $\mathbf{x}(s, a) \in \mathbb{R}^{d'}$ constructed by any of the methods described in Section 9.5.

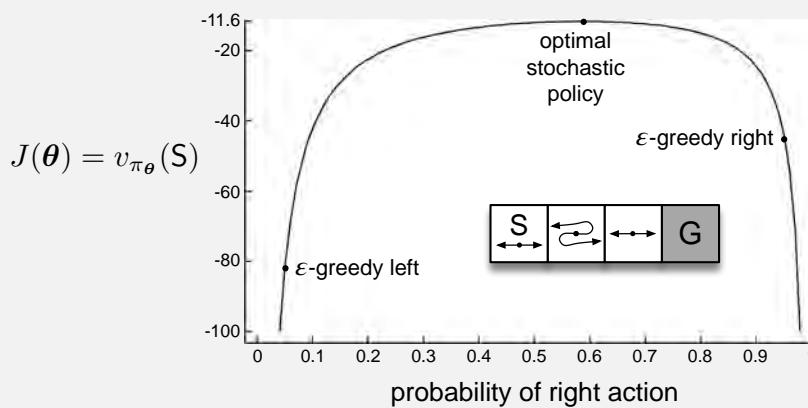
One advantage of parameterizing policies according to the soft-max in action preferences is that the approximate policy can approach a deterministic policy, whereas with ε -greedy action selection over action values there is always an ε probability of selecting a random action. Of course, one could select according to a soft-max distribution based on action values, but this alone would not allow the policy to approach a deterministic policy. Instead, the action-value estimates would converge to their corresponding true values, which would differ by a finite amount, translating to specific probabilities other than 0 and 1. If the soft-max distribution included a temperature parameter, then the temperature could be reduced over time to approach determinism, but in practice it would be difficult to choose the reduction schedule, or even the initial temperature, without more prior knowledge of the true action values than we would like to assume. Action preferences

are different because they do not approach specific values; instead they are driven to produce the optimal stochastic policy. If the optimal policy is deterministic, then the preferences of the optimal actions will be driven infinitely higher than all suboptimal actions (if permitted by the parameterization).

A second advantage of parameterizing policies according to the soft-max in action preferences is that it enables the selection of actions with arbitrary probabilities. In problems with significant function approximation, the best approximate policy may be stochastic. For example, in card games with imperfect information the optimal play is often to do two different things with specific probabilities, such as when bluffing in Poker. Action-value methods have no natural way of finding stochastic optimal policies, whereas policy approximating methods can, as shown in Example 13.1.

Example 13.1 Short corridor with switched actions

Consider the small corridor gridworld shown inset in the graph below. The reward is -1 per step, as usual. In each of the three nonterminal states there are only two actions, right and left. These actions have their usual consequences in the first and third states (left causes no movement in the first state), but in the second state they are reversed, so that right moves to the left and left moves to the right. The problem is difficult because all the states appear identical under the function approximation. In particular, we define $\mathbf{x}(s, \text{right}) = [1, 0]^\top$ and $\mathbf{x}(s, \text{left}) = [0, 1]^\top$, for all s . An action-value method with ε -greedy action selection is forced to choose between just two policies: choosing right with high probability $1 - \varepsilon/2$ on all steps or choosing left with the same high probability on all time steps. If $\varepsilon = 0.1$, then these two policies achieve a value (at the start state) of less than -44 and -82 , respectively, as shown in the graph. A method can do significantly better if it can learn a specific probability with which to select right. The best probability is about 0.59 , which achieves a value of about -11.6 .



Perhaps the simplest advantage that policy parameterization may have over action-value parameterization is that the policy may be a simpler function to approximate. Problems vary in the complexity of their policies and action-value functions. For some, the action-value function is simpler and thus easier to approximate. For others, the policy is simpler. In the latter case a policy-based method will typically learn faster and yield a superior asymptotic policy (as in Tetris; see Şimşek, Algórtá, and Kothiyal, 2016).

Finally, we note that the choice of policy parameterization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the reinforcement learning system. This is often the most important reason for using a policy-based learning method.

Exercise 13.1 Use your knowledge of the gridworld and its dynamics to determine an *exact* symbolic expression for the optimal probability of selecting the right action in Example 13.1. \square

13.2 The Policy Gradient Theorem

In addition to the practical advantages of policy parameterization over ε -greedy action selection, there is also an important theoretical advantage. With continuous policy parameterization the action probabilities change smoothly as a function of the learned parameter, whereas in ε -greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action values, if that change results in a different action having the maximal value. Largely because of this, stronger convergence guarantees are available for policy-gradient methods than for action-value methods. In particular, it is the continuity of the policy dependence on the parameters that enables policy-gradient methods to approximate gradient ascent (13.1).

The episodic and continuing cases define the performance measure, $J(\boldsymbol{\theta})$, differently and thus have to be treated separately to some extent. Nevertheless, we will try to present both cases uniformly, and we develop a notation so that the major theoretical results can be described with a single set of equations.

In this section we treat the episodic case, for which we define the performance measure as the value of the start state of the episode. We can simplify the notation without losing any meaningful generality by assuming that every episode starts in some particular (non-random) state s_0 . Then, in the episodic case we define performance as

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0), \tag{13.4}$$

where $v_{\pi_{\boldsymbol{\theta}}}$ is the true value function for $\pi_{\boldsymbol{\theta}}$, the policy determined by $\boldsymbol{\theta}$. From here on in our discussion we will assume no discounting ($\gamma = 1$) for the episodic case, although for completeness we do include the possibility of discounting in the boxed algorithms.

With function approximation it may seem challenging to change the policy parameter in a way that ensures improvement. The problem is that performance depends on both the action selections and the distribution of states in which those selections are made, and that both of these are affected by the policy parameter. Given a state, the effect of the policy parameter on the actions, and thus on reward, can be computed in a relatively straightforward way from knowledge of the parameterization. But the effect of the policy

Proof of the Policy Gradient Theorem (episodic case)

With just elementary calculus and re-arranging of terms, we can prove the policy gradient theorem from first principles. To keep the notation simple, we leave it implicit in all cases that π is a function of θ , and all gradients are also implicitly with respect to θ . First note that the gradient of the state-value function can be written in terms of the action-value function as

$$\nabla v_\pi(s) = \nabla \left[\sum_a \pi(a|s) q_\pi(s, a) \right], \quad \text{for all } s \in \mathcal{S} \quad (\text{Exercise 3.18})$$

$$= \sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \nabla q_\pi(s, a) \right] \quad (\text{product rule of calculus})$$

$$= \sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \nabla \sum_{s', r} p(s', r|s, a) (r + v_\pi(s')) \right] \quad (\text{Exercise 3.19 and Equation 3.2})$$

$$= \sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s'|s, a) \nabla v_\pi(s') \right] \quad (\text{Eq. 3.4})$$

$$= \sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s'|s, a) \right. \quad (\text{unrolling})$$

$$\left. \sum_{a'} \left[\nabla \pi(a'|s') q_\pi(s', a') + \pi(a'|s') \sum_{s''} p(s''|s', a') \nabla v_\pi(s'') \right] \right]$$

$$= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \rightarrow x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x, a),$$

after repeated unrolling, where $\Pr(s \rightarrow x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π . It is then immediate that

$$\begin{aligned} \nabla J(\theta) &= \nabla v_\pi(s_0) \\ &= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \rightarrow s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s, a) \\ &= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \quad (\text{box page 199}) \\ &= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) q_\pi(s, a) \\ &= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \quad (\text{Eq. 9.3}) \\ &\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \quad (\text{Q.E.D.}) \end{aligned}$$

on the state distribution is a function of the environment and is typically unknown. How can we estimate the performance gradient with respect to the policy parameter when the gradient depends on the unknown effect of policy changes on the state distribution?

Fortunately, there is an excellent theoretical answer to this challenge in the form of the *policy gradient theorem*, which provides an analytic expression for the gradient of performance with respect to the policy parameter (which is what we need to approximate for gradient ascent (13.1)) that does *not* involve the derivative of the state distribution. The policy gradient theorem for the episodic case establishes that

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta}), \quad (13.5)$$

where the gradients are column vectors of partial derivatives with respect to the components of $\boldsymbol{\theta}$, and π denotes the policy corresponding to parameter vector $\boldsymbol{\theta}$. The symbol \propto here means “proportional to”. In the episodic case, the constant of proportionality is the average length of an episode, and in the continuing case it is 1, so that the relationship is actually an equality. The distribution μ here (as in Chapters 9 and 10) is the on-policy distribution under π (see page 199). The policy gradient theorem is proved for the episodic case in the box on the previous page.

13.3 REINFORCE: Monte Carlo Policy Gradient

We are now ready to derive our first policy-gradient learning algorithm. Recall our overall strategy of stochastic gradient ascent (13.1), which requires a way to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter. The sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α , which is otherwise arbitrary. The policy gradient theorem gives an exact expression proportional to the gradient; all that is needed is some way of sampling whose expectation equals or approximates this expression. Notice that the right-hand side of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π ; if π is followed, then states will be encountered in these proportions. Thus

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \boldsymbol{\theta}) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \boldsymbol{\theta}) \right]. \end{aligned} \quad (13.6)$$

We could stop here and instantiate our stochastic gradient-ascent algorithm (13.1) as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \sum_a \hat{q}(S_t, a, \mathbf{w}) \nabla \pi(a|S_t, \boldsymbol{\theta}), \quad (13.7)$$

where \hat{q} is some learned approximation to q_π . This algorithm, which has been called an *all-actions* method because its update involves all of the actions, is promising and

deserving of further study, but our current interest is the classical REINFORCE algorithm (Williams, 1992) whose update at time t involves just A_t , the one action actually taken at time t .

We continue our derivation of REINFORCE by introducing A_t in the same way as we introduced S_t in (13.6)—by replacing a sum over the random variable’s possible values by an expectation under π , and then sampling the expectation. Equation (13.6) involves an appropriate sum over actions, but each term is not weighted by $\pi(a|S_t, \boldsymbol{\theta})$ as is needed for an expectation under π . So we introduce such a weighting, without changing the equality, by multiplying and then dividing the summed terms by $\pi(a|S_t, \boldsymbol{\theta})$. Continuing from (13.6), we have

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \mathbb{E}_\pi \left[\sum_a \pi(a|S_t, \boldsymbol{\theta}) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \boldsymbol{\theta})}{\pi(a|S_t, \boldsymbol{\theta})} \right] \\ &= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] \quad (\text{replacing } a \text{ by the sample } A_t \sim \pi) \\ &= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right], \quad (\text{because } \mathbb{E}_\pi[G_t|S_t, A_t] = q_\pi(S_t, A_t))\end{aligned}$$

where G_t is the return as usual. The final expression in brackets is exactly what is needed, a quantity that can be sampled on each time step whose expectation is proportional to the gradient. Using this sample to instantiate our generic stochastic gradient ascent algorithm (13.1) yields the REINFORCE update:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}. \quad (13.8)$$

This update has an intuitive appeal. Each increment is proportional to the product of a return G_t and a vector, the gradient of the probability of taking the action actually taken divided by the probability of taking that action. The vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t . The update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability. The former makes sense because it causes the parameter to move most in the directions that favor actions that yield the highest return. The latter makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return.

Note that REINFORCE uses the complete return from time t , which includes all future rewards up until the end of the episode. In this sense REINFORCE is a Monte Carlo algorithm and is well defined only for the episodic case with all updates made in retrospect after the episode is completed (like the Monte Carlo algorithms in Chapter 5). This is shown explicitly in the boxed algorithm on the next page.

Notice that the update in the last line of pseudocode appears rather different from the REINFORCE update rule (13.8). One difference is that the pseudocode uses the compact expression $\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}_t)$ for the fractional vector $\frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$ in (13.8). That these two expressions for the vector are equivalent follows from the identity $\nabla \ln x = \frac{\nabla x}{x}$.

This vector has been given several names and notations in the literature; we will refer to it simply as the *eligibility vector*. Note that it is the only place that the policy parameterization appears in the algorithm.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \theta)$

Algorithm parameter: step size $\alpha > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|s, \theta)$

Loop for each step of the episode $t = 0, 1, \dots, T - 1$:

$$\begin{aligned} G &\leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k \\ \theta &\leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t | S_t, \theta) \end{aligned} \tag{G_t}$$

The second difference between the pseudocode update and the REINFORCE update equation (13.8) is that the former includes a factor of γ^t . This is because, as mentioned earlier, in the text we are treating the non-discounted case ($\gamma = 1$) while in the boxed algorithms we are giving the algorithms for the general discounted case. All of the ideas go through in the discounted case with appropriate adjustments (including to the box on page 199) but involve additional complexity that distracts from the main ideas.

*Exercise 13.2 Generalize the box on page 199, the policy gradient theorem (13.5), the proof of the policy gradient theorem (page 325), and the steps leading to the REINFORCE update equation (13.8), so that (13.8) ends up with a factor of γ^t and thus aligns with the general algorithm given in the pseudocode. \square

Figure 13.1 shows the performance of REINFORCE on the short-corridor gridworld from Example 13.1.

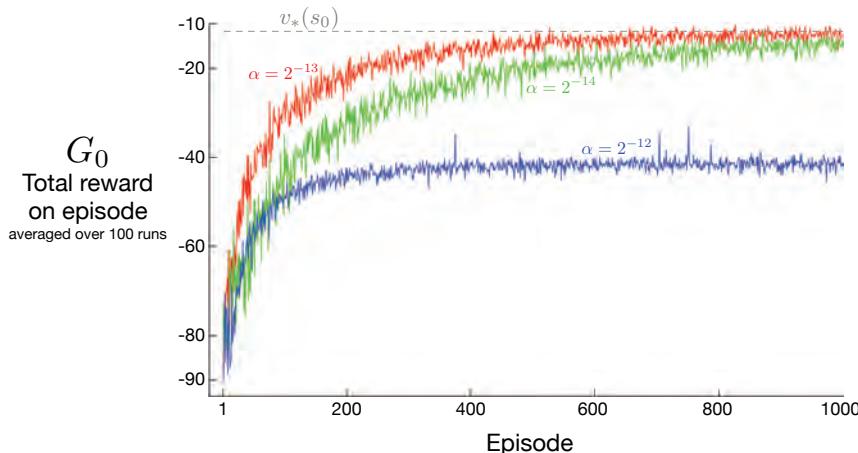


Figure 13.1: REINFORCE on the short-corridor gridworld (Example 13.1). With a good step size, the total reward per episode approaches the optimal value of the start state.

As a stochastic gradient method, REINFORCE has good theoretical convergence properties. By construction, the expected update over an episode is in the same direction as the performance gradient. This assures an improvement in expected performance for sufficiently small α , and convergence to a local optimum under standard stochastic approximation conditions for decreasing α . However, as a Monte Carlo method REINFORCE may be of high variance and thus produce slow learning.

Exercise 13.3 In Section 13.1 we considered policy parameterizations using the soft-max in action preferences (13.2) with linear action preferences (13.3). For this parameterization, prove that the eligibility vector is

$$\nabla \ln \pi(a|s, \boldsymbol{\theta}) = \mathbf{x}(s, a) - \sum_b \pi(b|s, \boldsymbol{\theta}) \mathbf{x}(s, b), \quad (13.9)$$

using the definitions and elementary calculus. \square

13.4 REINFORCE with Baseline

The policy gradient theorem (13.5) can be generalized to include a comparison of the action value to an arbitrary *baseline* $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \left(q_\pi(s, a) - b(s) \right) \nabla \pi(a|s, \boldsymbol{\theta}). \quad (13.10)$$

The baseline can be any function, even a random variable, as long as it does not vary with a ; the equation remains valid because the subtracted quantity is zero:

$$\sum_a b(s) \nabla \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla \sum_a \pi(a|s, \boldsymbol{\theta}) = b(s) \nabla 1 = 0.$$

The policy gradient theorem with baseline (13.10) can be used to derive an update rule using similar steps as in the previous section. The update rule that we end up with is a new version of REINFORCE that includes a general baseline:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(G_t - b(S_t) \right) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}. \quad (13.11)$$

Because the baseline could be uniformly zero, this update is a strict generalization of REINFORCE. In general, the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance. For example, we saw in Section 2.8 that an analogous baseline can significantly reduce the variance (and thus speed the learning) of gradient bandit algorithms. In the bandit algorithms the baseline was just a number (the average of the rewards seen so far), but for MDPs the baseline should vary with state. In some states all actions have high values and we need a high baseline to differentiate the higher valued actions from the less highly valued ones; in other states all actions will have low values and a low baseline is appropriate.

One natural choice for the baseline is an estimate of the state value, $\hat{v}(S_t, \mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^d$ is a weight vector learned by one of the methods presented in previous chapters.

Because REINFORCE is a Monte Carlo method for learning the policy parameter, θ , it seems natural to also use a Monte Carlo method to learn the state-value weights, \mathbf{w} . A complete pseudocode algorithm for REINFORCE with baseline using such a learned state-value function as the baseline is given in the box below.

REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \theta)$

Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$

Algorithm parameters: step sizes $\alpha^\theta > 0$, $\alpha^w > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot | \cdot, \theta)$

Loop for each step of the episode $t = 0, 1, \dots, T - 1$:

$$\begin{aligned} G &\leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k && (G_t) \\ \delta &\leftarrow G - \hat{v}(S_t, \mathbf{w}) \\ \mathbf{w} &\leftarrow \mathbf{w} + \alpha^w \delta \nabla \hat{v}(S_t, \mathbf{w}) \\ \theta &\leftarrow \theta + \alpha^\theta \gamma^t \delta \nabla \ln \pi(A_t | S_t, \theta) \end{aligned}$$

This algorithm has two step sizes, denoted α^θ and α^w (where α^θ is the α in (13.11)). Choosing the step size for values (here α^w) is relatively easy; in the linear case we have rules of thumb for setting it, such as $\alpha^w = 0.1/\mathbb{E}[\|\nabla \hat{v}(S_t, \mathbf{w})\|_\mu^2]$ (see Section 9.6). It is much less clear how to set the step size for the policy parameters, α^θ , whose best value depends on the range of variation of the rewards and on the policy parameterization.

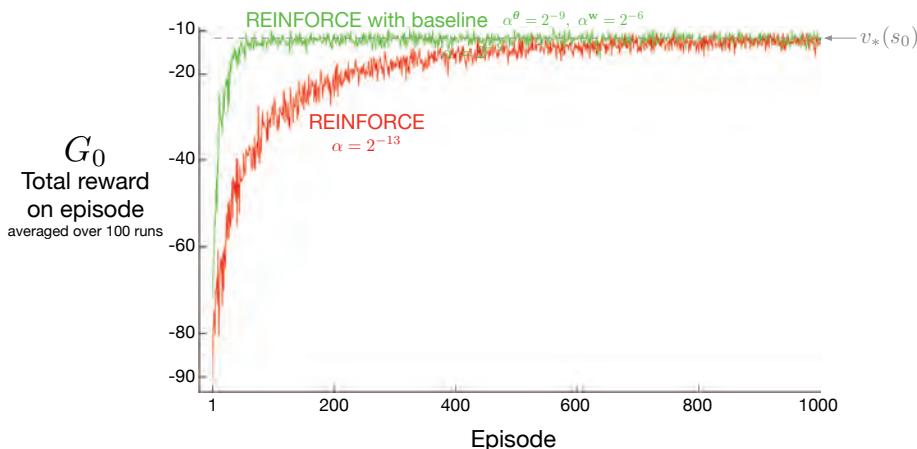


Figure 13.2: Adding a baseline to REINFORCE can make it learn much faster, as illustrated here on the short-corridor gridworld (Example 13.1). The step size used here for plain REINFORCE is that at which it performs best (to the nearest power of two; see Figure 13.1).

Figure 13.2 compares the behavior of REINFORCE with and without a baseline on the short-corridor gridworld (Example 13.1). Here the approximate state-value function used in the baseline is $\hat{v}(s, \mathbf{w}) = w$. That is, \mathbf{w} is a single component, w .

13.5 Actor–Critic Methods

In REINFORCE with baseline, the learned state-value function estimates the value of the *first* state of each state transition. This estimate sets a baseline for the subsequent return, but is made prior to the transition’s action and thus cannot be used to assess that action. In actor–critic methods, on the other hand, the state-value function is applied also to the *second* state of the transition. The estimated value of the second state, when discounted and added to the reward, constitutes the one-step return, $G_{t:t+1}$, which is a useful estimate of the actual return and thus *is* a way of assessing the action. As we have seen in the TD learning of value functions throughout this book, the one-step return is often superior to the actual return in terms of its variance and computational congeniality, even though it introduces bias. We also know how we can flexibly modulate the extent of the bias with n -step returns and eligibility traces (Chapters 7 and 12). When the state-value function is used to assess actions in this way it is called a *critic*, and the overall policy-gradient method is termed an *actor–critic* method. Note that the bias in the gradient estimate is not due to bootstrapping as such; the actor would be biased even if the critic was learned by a Monte Carlo method.

First consider one-step actor–critic methods, the analog of the TD methods introduced in Chapter 6 such as TD(0), Sarsa(0), and Q-learning. The main appeal of one-step methods is that they are fully online and incremental, yet avoid the complexities of eligibility traces. They are a special case of the eligibility trace methods, but easier to understand. One-step actor–critic methods replace the full return of REINFORCE (13.11) with the one-step return (and use a learned state-value function as the baseline) as follows:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(G_{t:t+1} - \hat{v}(S_t, \mathbf{w}) \right) \frac{\nabla \pi(A_t | S_t, \boldsymbol{\theta}_t)}{\pi(A_t | S_t, \boldsymbol{\theta}_t)} \quad (13.12)$$

$$= \boldsymbol{\theta}_t + \alpha \left(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \right) \frac{\nabla \pi(A_t | S_t, \boldsymbol{\theta}_t)}{\pi(A_t | S_t, \boldsymbol{\theta}_t)} \quad (13.13)$$

$$= \boldsymbol{\theta}_t + \alpha \delta_t \frac{\nabla \pi(A_t | S_t, \boldsymbol{\theta}_t)}{\pi(A_t | S_t, \boldsymbol{\theta}_t)}. \quad (13.14)$$

The natural state-value-function learning method to pair with this is semi-gradient TD(0). Pseudocode for the complete algorithm is given in the box at the top of the next page. Note that it is now a fully online, incremental algorithm, with states, actions, and rewards processed as they occur and then never revisited.

One-step Actor–Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
 Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
 Parameters: step sizes $\alpha^\theta > 0, \alpha^w > 0$
 Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
 Loop forever (for each episode):
 Initialize S (first state of episode)
 $I \leftarrow 1$
 Loop while S is not terminal (for each time step):
 $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
 Take action A , observe S', R
 $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$ (if S' is terminal, then $\hat{v}(S', \mathbf{w}) \doteq 0$)
 $\mathbf{w} \leftarrow \mathbf{w} + \alpha^w \delta \nabla \hat{v}(S, \mathbf{w})$
 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^\theta I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
 $I \leftarrow \gamma I$
 $S \leftarrow S'$

The generalizations to the forward view of n -step methods and then to a λ -return algorithm are straightforward. The one-step return in (13.12) is merely replaced by $G_{t:t+n}$ or G_t^λ respectively. The backward view of the λ -return algorithm is also straightforward, using separate eligibility traces for the actor and critic, each after the patterns in Chapter 12. Pseudocode for the complete algorithm is given in the box below.

Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
 Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
 Parameters: trace-decay rates $\lambda^\theta \in [0, 1], \lambda^w \in [0, 1]$; step sizes $\alpha^\theta > 0, \alpha^w > 0$
 Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
 Loop forever (for each episode):
 Initialize S (first state of episode)
 $\mathbf{z}^\theta \leftarrow \mathbf{0}$ (d' -component eligibility trace vector)
 $\mathbf{z}^w \leftarrow \mathbf{0}$ (d -component eligibility trace vector)
 $I \leftarrow 1$
 Loop while S is not terminal (for each time step):
 $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
 Take action A , observe S', R
 $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$ (if S' is terminal, then $\hat{v}(S', \mathbf{w}) \doteq 0$)
 $\mathbf{z}^w \leftarrow \gamma \lambda^w \mathbf{z}^w + \nabla \hat{v}(S, \mathbf{w})$
 $\boldsymbol{\theta} \leftarrow \gamma \lambda^\theta \mathbf{z}^\theta + I \nabla \ln \pi(A|S, \boldsymbol{\theta})$
 $\mathbf{w} \leftarrow \mathbf{w} + \alpha^w \delta \mathbf{z}^w$
 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^\theta \delta \mathbf{z}^\theta$
 $I \leftarrow \gamma I$
 $S \leftarrow S'$

13.6 Policy Gradient for Continuing Problems

As discussed in Section 10.3, for continuing problems without episode boundaries we need to define performance in terms of the average rate of reward per time step:

$$\begin{aligned} J(\boldsymbol{\theta}) \doteq r(\pi) &\doteq \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{t=1}^h \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \lim_{t \rightarrow \infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)r, \end{aligned} \tag{13.15}$$

where μ is the steady-state distribution under π , $\mu(s) \doteq \lim_{t \rightarrow \infty} \Pr\{S_t = s \mid A_{0:t} \sim \pi\}$, which is assumed to exist and to be independent of S_0 (an ergodicity assumption). Remember that this is the special distribution under which, if you select actions according to π , you remain in the same distribution:

$$\sum_s \mu(s) \sum_a \pi(a|s, \boldsymbol{\theta}) p(s'|s, a) = \mu(s'), \text{ for all } s' \in \mathcal{S}. \tag{13.16}$$

Complete pseudocode for the actor–critic algorithm in the continuing case (backward view) is given in the box below.

Actor–Critic with Eligibility Traces (continuing), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

```

Input: a differentiable policy parameterization  $\pi(a|s, \boldsymbol{\theta})$ 
Input: a differentiable state-value function parameterization  $\hat{v}(s, \mathbf{w})$ 
Algorithm parameters:  $\lambda^{\mathbf{w}} \in [0, 1]$ ,  $\lambda^{\boldsymbol{\theta}} \in [0, 1]$ ,  $\alpha^{\mathbf{w}} > 0$ ,  $\alpha^{\boldsymbol{\theta}} > 0$ ,  $\alpha^{\bar{R}} > 0$ 
Initialize  $\bar{R} \in \mathbb{R}$  (e.g., to 0)
Initialize state-value weights  $\mathbf{w} \in \mathbb{R}^d$  and policy parameter  $\boldsymbol{\theta} \in \mathbb{R}^{d'}$  (e.g., to  $\mathbf{0}$ )
Initialize  $S \in \mathcal{S}$  (e.g., to  $s_0$ )
 $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$  ( $d$ -component eligibility trace vector)
 $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$  ( $d'$ -component eligibility trace vector)
Loop forever (for each time step):
   $A \sim \pi(\cdot|S, \boldsymbol{\theta})$ 
  Take action  $A$ , observe  $S', R$ 
   $\delta \leftarrow R - \bar{R} + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$ 
   $\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}} \delta$ 
   $\mathbf{z}^{\mathbf{w}} \leftarrow \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S, \mathbf{w})$ 
   $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + \nabla \ln \pi(A|S, \boldsymbol{\theta})$ 
   $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$ 
   $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$ 
   $S \leftarrow S'$ 

```

Naturally, in the continuing case, we define values, $v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s]$ and $q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$, with respect to the differential return:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots \quad (13.17)$$

With these alternate definitions, the policy gradient theorem as given for the episodic case (13.5) remains true for the continuing case. A proof is given in the box on the next page. The forward and backward view equations also remain the same.

Proof of the Policy Gradient Theorem (continuing case)

The proof of the policy gradient theorem for the continuing case begins similarly to the episodic case. Again we leave it implicit in all cases that π is a function of θ and that the gradients are with respect to θ . Recall that in the continuing case $J(\theta) = r(\pi)$ (13.15) and that v_π and q_π denote values with respect to the differential return (13.17). The gradient of the state-value function can be written, for any $s \in \mathcal{S}$, as

$$\begin{aligned} \nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s) q_\pi(s, a) \right], \quad \text{for all } s \in \mathcal{S} \\ &= \sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \nabla q_\pi(s, a) \right] \quad (\text{product rule of calculus}) \\ &= \sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \nabla \sum_{s', r} p(s', r | s, a) (r - r(\theta) + v_\pi(s')) \right] \\ &= \sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \left[-\nabla r(\theta) + \sum_{s'} p(s' | s, a) \nabla v_\pi(s') \right] \right]. \end{aligned} \quad (\text{Exercise 3.18})$$

After re-arranging terms, we obtain

$$\nabla r(\theta) = \sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s' | s, a) \nabla v_\pi(s') \right] - \nabla v_\pi(s).$$

Notice that the left-hand side can be written $\nabla J(\theta)$, and that it does not depend on s . Thus the right-hand side does not depend on s either, and we can safely sum it over all $s \in \mathcal{S}$, weighted by $\mu(s)$, without changing it (because $\sum_s \mu(s) = 1$):

$$\begin{aligned} \nabla J(\theta) &= \sum_s \mu(s) \left(\sum_a \left[\nabla \pi(a|s) q_\pi(s, a) + \pi(a|s) \sum_{s'} p(s' | s, a) \nabla v_\pi(s') \right] - \nabla v_\pi(s) \right) \\ &= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \\ &\quad + \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s'} p(s' | s, a) \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s) \end{aligned}$$

$$\begin{aligned}
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \\
&\quad + \underbrace{\sum_{s'} \sum_s \mu(s) \sum_a \pi(a|s) p(s'|s, a) \nabla v_\pi(s')}_{\mu(s')} - \sum_s \mu(s) \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) + \sum_{s'} \mu(s') \nabla v_\pi(s') - \sum_s \mu(s) \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a). \quad \text{Q.E.D.}
\end{aligned} \tag{13.16}$$

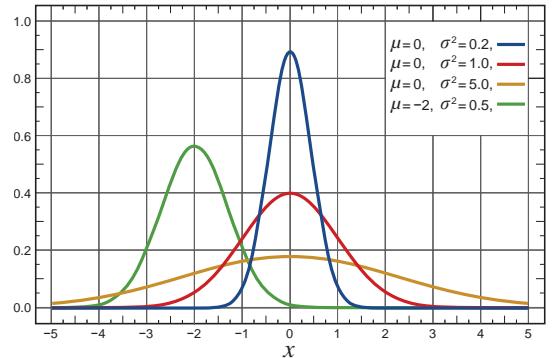
13.7 Policy Parameterization for Continuous Actions

Policy-based methods offer practical ways of dealing with large action spaces, even continuous spaces with an infinite number of actions. Instead of computing learned probabilities for each of the many actions, we instead learn statistics of the probability distribution. For example, the action set might be the real numbers, with actions chosen from a normal (Gaussian) distribution.

The probability density function for the normal distribution is conventionally written

$$p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{13.18}$$

where μ and σ here are the mean and standard deviation of the normal distribution, and of course π here is just the number $\pi \approx 3.14159$. The probability density functions for several different means and standard deviations are shown to the right. The value $p(x)$ is the *density* of the probability at x , not the probability. It can be greater than 1; it is the total area under $p(x)$ that must sum to 1. In general, one can take the integral under $p(x)$ for any range of x values to get the probability of x falling within that range.



To produce a policy parameterization, the policy can be defined as the normal probability density over a real-valued scalar action, with mean and standard deviation given by parametric function approximators that depend on the state. That is,

$$\pi(a|s, \theta) \doteq \frac{1}{\sigma(s, \theta)\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s, \theta))^2}{2\sigma(s, \theta)^2}\right), \tag{13.19}$$

where $\mu : \mathcal{S} \times \mathbb{R}^{d'} \rightarrow \mathbb{R}$ and $\sigma : \mathcal{S} \times \mathbb{R}^{d'} \rightarrow \mathbb{R}^+$ are two parameterized function approximators.

To complete the example we need only give a form for these approximators. For this we divide the policy's parameter vector into two parts, $\boldsymbol{\theta} = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma]^\top$, one part to be used for the approximation of the mean and one part for the approximation of the standard deviation. The mean can be approximated as a linear function. The standard deviation must always be positive and is better approximated as the exponential of a linear function. Thus

$$\mu(s, \boldsymbol{\theta}) \doteq \boldsymbol{\theta}_\mu^\top \mathbf{x}_\mu(s) \quad \text{and} \quad \sigma(s, \boldsymbol{\theta}) \doteq \exp\left(\boldsymbol{\theta}_\sigma^\top \mathbf{x}_\sigma(s)\right), \quad (13.20)$$

where $\mathbf{x}_\mu(s)$ and $\mathbf{x}_\sigma(s)$ are state feature vectors perhaps constructed by one of the methods described in Section 9.5. With these definitions, all the algorithms described in the rest of this chapter can be applied to learn to select real-valued actions.

Exercise 13.4 Show that for the Gaussian policy parameterization (Equations 13.19 and 13.20) the eligibility vector has the following two parts:

$$\nabla \ln \pi(a|s, \boldsymbol{\theta}_\mu) = \frac{\nabla \pi(a|s, \boldsymbol{\theta}_\mu)}{\pi(a|s, \boldsymbol{\theta})} = \frac{1}{\sigma(s, \boldsymbol{\theta})^2} (a - \mu(s, \boldsymbol{\theta})) \mathbf{x}_\mu(s), \text{ and}$$

$$\nabla \ln \pi(a|s, \boldsymbol{\theta}_\sigma) = \frac{\nabla \pi(a|s, \boldsymbol{\theta}_\sigma)}{\pi(a|s, \boldsymbol{\theta})} = \left(\frac{(a - \mu(s, \boldsymbol{\theta}))^2}{\sigma(s, \boldsymbol{\theta})^2} - 1 \right) \mathbf{x}_\sigma(s). \quad \square$$

Exercise 13.5 A *Bernoulli-logistic unit* is a stochastic neuron-like unit used in some ANNs (Section 9.7). Its input at time t is a feature vector $\mathbf{x}(S_t)$; its output, A_t , is a random variable having two values, 0 and 1, with $\Pr\{A_t = 1\} = P_t$ and $\Pr\{A_t = 0\} = 1 - P_t$ (the Bernoulli distribution). Let $h(s, 0, \boldsymbol{\theta})$ and $h(s, 1, \boldsymbol{\theta})$ be the preferences in state s for the unit's two actions given policy parameter $\boldsymbol{\theta}$. Assume that the difference between the action preferences is given by a weighted sum of the unit's input vector, that is, assume that $h(s, 1, \boldsymbol{\theta}) - h(s, 0, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s)$, where $\boldsymbol{\theta}$ is the unit's weight vector.

- (a) Show that if the exponential soft-max distribution (13.2) is used to convert action preferences to policies, then $P_t = \pi(1|S_t, \boldsymbol{\theta}_t) = 1/(1 + \exp(-\boldsymbol{\theta}_t^\top \mathbf{x}(S_t)))$ (the logistic function).
- (b) What is the Monte-Carlo REINFORCE update of $\boldsymbol{\theta}_t$ to $\boldsymbol{\theta}_{t+1}$ upon receipt of return G_t ?
- (c) Express the eligibility $\nabla \ln \pi(a|s, \boldsymbol{\theta})$ for a Bernoulli-logistic unit, in terms of a , $\mathbf{x}(s)$, and $\pi(a|s, \boldsymbol{\theta})$ by calculating the gradient.

Hint for part (c): Define $P = \pi(1|s, \boldsymbol{\theta})$ and compute the derivative of the logarithm, for each action, using the chain rule on P . Combine the two results into one expression that depends on a and P , and then use the chain rule again, this time on $\boldsymbol{\theta}^\top \mathbf{x}(s)$, noting that the derivative of the logistic function $f(x) = 1/(1 + e^{-x})$ is $f(x)(1 - f(x))$. \square

13.8 Summary

Prior to this chapter, this book focused on *action-value methods*—meaning methods that learn action values and then use them to determine action selections. In this chapter, on the other hand, we considered methods that learn a parameterized policy that enables actions to be taken without consulting action-value estimates. In particular, we have considered *policy-gradient methods*—meaning methods that update the policy parameter on each step in the direction of an estimate of the gradient of performance with respect to the policy parameter.

Methods that learn and store a policy parameter have many advantages. They can learn specific probabilities for taking the actions. They can learn appropriate levels of exploration and approach deterministic policies asymptotically. They can naturally handle continuous action spaces. All these things are easy for policy-based methods but awkward or impossible for ϵ -greedy methods and for action-value methods in general. In addition, on some problems the policy is just simpler to represent parametrically than the value function; these problems are more suited to parameterized policy methods.

Parameterized policy methods also have an important theoretical advantage over action-value methods in the form of the *policy gradient theorem*, which gives an exact formula for how performance is affected by the policy parameter that does not involve derivatives of the state distribution. This theorem provides a theoretical foundation for all policy gradient methods.

The REINFORCE method follows directly from the policy gradient theorem. Adding a state-value function as a *baseline* reduces REINFORCE’s variance without introducing bias. If the state-value function is also used to assess—or criticize—the policy’s action selections, then the value function is called a *critic* and the policy is called an *actor*; the overall method is called an *actor-critic* method. The critic introduces bias into the actor’s gradient estimates, but is often desirable for the same reason that bootstrapping TD methods are often superior to Monte Carlo methods (substantially reduced variance).

Overall, policy-gradient methods provide a significantly different set of strengths and weaknesses than action-value methods. Today they are less well understood in some respects, but a subject of excitement and ongoing research.

Bibliographical and Historical Remarks

Methods that we now see as related to policy gradients were actually some of the earliest to be studied in reinforcement learning (Witten, 1977; Barto, Sutton, and Anderson, 1983; Sutton, 1984; Williams, 1987, 1992) and in predecessor fields (see Phansalkar and Thathachar, 1995). They were largely supplanted in the 1990s by the action-value methods that are the focus of the other chapters of this book. In recent years, however, attention has returned to actor–critic methods and to policy-gradient methods in general. Among the further developments beyond what we cover here are natural-gradient methods (Amari, 1998; Kakade, 2002, Peters, Vijayakumar and Schaal, 2005; Peters and Schaal, 2008; Park, Kim and Kang, 2005; Bhatnagar, Sutton, Ghavamzadeh and Lee, 2009; see Grondman, Busoniu, Lopes and Babuska, 2012), deterministic policy-gradient methods

(Silver et al., 2014), off-policy policy-gradient methods (Degris, White, and Sutton, 2012; Maei, 2018), and entropy regularization (see Schulman, Chen, and Abbeel, 2017). Major applications include acrobatic helicopter autopilots and AlphaGo (Section 16.6).

Our presentation in this chapter is based primarily on that by Sutton, McAllester, Singh, and Mansour (2000), who introduced the term “policy gradient methods.” A useful overview is provided by Bhatnagar et al. (2009). One of the earliest related works is by Aleksandrov, Sysoyev, and Shemeneva (1968). Thomas (2014) first realized that the factor of γ^t , as specified in the boxed algorithms of this chapter, was needed in the case of discounted episodic problems.

13.1 Example 13.1 and the results with it in this chapter were developed with Eric Graves.

13.2 The policy gradient theorem here and on page 334 was first obtained by Marbach and Tsitsiklis (1998, 2001) and then independently by Sutton et al. (2000). A similar expression was obtained by Cao and Chen (1997). Other early results are due to Konda and Tsitsiklis (2000, 2003), Baxter and Bartlett (2001), and Baxter, Bartlett, and Weaver (2001). Some additional results are developed by Sutton, Singh, and McAllester (2000).

13.3 REINFORCE is due to Williams (1987, 1992). Phansalkar and Thathachar (1995) proved both local and global convergence theorems for modified versions of REINFORCE algorithms.

The all-actions algorithm was first presented in an unpublished but widely circulated incomplete paper (Sutton, Singh, and McAllester, 2000) and then developed further by Ciosek and Whiteson (2017, 2018), who termed it “expected policy gradients,” and by Asadi, Allen, Roderick, Mohamed, Konidaris, and Littman (2017), who called it “mean actor critic.”

13.4 The baseline was introduced in Williams’s (1987, 1992) original work. Greensmith, Bartlett, and Baxter (2004) analyzed an arguably better baseline (see Dick, 2015). Thomas and Brunskill (2017) argue that an action-dependent baseline can be used without incurring bias.

13.5–6 Actor–critic methods were among the earliest to be investigated in reinforcement learning (Witten, 1977; Barto, Sutton, and Anderson, 1983; Sutton, 1984). The algorithms presented here are based on the work of Degris, White, and Sutton (2012). Actor–critic methods are sometimes referred to as advantage actor–critic (“A2C”) methods in the literature.

13.7 The first to show how continuous actions could be handled this way appears to have been Williams (1987, 1992). The figure on page 335 is adapted from Wikipedia.

Part III: Looking Deeper

In this last part of the book we look beyond the standard reinforcement learning ideas presented in the first two parts of the book to briefly survey their relationships with psychology and neuroscience, a sampling of reinforcement learning applications, and some of the active frontiers for future reinforcement learning research.

Chapter 14

Psychology

In previous chapters we developed ideas for algorithms based on computational considerations alone. In this chapter we look at some of these algorithms from another perspective: the perspective of psychology and its study of how animals learn. The goals of this chapter are, first, to discuss ways that reinforcement learning ideas and algorithms correspond to what psychologists have discovered about animal learning, and second, to explain the influence reinforcement learning is having on the study of animal learning. The clear formalism provided by reinforcement learning that systemizes tasks, returns, and algorithms is proving to be enormously useful in making sense of experimental data, in suggesting new kinds of experiments, and in pointing to factors that may be critical to manipulate and to measure. The idea of optimizing return over the long term that is at the core of reinforcement learning is contributing to our understanding of otherwise puzzling features of animal learning and behavior.

Some of the correspondences between reinforcement learning and psychological theories are not surprising because the development of reinforcement learning drew inspiration from psychological learning theories. However, as developed in this book, reinforcement learning explores idealized situations from the perspective of an artificial intelligence researcher or engineer, with the goal of solving computational problems with efficient algorithms, rather than to replicate or explain in detail how animals learn. As a result, some of the correspondences we describe connect ideas that arose independently in their respective fields. We believe these points of contact are specially meaningful because they expose computational principles important to learning, whether it is learning by artificial or by natural systems.

For the most part, we describe correspondences between reinforcement learning and learning theories developed to explain how animals like rats, pigeons, and rabbits learn in controlled laboratory experiments. Thousands of these experiments were conducted throughout the 20th century, and many are still being conducted today. Although sometimes dismissed as irrelevant to wider issues in psychology, these experiments probe subtle properties of animal learning, often motivated by precise theoretical questions. As psychology shifted its focus to more cognitive aspects of behavior, that is, to mental

processes such as thought and reasoning, animal learning experiments came to play less of a role in psychology than they once did. But this experimentation led to the discovery of learning principles that are elemental and widespread throughout the animal kingdom, principles that should not be neglected in designing artificial learning systems. In addition, as we shall see, some aspects of cognitive processing connect naturally to the computational perspective provided by reinforcement learning.

This chapter's final section includes references relevant to the connections we discuss as well as to connections we neglect. We hope this chapter encourages readers to probe all of these connections more deeply. Also included in this final section is a discussion of how the terminology used in reinforcement learning relates to that of psychology. Many of the terms and phrases used in reinforcement learning are borrowed from animal learning theories, but the computational/engineering meanings of these terms and phrases do not always coincide with their meanings in psychology.

14.1 Prediction and Control

The algorithms we describe in this book fall into two broad categories: algorithms for *prediction* and algorithms for *control*.¹ These categories arise naturally in solution methods for the reinforcement learning problem presented in Chapter 3. In many ways these categories respectively correspond to categories of learning extensively studied by psychologists: *classical*, or *Pavlovian*, *conditioning* and *instrumental*, or *operant*, *conditioning*. These correspondences are not completely accidental because of psychology's influence on reinforcement learning, but they are nevertheless striking because they connect ideas arising from different objectives.

The prediction algorithms presented in this book estimate quantities that depend on how features of an agent's environment are expected to unfold over the future. We specifically focus on estimating the amount of reward an agent can expect to receive over the future while it interacts with its environment. In this role, prediction algorithms are *policy evaluation algorithms*, which are integral components of algorithms for improving policies. But prediction algorithms are not limited to predicting future reward; they can predict any feature of the environment (see, for example, Modayil, White, and Sutton, 2014). The correspondence between prediction algorithms and classical conditioning rests on their common property of predicting upcoming stimuli, whether or not those stimuli are rewarding (or punishing).

The situation in an instrumental, or operant, conditioning experiment is different. Here, the experimental apparatus is set up so that an animal is given something it likes (a reward) or something it dislikes (a penalty) depending on what the animal did. The animal learns to increase its tendency to produce rewarded behavior and to decrease its tendency to produce penalized behavior. The reinforcing stimulus is said to be *contingent* on the animal's behavior, whereas in classical conditioning it is not (although it is difficult to remove all behavior contingencies in a classical conditioning experiment). Instrumental

¹What control means for us is different from what it typically means in animal learning theories; there the environment controls the agent instead of the other way around. See our comments on terminology at the end of this chapter.

conditioning experiments are like those that inspired Thorndike's Law of Effect that we briefly discuss in Chapter 1. *Control* is at the core of this form of learning, which corresponds to the operation of reinforcement learning's policy-improvement algorithms.

Thinking of classical conditioning in terms of prediction, and instrumental conditioning in terms of control, is a starting point for connecting our computational view of reinforcement learning to animal learning, but in reality, the situation is more complicated than this. There is more to classical conditioning than prediction; it also involves action, and so is a mode of control, sometimes called *Pavlovian control*. Further, classical and instrumental conditioning interact in interesting ways, with both sorts of learning likely being engaged in most experimental situations. Despite these complications, aligning the classical/instrumental distinction with the prediction/control distinction is a convenient first approximation in connecting reinforcement learning to animal learning.

In psychology, the term reinforcement is used to describe learning in both classical and instrumental conditioning. Originally referring only to the strengthening of a pattern of behavior, it is frequently also used for the weakening of a pattern of behavior. A stimulus considered to be the cause of the change in behavior is called a reinforcer, whether or not it is contingent on the animal's previous behavior. At the end of this chapter we discuss this terminology in more detail and how it relates to terminology used in machine learning.

14.2 Classical Conditioning

While studying the activity of the digestive system, the celebrated Russian physiologist Ivan Pavlov found that an animal's innate responses to certain triggering stimuli can come to be triggered by other stimuli that are quite unrelated to the inborn triggers. His experimental subjects were dogs that had undergone minor surgery to allow the intensity of their salivary reflex to be accurately measured. In one case he describes, the dog did not salivate under most circumstances, but about 5 seconds after being presented with food it produced about six drops of saliva over the next several seconds. After several repetitions of presenting another stimulus, one not related to food, in this case the sound of a metronome, shortly before the introduction of food, the dog salivated in response to the sound of the metronome in the same way it did to the food. "The activity of the salivary gland has thus been called into play by impulses of sound—a stimulus quite alien to food" (Pavlov, 1927, p. 22). Summarizing the significance of this finding, Pavlov wrote:

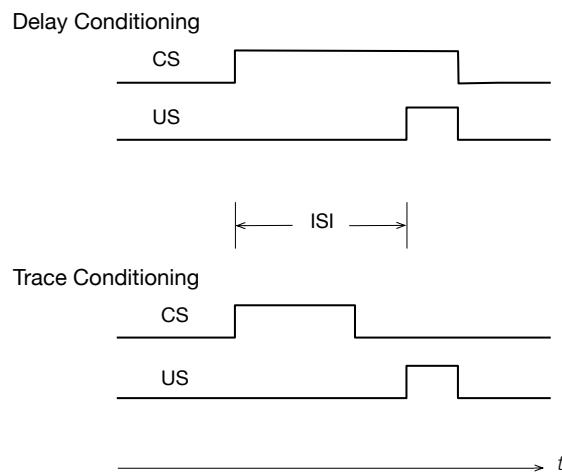
It is pretty evident that under natural conditions the normal animal must respond not only to stimuli which themselves bring immediate benefit or harm, but also to other physical or chemical agencies—waves of sound, light, and the like—which in themselves only *signal* the approach of these stimuli; though it is not the sight and sound of the beast of prey which is in itself harmful to the smaller animal, but its teeth and claws. (Pavlov, 1927, p. 14)

Connecting new stimuli to innate reflexes in this way is now called classical, or Pavlovian, conditioning. Pavlov (or more exactly, his translators) called inborn responses (e.g.,

salivation in his demonstration described above) “unconditioned responses” (URs), their natural triggering stimuli (e.g., food) “unconditioned stimuli” (USs), and new responses triggered by predictive stimuli (e.g., here also salivation) “conditioned responses” (CRs). A stimulus that is initially neutral, meaning that it does not normally elicit strong responses (e.g., the metronome sound), becomes a “conditioned stimulus” (CS) as the animal learns that it predicts the US and so comes to produce a CR in response to the CS. These terms are still used in describing classical conditioning experiments (though better translations would have been “conditional” and “unconditional” instead of conditioned and unconditioned). The US is called a reinforcer because it reinforces producing a CR in response to the CS.

The arrangement of stimuli in two common types of classical conditioning experiments is shown to the right. In *delay conditioning*, the CS extends throughout the interstimulus interval, or ISI, which is the time interval between the CS onset and the US onset (with the CS ending when the US ends in a common version shown here). In *trace conditioning*, the US begins after the CS ends, and the time interval between CS offset and US onset is called the trace interval.

The salivation of Pavlov’s dogs to the sound of a metronome is just one example of classical conditioning, which has been intensively studied across many response systems of many species of animals. URs are often preparatory in some way, like the salivation of Pavlov’s dog, or protective in some way, like an eye blink in response to something irritating to the eye, or freezing in response to seeing a predator. Experiencing the CS-US predictive relationship over a series of trials causes the animal to learn that the CS predicts the US so that the animal can respond to the CS with a CR that prepares the animal for, or protects it from, the predicted US. Some CRs are similar to the UR but begin earlier and differ in ways that increase their effectiveness. In one intensively studied type of experiment, for example, a tone CS reliably predicts a puff of air (the US) to a rabbit’s eye, triggering a UR consisting of the closure of a protective inner eyelid called the nictitating membrane. After one or more trials, the tone comes to trigger a CR consisting of membrane closure that begins before the air puff and eventually becomes timed so that peak closure occurs just when the air puff is likely to occur. This CR, being initiated in anticipation of the air puff and appropriately timed, offers better protection than simply initiating closure as a reaction to the irritating US. The ability to act in anticipation of important events by learning about predictive relationships among stimuli is so beneficial that it is widely present across the animal kingdom.



14.2.1 Blocking and Higher-order Conditioning

Many interesting properties of classical conditioning have been observed in experiments. Beyond the anticipatory nature of CRs, two widely observed properties figured prominently in the development of classical conditioning models: *blocking* and *higher-order conditioning*. Blocking occurs when an animal fails to learn a CR when a potential CS is presented along with another CS that had been used previously to condition the animal to produce that CR. For example, in the first stage of a blocking experiment involving rabbit nictitating membrane conditioning, a rabbit is first conditioned with a tone CS and an air puff US to produce the CR of closing its nictitating membrane in anticipation of the air puff. The experiment's second stage consists of additional trials in which a second stimulus, say a light, is added to the tone to form a compound tone/light CS followed by the same air puff US. In the experiment's third phase, the second stimulus alone—the light—is presented to the rabbit to see if the rabbit has learned to respond to it with a CR. It turns out that the rabbit produces very few, or no, CRs in response to the light: learning to the light had been *blocked* by the previous learning to the tone.² Blocking results like this challenged the idea that conditioning depends only on simple temporal contiguity, that is, that a necessary and sufficient condition for conditioning is that a US frequently follows a CS closely in time. In the next section we describe the *Rescorla–Wagner model* (Rescorla and Wagner, 1972) that offered an influential explanation for blocking.

Higher-order conditioning occurs when a previously-conditioned CS acts as a US in conditioning another initially neutral stimulus. Pavlov described an experiment in which his assistant first conditioned a dog to salivate to the sound of a metronome that predicted a food US, as described above. After this stage of conditioning, a number of trials were conducted in which a black square, to which the dog was initially indifferent, was placed in the dog's line of vision followed by the sound of the metronome—and this was *not* followed by food. In just ten trials, the dog began to salivate merely upon seeing the black square, despite the fact that the sight of it had never been followed by food. The sound of the metronome itself acted as a US in conditioning a salivation CR to the black square CS. This was second-order conditioning. If the black square had been used as a US to establish salivation CRs to another otherwise neutral CS, it would have been third-order conditioning, and so on. Higher-order conditioning is difficult to demonstrate, especially above the second order, in part because a higher-order reinforcer loses its reinforcing value due to not being repeatedly followed by the original US during higher-order conditioning trials. But under the right conditions, such as intermixing first-order trials with higher-order trials or by providing a general energizing stimulus, higher-order conditioning beyond the second order can be demonstrated. As we describe below, the *TD model of classical conditioning* uses the bootstrapping idea that is central to our approach to extend the Rescorla–Wagner model's account of blocking to include both the anticipatory nature of CRs and higher-order conditioning.

²Comparison with a control group is necessary to show that the previous conditioning to the tone is responsible for blocking learning to the light. This is done by trials with the tone/light CS but with no prior conditioning to the tone. Learning to the light in this case is unimpaired. Moore and Schmajuk (2008) give a full account of this procedure.

Higher-order instrumental conditioning occurs as well. In this case, a stimulus that consistently predicts primary reinforcement becomes a reinforcer itself, where reinforcement is primary if its rewarding or penalizing quality has been built into the animal by evolution. The predicting stimulus becomes a *secondary reinforcer*, or more generally, a *higher-order* or *conditioned reinforcer*—the latter being a better term when the predicted reinforcing stimulus is itself a secondary, or an even higher-order, reinforcer. A conditioned reinforcer delivers *conditioned reinforcement*: conditioned reward or conditioned penalty. Conditioned reinforcement acts like primary reinforcement in increasing an animal’s tendency to produce behavior that leads to conditioned reward, and to decrease an animal’s tendency to produce behavior that leads to conditioned penalty. (See our comments at the end of this chapter that explain how our terminology sometimes differs, as it does here, from terminology used in psychology.)

Conditioned reinforcement is a key phenomenon that explains, for instance, why we work for the conditioned reinforcer money, whose worth derives solely from what is predicted by having it. In actor–critic methods described in Section 13.5 (and discussed in the context of neuroscience in Sections 15.7 and 15.8), the critic uses a TD method to evaluate the actor’s policy, and its value estimates provide conditioned reinforcement to the actor, allowing the actor to improve its policy. This analog of higher-order instrumental conditioning helps address the credit-assignment problem mentioned in Section 1.7 because the critic gives moment-by-moment reinforcement to the actor when the primary reward signal is delayed. We discuss this more below in Section 14.4.

14.2.2 The Rescorla–Wagner Model

Rescorla and Wagner created their model mainly to account for blocking. The core idea of the Rescorla–Wagner model is that an animal only learns when events violate its expectations, in other words, only when the animal is surprised (although without necessarily implying any *conscious* expectation or emotion). We first present Rescorla and Wagner’s model using their terminology and notation before shifting to the terminology and notation we use to describe the TD model.

Here is how Rescorla and Wagner described their model. The model adjusts the “associative strength” of each component stimulus of a compound CS, which is a number representing how strongly or reliably that component is predictive of a US. When a compound CS consisting of several component stimuli is presented in a classical conditioning trial, the associative strength of each component stimulus changes in a way that depends on an associative strength associated with the entire stimulus compound, called the “aggregate associative strength,” and not just on the associative strength of each component itself.

Rescorla and Wagner considered a compound CS AX, consisting of component stimuli A and X, where the animal may have already experienced stimulus A, and stimulus X might be new to the animal. Let V_A , V_X , and V_{AX} respectively denote the associative strengths of stimuli A, X, and the compound AX. Suppose that on a trial the compound CS AX is followed by a US, which we label stimulus Y. Then the associative strengths of

the stimulus components change according to these expressions:

$$\begin{aligned}\Delta V_A &= \alpha_A \beta_Y (R_Y - V_{AX}) \\ \Delta V_X &= \alpha_X \beta_Y (R_Y - V_{AX}),\end{aligned}$$

where $\alpha_A \beta_Y$ and $\alpha_X \beta_Y$ are the step-size parameters, which depend on the identities of the CS components and the US, and R_Y is the asymptotic level of associative strength that the US Y can support. (Rescorla and Wagner used λ here instead of R , but we use R to avoid confusion with our use of λ and because we usually think of this as the magnitude of a reward signal, with the caveat that the US in classical conditioning is not necessarily rewarding or penalizing.) A key assumption of the model is that the aggregate associative strength V_{AX} is equal to $V_A + V_X$. The associative strengths as changed by these Δ s become the associative strengths at the beginning of the next trial.

To be complete, the model needs a response-generation mechanism, which is a way of mapping values of V s to CRs. Because this mapping would depend on details of the experimental situation, Rescorla and Wagner did not specify a mapping but simply assumed that larger V s would produce stronger or more likely CRs, and that negative V s would mean that there would be no CRs.

The Rescorla–Wagner model accounts for the acquisition of CRs in a way that explains blocking. As long as the aggregate associative strength, V_{AX} , of the stimulus compound is below the asymptotic level of associative strength, R_Y , that the US Y can support, the prediction error $R_Y - V_{AX}$ is positive. This means that over successive trials the associative strengths V_A and V_X of the component stimuli increase until the aggregate associative strength V_{AX} equals R_Y , at which point the associative strengths stop changing (unless the US changes). When a new component is added to a compound CS to which the animal has already been conditioned, further conditioning with the augmented compound produces little or no increase in the associative strength of the added CS component because the error has already been reduced to zero, or to a low value. The occurrence of the US is already predicted nearly perfectly, so little or no error—or surprise—is introduced by the new CS component. Prior learning blocks learning to the new component.

To transition from Rescorla and Wagner’s model to the TD model of classical conditioning (which we just call the TD model), we first recast their model in terms of the concepts that we are using throughout this book. Specifically, we match the notation we use for learning with linear function approximation (Section 9.4), and we think of the conditioning process as one of learning to predict the “magnitude of the US” on a trial on the basis of the compound CS presented on that trial, where the magnitude of a US Y is the R_Y of the Rescorla–Wagner model as given above. We also introduce states. Because the Rescorla–Wagner model is a *trial-level* model, meaning that it deals with how associative strengths change from trial to trial without considering any details about what happens within and between trials, we do not have to consider how states change during a trial until we present the full TD model in the following section. Instead, here we simply think of a state as a way of labeling a trial in terms of the collection of component CSs that are present on the trial.

Therefore, assume that trial-type, or state, s is described by a real-valued vector of features $\mathbf{x}(s) = (x_1(s), x_2(s), \dots, x_d(s))^\top$ where $x_i(s) = 1$ if CS _{i} , the i^{th} component of a

compound CS, is present on the trial and 0 otherwise. Then if the d -dimensional vector of associative strengths is \mathbf{w} , the aggregate associative strength for trial-type s is

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s). \quad (14.1)$$

This corresponds to a *value estimate* in reinforcement learning, and we think of it as the *US prediction*.

Now temporally let t denote the number of a complete trial and not its usual meaning as a time step (we revert to t 's usual meaning when we extend this to the TD model below), and assume that S_t is the state corresponding to trial t . Conditioning trial t updates the associative strength vector \mathbf{w}_t to \mathbf{w}_{t+1} as follows:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{x}(S_t), \quad (14.2)$$

where α is the step-size parameter, and—because here we are describing the Rescorla–Wagner model— δ_t is the *prediction error*

$$\delta_t = R_t - \hat{v}(S_t, \mathbf{w}_t). \quad (14.3)$$

R_t is the target of the prediction on trial t , that is, the magnitude of the US, or in Rescorla and Wagner's terms, the associative strength that the US on the trial can support. Note that because of the factor $\mathbf{x}(S_t)$ in (14.2), only the associative strengths of CS components present on a trial are adjusted as a result of that trial. You can think of the prediction error as a measure of surprise, and the aggregate associative strength as the animal's expectation that is violated when it does not match the target US magnitude.

From the perspective of machine learning, the Rescorla–Wagner model is an error-correction supervised learning rule. It is essentially the same as the Least Mean Square (LMS), or Widrow–Hoff, learning rule (Widrow and Hoff, 1960) that finds the weights—here the associative strengths—that make the average of the squares of all the errors as close to zero as possible. It is a “curve-fitting,” or regression, algorithm that is widely used in engineering and scientific applications (see Section 9.4).³

The Rescorla–Wagner model was very influential in the history of animal learning theory because it showed that a “mechanistic” theory could account for the main facts about blocking without resorting to more complex cognitive theories involving, for example, an animal's explicit recognition that another stimulus component had been added and then scanning its short-term memory backward to reassess the predictive relationships involving the US. The Rescorla–Wagner model showed how traditional contiguity theories of conditioning—that temporal contiguity of stimuli was a necessary and sufficient condition for learning—could be adjusted in a simple way to account for blocking (Moore and Schmajuk, 2008).

The Rescorla–Wagner model provides a simple account of blocking and some other features of classical conditioning, but it is not a complete or perfect model of classical

³The only differences between the LMS rule and the Rescorla–Wagner model are that for LMS the input vectors \mathbf{x}_t can have any real numbers as components, and—at least in the simplest version of the LMS rule—the step-size parameter α does not depend on the input vector or the identity of the stimulus setting the prediction target.

conditioning. Different ideas account for a variety of other observed effects, and progress is still being made toward understanding the many subtleties of classical conditioning. The TD model, which we describe next, though also not a complete or perfect model model of classical conditioning, extends the Rescorla–Wagner model to address how within-trial and between-trial timing relationships among stimuli can influence learning and how higher-order conditioning might arise.

14.2.3 The TD Model

The TD model is a *real-time* model, as opposed to a trial-level model like the Rescorla–Wagner model. A single step t in the Rescorla–Wagner model represents an entire conditioning trial. The model does not apply to details about what happens during the time a trial is taking place, or what might happen between trials. Within each trial an animal might experience various stimuli whose onsets occur at particular times and that have particular durations. These timing relationships strongly influence learning. The Rescorla–Wagner model also does not include a mechanism for higher-order conditioning, whereas for the TD model, higher-order conditioning is a natural consequence of the bootstrapping idea that is at the base of TD algorithms.

To describe the TD model we begin with the formulation of the Rescorla–Wagner model above, but t now labels time steps within or between trials instead of complete trials. Think of the time between t and $t + 1$ as a small time interval, say .01 second, and think of a trial as a sequences of states, one associated with each time step, where the state at step t now represents details of how stimuli are represented at t instead of just a label for the CS components present on a trial. In fact, we can completely abandon the idea of trials. From the point of view of the animal, a trial is just a fragment of its continuing experience interacting with its world. Following our usual view of an agent interacting with its environment, imagine that the animal is experiencing an endless sequence of states s , each represented by a feature vector $\mathbf{x}(s)$. That said, it is still often convenient to refer to trials as fragments of time during which patterns of stimuli repeat in an experiment.

State features are not restricted to describing the external stimuli that an animal experiences; they can describe neural activity patterns that external stimuli produce in an animal’s brain, and these patterns can be history-dependent, meaning that they can be persistent patterns produced by sequences of external stimuli. Of course, we do not know exactly what these neural activity patterns are, but a real-time model like the TD model allows one to explore the consequences on learning of different hypotheses about the internal representations of external stimuli. For these reasons, the TD model does not commit to any particular state representation. In addition, because the TD model includes discounting and eligibility traces that span time intervals between stimuli, the model also makes it possible to explore how discounting and eligibility traces interact with stimulus representations in making predictions about the results of classical conditioning experiments.

Below we describe some of the state representations that have been used with the TD model and some of their implications, but for the moment we stay agnostic about

the representation and just assume that each state s is represented by a feature vector $\mathbf{x}(s) = (x_1(s), x_2(s), \dots, x_n(s))^\top$. Then the aggregate associative strength corresponding to a state s is given by (14.1), the same as for the Rescorla–Wagner model, but the TD model updates the associative strength vector, \mathbf{w} , differently. With t now labeling a time step instead of a complete trial, the TD model governs learning according to this update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t, \quad (14.4)$$

which replaces $\mathbf{x}_t(S_t)$ in the Rescorla–Wagner update (14.2) with \mathbf{z}_t , a vector of eligibility traces, and instead of the δ_t of (14.3), here δ_t is a TD error:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t), \quad (14.5)$$

where γ is a discount factor (between 0 and 1), R_t is the prediction target at time t , and $\hat{v}(S_{t+1}, \mathbf{w}_t)$ and $\hat{v}(S_t, \mathbf{w}_t)$ are aggregate associative strengths at $t+1$ and t as defined by (14.1).

Each component i of the eligibility-trace vector \mathbf{z}_t increments or decrements according to the component $x_i(S_t)$ of the feature vector $\mathbf{x}(S_t)$, and otherwise decays with a rate determined by $\gamma\lambda$:

$$\mathbf{z}_t = \gamma\lambda \mathbf{z}_{t-1} + \mathbf{x}(S_t). \quad (14.6)$$

Here λ is the usual eligibility trace decay parameter.

Note that if $\gamma = 0$, the TD model reduces to the Rescorla–Wagner model with the exceptions that: the meaning of t is different in each case (a trial number for the Rescorla–Wagner model and a time step for the TD model), and in the TD model there is a one-time-step lead in the prediction target R . The TD model is equivalent to the backward view of the semi-gradient TD(λ) algorithm with linear function approximation (Chapter 12), except that R_t in the model does not have to be a reward signal as it does when the TD algorithm is used to learn a value function for policy-improvement.

14.2.4 TD Model Simulations

Real-time conditioning models like the TD model are interesting primarily because they make predictions for a wide range of situations that cannot be represented by trial-level models. These situations involve the timing and durations of conditionable stimuli, the timing of these stimuli in relation to the timing of the US, and the timing and shapes of CRs. For example, the US generally must begin after the onset of a neutral stimulus for conditioning to occur, with the rate and effectiveness of learning depending on the inter-stimulus interval, or ISI, the interval between the onsets of the CS and the US. When CRs appear, they generally begin before the appearance of the US and their temporal profiles change during learning. In conditioning with compound CSs, the component stimuli of the compound CSs may not all begin and end at the same time, sometimes forming what is called a *serial compound* in which the component stimuli occur in a sequence over time. Timing considerations like these make it important to consider how

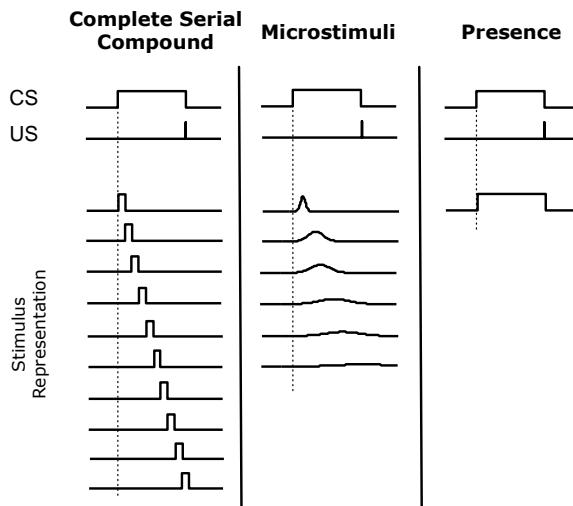


Figure 14.1: Three stimulus representations (in columns) sometimes used with the TD model. Each row represents one element of the stimulus representation. The three representations vary along a temporal generalization gradient, with no generalization between nearby time points in the complete serial compound (left column) and complete generalization between nearby time points in the presence representation (right column). The microstimulus representation occupies a middle ground. The degree of temporal generalization determines the temporal granularity with which US predictions are learned. Adapted with minor changes from *Learning & Behavior, Evaluating the TD Model of Classical Conditioning*, volume 40, 2012, p. 311, E. A. Ludvig, R. S. Sutton, E. J. Kehoe. With permission of Springer.

stimuli are represented, how these representations unfold over time during and between trials, and how they interact with discounting and eligibility traces.

Figure 14.1 shows three of the stimulus representations that have been used in exploring the behavior of the TD model: the *complete serial compound* (CSC), the *microstimulus* (MS), and the *presence* representations (Ludvig, Sutton, and Kehoe, 2012). These representations differ in the degree to which they force generalization among nearby time points during which a stimulus is present.

The simplest of the representations shown in Figure 14.1 is the presence representation in the figure's right column. This representation has a single feature for each component CS present on a trial, where the feature has value 1 whenever that component is present, and 0 otherwise.⁴ The presence representation is not a realistic hypothesis about how stimuli are represented in an animal's brain, but as we describe below, the TD model with this representation can produce many of the timing phenomena seen in classical conditioning.

⁴In our formalism, there is a different state, S_t , for each time step t during a trial, and for a trial in which a compound CS consists of n component CSs of various durations occurring at various times throughout the trial, there is a feature, x_i , for each component CS_i, $i = 1, \dots, n$, where $x_i(S_t) = 1$ for all times t when the CS_i is present, and equals zero otherwise.

For the CSC representation (left column of Figure 14.1), the onset of each external stimulus initiates a sequence of precisely-timed short-duration internal signals that continues until the external stimulus ends.⁵ This is like assuming the animal's nervous system has a clock that keeps precise track of time during stimulus presentations; it is what engineers call a "tapped delay line." Like the presence representation, the CSC representation is unrealistic as a hypothesis about how the brain internally represents stimuli, but Ludvig et al. (2012) call it a "useful fiction" because it can reveal details of how the TD model works when relatively unconstrained by the stimulus representation. The CSC representation is also used in most TD models of dopamine-producing neurons in the brain, a topic we take up in Chapter 15. The CSC representation is often viewed as an essential part of the TD model, although this view is mistaken.

The MS representation (center column of Figure 14.1) is like the CSC representation in that each external stimulus initiates a cascade of internal stimuli, but in this case the internal stimuli—the microstimuli—are not of such limited and non-overlapping form; they are extended over time and overlap. As time elapses from stimulus onset, different sets of microstimuli become more or less active, and each subsequent microstimulus becomes progressively wider in time and reaches a lower maximal level. Of course, there are many MS representations depending on the nature of the microstimuli, and a number of examples of MS representations have been studied in the literature, in some cases along with proposals for how an animal's brain might generate them (see the Bibliographic and Historical Comments at the end of this chapter). MS representations are more realistic than the presence or CSC representations as hypotheses about neural representations of stimuli, and they allow the behavior of the TD model to be related to a broader collection of phenomena observed in animal experiments. In particular, by assuming that cascades of microstimuli are initiated by USs as well as by CSs, and by studying the significant effects on learning of interactions between microstimuli, eligibility traces, and discounting, the TD model is helping to frame hypotheses to account for many of the subtle phenomena of classical conditioning and how an animal's brain might produce them. We say more about this below, particularly in Chapter 15 where we discuss reinforcement learning and neuroscience.

Even with the simple presence representation, however, the TD model produces all the basic properties of classical conditioning that are accounted for by the Rescorla–Wagner model, plus features of conditioning that are beyond the scope of trial-level models. For example, as we have already mentioned, a conspicuous feature of classical conditioning is that the US generally must begin *after* the onset of a neutral stimulus for conditioning to occur, and that after conditioning, the CR begins *before* the appearance of the US. In other words, conditioning generally requires a positive ISI, and the CR generally anticipates the US. How the strength of conditioning (e.g., the percentage of CRs elicited by a CS) depends on the ISI varies substantially across species and response systems, but it typically has the following properties: it is negligible for a zero or negative ISI, i.e., when

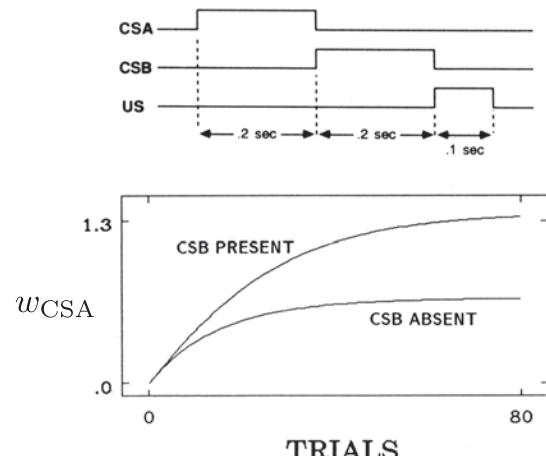
⁵In our formalism, for each CS component CS_i present on a trial, and for each time step t during a trial, there is a separate feature x_i^t , where $x_i^t(S_{t'}) = 1$ if $t = t'$ for any t' at which CS_i is present, and equals 0 otherwise. This is different from the CSC representation in Sutton and Barto (1990) in which there are the same distinct features for each time step but no reference to external stimuli; hence the name complete serial compound.

the US onset occurs simultaneously with, or earlier than, the CS onset (although research has found that associative strengths sometimes increase slightly or become negative with negative ISIs); it increases to a maximum at a positive ISI where conditioning is most effective; and it then decreases to zero after an interval that varies widely with response systems. The precise shape of this dependency for the TD model depends on the values of its parameters and details of the stimulus representation, but these basic features of ISI-dependency are core properties of the TD model.

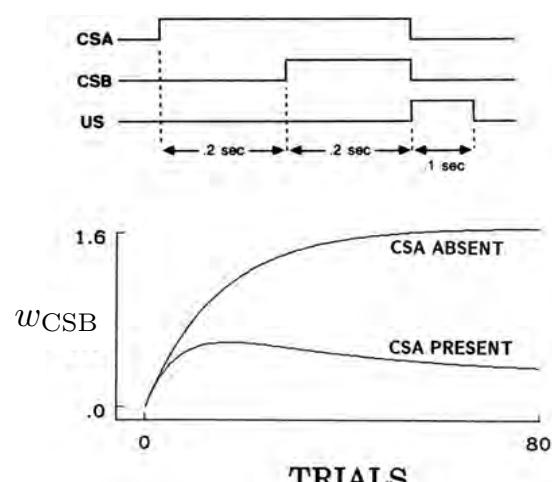
One of the theoretical issues arising with serial-compound conditioning, that is, conditioning with a compound CS whose components occur in a sequence, concerns the facilitation of remote associations. It has been found that if the empty trace interval between a first CS (CSA) and the US is filled with a second CS (CSB) to form a serial-compound stimulus, then conditioning to CSA is facilitated. Shown to the right is the behavior of the TD model with the presence representation in a simulation of such an experiment whose timing details are shown above. Consistent with the experimental results (Kehoe, 1982), the model shows facilitation of both the rate of conditioning and the asymptotic level of conditioning of the first CS due to the presence of the second CS.

A well-known demonstration of the effects on conditioning of temporal relationships among stimuli within a trial is an experiment by Egger and Miller (1962) that involved two overlapping CSs in a delay configuration as shown to the right (top). Although CSB was in a better temporal relationship with the US, the presence of CSA substantially reduced conditioning to CSB as compared to controls in which CSA was absent. Directly to the right is shown the same result being generated by the TD model in a simulation of this experiment with the presence representation.

The TD model accounts for blocking because it is an error-correcting learning



Facilitation of remote associations in the TD model



The Egger-Miller effect in the TD model

rule like the Rescorla-Wagner model. Beyond accounting for basic blocking results, however, the TD model predicts (with the presence representation and more complex representations as well) that blocking is reversed if the blocked stimulus is moved earlier in time (like CSA in the diagram to the right) so that its onset occurs before the onset of the blocking stimulus. This feature of the TD model's behavior deserves attention because it had not been observed at the time of the model's introduction. Recall that in blocking, if an animal has already learned that one CS predicts a US, then learning that a newly-added second CS also predicts the US is much reduced, i.e., is blocked. But if the newly-added second CS begins earlier than the pretrained CS, then—according to the TD model—learning to the newly-added CS is not blocked. In fact, as training continues and the newly-added CS gains associative strength, the pretrained CS loses associative strength. The behavior of the TD model under these conditions is shown in the lower part of Figure 14.2. This simulation experiment differed from the Egger-Miller experiment (bottom of the preceding page) in that the shorter CS with the later onset was given prior training until it was fully associated with the US. This surprising prediction led Kehoe, Schreurs, and Graham (1987) to conduct the experiment using the well-studied rabbit nictitating membrane preparation. Their results confirmed the model's prediction, and they noted that non-TD models have considerable difficulty explaining their data.

With the TD model, an earlier predictive stimulus takes precedence over a later predictive stimulus because, like all the prediction methods described in this book, the TD model is based on the backing-up or bootstrapping idea: updates to associative strengths shift the strengths at a particular state toward the strength at later states. Another consequence of bootstrapping is that the TD model provides an account of higher-order conditioning, a feature of classical conditioning that is beyond the scope of the Rescorla-Wagner and similar models. As we described above, higher-order conditioning is the phenomenon in which a previously-conditioned CS can act as a US in conditioning another initially neutral stimulus. Figure 14.3 shows the behavior of the TD model (again with the presence representation) in a higher-order conditioning experiment—in this case it is second-order conditioning. In the first phase (not shown in the figure), CSB is trained to predict a US so that its associative strength increases, here to 1.65. In the second phase, CSA is paired with CSB in the absence of the US, in the sequential arrangement shown at the top of the figure. CSA acquires associative strength even though it is never paired with the US. With continued training, CSA's associative strength reaches a peak and then decreases because the associative strength of CSB, the secondary reinforcer, decreases so that it loses its ability to provide secondary reinforcement. CSB's associative

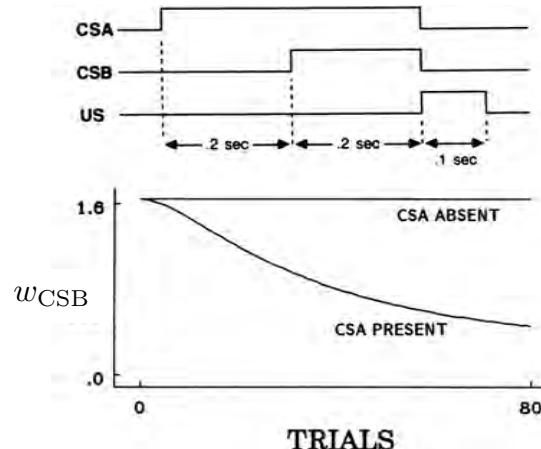


Figure 14.2: Temporal primacy overriding blocking in the TD model.

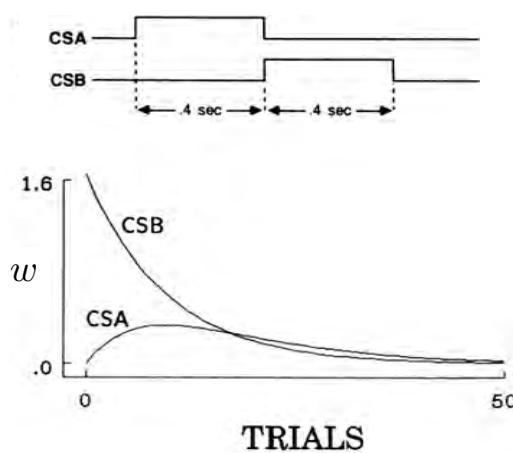


Figure 14.3: Second-order conditioning with the TD model.

from $\hat{v}(S_t, \mathbf{w}_t)$, making δ_t non-zero (a temporal difference). This difference has the same status as R_{t+1} in (14.5), implying that as far as learning is concerned there is no difference between a temporal difference and the occurrence of a US. In fact, this feature of the TD algorithm is one of the major reasons for its development, which we now understand through its connection to dynamic programming as described in Chapter 6. Bootstrapping values is intimately related to second-order, and higher-order, conditioning.

In the examples of the TD model's behavior described above, we examined only the changes in the associative strengths of the CS components; we did not look at what the model predicts about properties of an animal's conditioned responses (CRs): their timing, shape, and how they develop over conditioning trials. These properties depend on the species, the response system being observed, and parameters of the conditioning trials, but in many experiments with different animals and different response systems, the magnitude of the CR, or the probability of a CR, increases as the expected time of the US approaches. For example, in classical conditioning of a rabbit's nictitating membrane response that we mentioned above, over conditioning trials the delay from CS onset to when the nictitating membrane begins to move across the eye decreases over trials, and the amplitude of this anticipatory closure gradually increases over the interval between the CS and the US until the membrane reaches maximal closure at the expected time of the US. The timing and shape of this CR is critical to its adaptive significance—covering the eye too early reduces vision (even though the nictitating membrane is translucent), while covering it too late is of little protective value. Capturing CR features like these is challenging for models of classical conditioning.

The TD model does not include as part of its definition any mechanism for translating the time course of the US prediction, $\hat{v}(S_t, \mathbf{w}_t)$, into a profile that can be compared

strength decreases because the US does not occur in these higher-order conditioning trials. These are *extinction trials* for CSB because its predictive relationship to the US is disrupted so that its ability to act as a reinforcer decreases. This same pattern is seen in animal experiments. This extinction of conditioned reinforcement in higher-order conditioning trials makes it difficult to demonstrate higher-order conditioning unless the original predictive relationships are periodically refreshed by occasionally inserting first-order trials.

The TD model produces an analog of second- and higher-order conditioning because $\gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)$ appears in the TD error δ_t (14.5). Due to the first phase of learning, $\gamma\hat{v}(S_{t+1}, \mathbf{w}_t)$ may differ

with the properties of an animal's CR. The simplest choice is to let the time course of a simulated CR equal the time course of the US prediction. In this case, features of simulated CRs and how they change over trials depend only on the stimulus representation chosen and the values of the model's parameters α , γ , and λ .

Figure 14.4 shows the time courses of US predictions at different points during learning with the three representations shown in Figure 14.1. For these simulations the US occurred 25 time steps after the onset of the CS, and $\alpha = .05$, $\lambda = .95$ and $\gamma = .97$. With the CSC representation (Figure 14.4 left), the curve of the US prediction formed by the TD model increases exponentially throughout the interval between the CS and the US until it reaches a maximum exactly when the US occurs (at time step 25). This exponential increase is the result of discounting in the TD model learning rule. With the presence representation (Figure 14.4 middle), the US prediction is nearly constant while the stimulus is present because there is only one weight, or associative strength, to be learned for each stimulus. Consequently, the TD model with the presence representation cannot recreate many features of CR timing. With an MS representation (Figure 14.4 right), the development of the TD model's US prediction is more complicated. After 200 trials the prediction's profile is a reasonable approximation of the US prediction curve produced with the CSC representation.

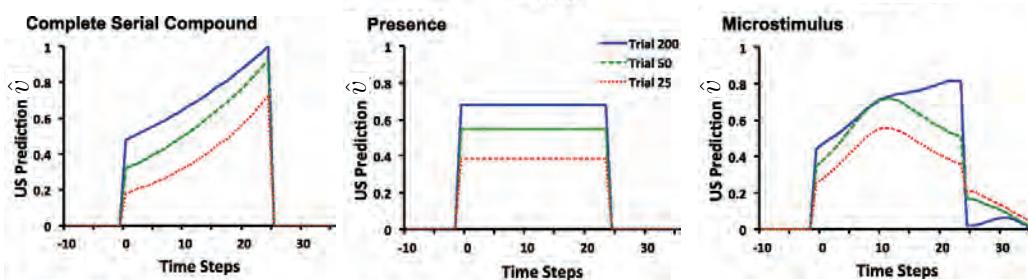


Figure 14.4: Time course of US prediction over the course of acquisition for the TD model with three different stimulus representations. Left: With the complete serial compound (CSC), the US prediction increases exponentially through the interval, peaking at the time of the US. At asymptote (trial 200), the US prediction peaks at the US intensity (1 in these simulations). Middle: With the presence representation, the US prediction converges to an almost constant level. This constant level is determined by the US intensity and the length of the CS-US interval. Right: With the microstimulus representation, at asymptote, the TD model approximates the exponentially increasing time course depicted with the CSC through a linear combination of the different microstimuli. Adapted with minor changes from *Learning & Behavior*, Evaluating the TD Model of Classical Conditioning, volume 40, 2012, E. A. Ludvig, R. S. Sutton, E. J. Kehoe. With permission of Springer.

The US prediction curves shown in Figure 14.4 were not intended to precisely match profiles of CRs as they develop during conditioning in any particular animal experiment, but they illustrate the strong influence that the stimulus representation has on predictions derived from the TD model. Further, although we can only mention it here, how the

stimulus representation interacts with discounting and eligibility traces is important in determining properties of the US prediction profiles produced by the TD model. Another dimension beyond what we can discuss here is the influence of different response-generation mechanisms that translate US predictions into CR profiles; the profiles shown in Figure 14.4 are “raw” US prediction profiles. Even without any special assumption about how an animal’s brain might produce overt responses from US predictions, however, the profiles in Figure 14.4 for the CSC and MS representations increase as the time of the US approaches and reach a maximum at the time of the US, as is seen in many animal conditioning experiments.

The TD model, when combined with particular stimulus representations and response-generation mechanisms, is able to account for a surprisingly wide range of phenomena observed in animal classical conditioning experiments, but it is far from being a perfect model. To generate other details of classical conditioning the model needs to be extended, perhaps by adding model-based elements and mechanisms for adaptively altering some of its parameters. Other approaches to modeling classical conditioning depart significantly from the Rescorla–Wagner-style error-correction process. Bayesian models, for example, work within a probabilistic framework in which experience revises probability estimates. All of these models usefully contribute to our understanding of classical conditioning.

Perhaps the most notable feature of the TD model is that it is based on a theory—the theory we have described in this book—that suggests an account of what an animal’s nervous system is *trying to do* while undergoing conditioning: it is trying to form accurate *long-term predictions*, consistent with the limitations imposed by the way stimuli are represented and how the nervous system works. In other words, it suggests a *normative account* of classical conditioning in which long-term, instead of immediate, prediction is a key feature.

The development of the TD model of classical conditioning is one instance in which the explicit goal was to model some of the details of animal learning behavior. In addition to its standing as an *algorithm*, then, TD learning is also the basis of this *model* of aspects of biological learning. As we discuss in Chapter 15, TD learning has also turned out to underlie an influential model of the activity of neurons that produce dopamine, a chemical in the brain of mammals that is deeply involved in reward processing. These are instances in which reinforcement learning theory makes detailed contact with animal behavioral and neural data.

We now turn to considering correspondences between reinforcement learning and animal behavior in instrumental conditioning experiments, the other major type of laboratory experiment studied by animal learning psychologists.

14.3 Instrumental Conditioning

In *instrumental conditioning* experiments learning depends on the consequences of behavior: the delivery of a reinforcing stimulus is contingent on what the animal does. In classical conditioning experiments, in contrast, the reinforcing stimulus—the US—is delivered independently of the animal’s behavior. Instrumental conditioning is usually considered to be the same as *operant conditioning*, the term B. F. Skinner (1938, 1963)

introduced for experiments with behavior-contingent reinforcement, though the experiments and theories of those who use these two terms differ in a number of ways, some of which we touch on below. We will exclusively use the term instrumental conditioning for experiments in which reinforcement is contingent upon behavior. The roots of instrumental conditioning go back to experiments performed by the American psychologist Edward Thorndike one hundred years before publication of the first edition of this book.

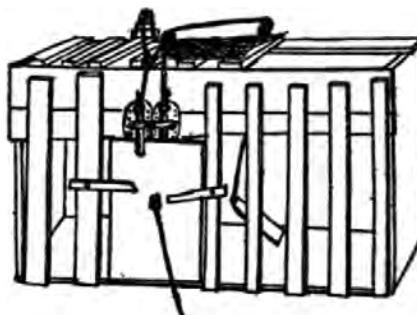
Thorndike observed the behavior of cats when they were placed in “puzzle boxes,” such as the one at the right, from which they could escape by appropriate actions. For example, a cat could open the door of one box by performing a sequence of three separate actions: depressing a platform at the back of the box, pulling a string by clawing at it, and pushing a bar up or down. When first placed in a puzzle box, with food visible outside, all but a few of Thorndike’s cats displayed “evident signs of discomfort” and extraordinarily vigorous activity “to strive instinctively to escape from confinement” (Thorndike, 1898).

In experiments with different cats and boxes with different escape mechanisms, Thorndike recorded the amounts of time each cat took to escape over multiple experiences in each box. He observed that the time almost invariably decreased with successive experiences, for example, from 300 seconds to 6 or 7 seconds. He described cats’ behavior in a puzzle box like this:

The cat that is clawing all over the box in her impulsive struggle will probably claw the string or loop or button so as to open the door. And gradually all the other non-successful impulses will be stamped out and the particular impulse leading to the successful act will be stamped in by the resulting pleasure, until, after many trials, the cat will, when put in the box, immediately claw the button or loop in a definite way. (Thorndike 1898, p. 13)

These and other experiments (some with dogs, chicks, monkeys, and even fish) led Thorndike to formulate a number of “laws” of learning, the most influential being the *Law of Effect*. This law describes what is generally known as learning by trial and error. As we mentioned in Chapter 1, many aspects of the Law of Effect have generated controversy, and its details have been modified over the years. Still the law—in one form or another—expresses an enduring principle of learning.

Essential features of reinforcement learning algorithms correspond to features of animal learning described by the Law of Effect. First, reinforcement learning algorithms are *selectional*, meaning that they try alternatives and select among them by comparing their consequences. Second, reinforcement learning algorithms are *associative*, meaning that the alternatives found by selection are associated with particular situations, or states, to form the agent’s policy. Like learning described by the Law of Effect, reinforcement



One of Thorndike’s puzzle boxes.

Reprinted from Thorndike, *Animal Intelligence: An Experimental Study of the Associative Processes in Animals*, *The Psychological Review, Series of Monograph Supplements II(4)*, Macmillan, New York, 1898.

learning is not just the process of *finding* actions that produce a lot of reward, but also of *connecting* these actions to situations or states. Thorndike used the phrase learning by “selecting and connecting” (Hilgard, 1956). Natural selection in evolution is a prime example of a selectional process, but it is not associative (at least as it is commonly understood); supervised learning is associative, but it is not selectional because it relies on instructions that directly tell the agent how to change its behavior.

In computational terms, the Law of Effect describes an elementary way of combining *search* and *memory*: search in the form of trying and selecting among many actions in each situation, and memory in the form of associations linking situations with the actions found—so far—to work best in those situations. Search and memory are essential components of all reinforcement learning algorithms, whether memory takes the form of an agent’s policy, value function, or environment model.

A reinforcement learning algorithm’s need to search means that it has to explore in some way. Animals clearly explore as well, and early animal learning researchers disagreed about the degree of guidance an animal uses in selecting its actions in situations like Thorndike’s puzzle boxes. Are actions the result of “absolutely random, blind groping” (Woodworth, 1938, p. 777), or is there some degree of guidance, either from prior learning, reasoning, or other means? Although some thinkers, including Thorndike, seem to have taken the former position, others favored more deliberate exploration. Reinforcement learning algorithms allow wide latitude for how much guidance an agent can employ in selecting actions. The forms of exploration we have used in the algorithms presented in this book, such as ε -greedy and upper-confidence-bound action selection, are merely among the simplest. More sophisticated methods are possible, with the only stipulation being that there has to be *some* form of exploration for the algorithms to work effectively.

The feature of our treatment of reinforcement learning allowing the set of actions available at any time to depend on the environment’s current state echoes something Thorndike observed in his cats’ puzzle-box behaviors. The cats selected actions from those that they instinctively perform in their current situation, which Thorndike called their “instinctual impulses.” First placed in a puzzle box, a cat instinctively scratches, claws, and bites with great energy: a cat’s instinctual responses to finding itself in a confined space. Successful actions are selected from these and not from every possible action or activity. This is like the feature of our formalism where the action selected from a state s belongs to a set of admissible actions, $\mathcal{A}(s)$. Specifying these sets is an important aspect of reinforcement learning because it can radically simplify learning. They are like an animal’s instinctual impulses. On the other hand, Thorndike’s cats might have been exploring according to an instinctual context-specific *ordering* over actions rather than by just selecting from a set of instinctual impulses. This is another way to make reinforcement learning easier.

Among the most prominent animal learning researchers influenced by the Law of Effect were Clark Hull (e.g., Hull, 1943) and B. F. Skinner (e.g., Skinner, 1938). At the center of their research was the idea of selecting behavior on the basis of its consequences. Reinforcement learning has features in common with Hull’s theory, which included eligibility-like mechanisms and secondary reinforcement to account for the ability to learn when there is a significant time interval between an action and the consequent reinforcing

stimulus (see Section 14.4). Randomness also played a role in Hull's theory through what he called "behavioral oscillation" to introduce exploratory behavior.

Skinner did not fully subscribe to the memory aspect of the Law of Effect. Being averse to the idea of associative linkages, he instead emphasized selection from spontaneously-emitted behavior. He introduced the term "operant" to emphasize the key role of an action's effects on an animal's environment. Unlike the experiments of Thorndike and others, which consisted of sequences of separate trials, Skinner's operant conditioning experiments allowed animal subjects to behave for extended periods of time without interruption. He invented the operant conditioning chamber, now called a "Skinner box," the most basic version of which contains a lever or key that an animal can press to obtain a reward, such as food or water, which would be delivered according to a well-defined rule, called a reinforcement schedule. By recording the cumulative number of lever presses as a function of time, Skinner and his followers could investigate the effect of different reinforcement schedules on the animal's rate of lever-pressing. Modeling results from experiments like these using the reinforcement learning principles we present in this book is not well developed, but we mention some exceptions in the Bibliographic and Historical Remarks section at the end of this chapter.

Another of Skinner's contributions resulted from his recognition of the effectiveness of training an animal by reinforcing successive approximations of the desired behavior, a process he called *shaping*. Although this technique had been used by others, including Skinner himself, its significance was impressed upon him when he and colleagues were attempting to train a pigeon to bowl by swiping a wooden ball with its beak. After waiting for a long time without seeing any swipe that they could reinforce, they

... decided to reinforce any response that had the slightest resemblance to a swipe—perhaps, at first, merely the behavior of looking at the ball—and then to select responses which more closely approximated the final form. The result amazed us. In a few minutes, the ball was caroming off the walls of the box as if the pigeon had been a champion squash player. (Skinner, 1958, p. 94)

Not only did the pigeon learn a behavior that is unusual for pigeons, it learned quickly through an interactive process in which its behavior and the reinforcement contingencies changed in response to each other. Skinner compared the process of altering reinforcement contingencies to the work of a sculptor shaping clay into a desired form. Shaping is a powerful technique for computational reinforcement learning systems as well. When it is difficult for an agent to receive any non-zero reward signal at all, either due to sparseness of rewarding situations or their inaccessibility given initial behavior, starting with an easier problem and incrementally increasing its difficulty as the agent learns can be an effective, and sometimes indispensable, strategy.

A concept from psychology that is especially relevant in the context of instrumental conditioning is *motivation*, which refers to processes that influence the direction and strength, or vigor, of behavior. Thorndike's cats, for example, were motivated to escape from puzzle boxes because they wanted the food that was sitting just outside. Obtaining this goal was rewarding to them and reinforced the actions allowing them to escape. It is difficult to link the concept of motivation, which has many dimensions, in a precise

way to reinforcement learning’s computational perspective, but there are clear links with some of its dimensions.

In one sense, a reinforcement learning agent’s reward signal is at the base of its motivation: the agent is motivated to maximize the total reward it receives over the long run. A key facet of motivation, then, is what makes an agent’s experience rewarding. In reinforcement learning, reward signals depend on the state of the reinforcement learning agent’s environment and the agent’s actions. Further, as pointed out in Chapter 1, the state of the agent’s environment not only includes information about what is external to the machine, like an organism or a robot, that houses the agent, but also what is internal to this machine. Some internal state components correspond to what psychologists call an animal’s *motivational state*, which influences what is rewarding to the animal. For example, an animal will be more rewarded by eating when it is hungry than when it has just finished a satisfying meal. The concept of state dependence is broad enough to allow for many types of modulating influences on the generation of reward signals.

Value functions provide a further link to psychologists’ concept of motivation. If the most basic motive for selecting an action is to obtain as much reward as possible, for a reinforcement learning agent that selects actions using a value function, a more proximal motive is to *ascend the gradient of its value function*, that is, to select actions expected to lead to the most highly-valued next states (or what is essentially the same thing, to select actions with the greatest action-values). For these agents, value functions are the main driving force determining the direction of their behavior.

Another dimension of motivation is that an animal’s motivational state not only influences learning, but also influences the strength, or vigor, of the animal’s behavior after learning. For example, after learning to find food in the goal box of a maze, a hungry rat will run faster to the goal box than one that is not hungry. This aspect of motivation does not link so cleanly to the reinforcement learning framework we present here, but in the Bibliographical and Historical Remarks section at the end of this chapter we cite several publications that propose theories of behavioral vigor based on reinforcement learning.

We turn now to the subject of learning when reinforcing stimuli occur well after the events they reinforce. The mechanisms used by reinforcement learning algorithms to enable learning with delayed reinforcement—eligibility traces and TD learning—closely correspond to psychologists’ hypotheses about how animals can learn under these conditions.

14.4 Delayed Reinforcement

The Law of Effect requires a backward effect on connections, and some early critics of the law could not conceive of how the present could affect something that was in the past. This concern was amplified by the fact that learning can even occur when there is a considerable delay between an action and the consequent reward or penalty. Similarly, in classical conditioning, learning can occur when US onset follows CS offset by a non-negligible time interval. We call this the problem of delayed reinforcement, which is related to what Minsky (1961) called the “credit-assignment problem for learning systems”: how do you

distribute credit for success among the many decisions that may have been involved in producing it? The reinforcement learning algorithms presented in this book include two basic mechanisms for addressing this problem. The first is the use of eligibility traces, and the second is the use of TD methods to learn value functions that provide nearly immediate evaluations of actions (in tasks like instrumental conditioning experiments) or that provide immediate prediction targets (in tasks like classical conditioning experiments). Both of these methods correspond to similar mechanisms proposed in theories of animal learning.

Pavlov (1927) pointed out that every stimulus must leave a trace in the nervous system that persists for some time after the stimulus ends, and he proposed that stimulus traces make learning possible when there is a temporal gap between the CS offset and the US onset. To this day, conditioning under these conditions is called *trace conditioning* (page 344). Assuming a trace of the CS remains when the US arrives, learning occurs through the simultaneous presence of the trace and the US. We discuss some proposals for trace mechanisms in the nervous system in Chapter 15.

Stimulus traces were also proposed as a means for bridging the time interval between actions and consequent rewards or penalties in instrumental conditioning. In Hull's influential learning theory, for example, "molar stimulus traces" accounted for what he called an animal's *goal gradient*, a description of how the maximum strength of an instrumentally-conditioned response decreases with increasing delay of reinforcement (Hull, 1932, 1943). Hull hypothesized that an animal's actions leave internal stimuli whose traces decay exponentially as functions of time since an action was taken. Looking at the animal learning data available at the time, he hypothesized that the traces effectively reach zero after 30 to 40 seconds.

The eligibility traces used in the algorithms described in this book are like Hull's traces: they are decaying traces of past state visitations, or of past state-action pairs. Eligibility traces were introduced by Klopff (1972) in his neuronal theory in which they are temporally-extended traces of past activity at synapses, the connections between neurons. Klopff's traces are more complex than the exponentially-decaying traces our algorithms use, and we discuss this more when we take up his theory in Section 15.9.

To account for goal gradients that extend over longer time periods than spanned by stimulus traces, Hull (1943) proposed that longer gradients result from conditioned reinforcement passing backwards from the goal, a process acting in conjunction with his molar stimulus traces. Animal experiments showed that if conditions favor the development of conditioned reinforcement during a delay period, learning does not decrease with increased delay as much as it does under conditions that obstruct secondary reinforcement. Conditioned reinforcement is favored if there are stimuli that regularly occur during the delay interval. Then it is as if reward is not actually delayed because there is more immediate conditioned reinforcement. Hull therefore envisioned that there is a primary gradient based on the delay of the primary reinforcement mediated by stimulus traces, and that this is progressively modified, and lengthened, by conditioned reinforcement.

Algorithms presented in this book that use both eligibility traces and value functions to enable learning with delayed reinforcement correspond to Hull's hypothesis about how animals are able to learn under these conditions. The actor-critic architecture discussed

in Sections 13.5, 15.7, and 15.8 illustrates this correspondence most clearly. The critic uses a TD algorithm to learn a value function associated with the system's current behavior, that is, to predict the current policy's return. The actor updates the current policy based on the critic's predictions, or more exactly, on changes in the critic's predictions. The TD error produced by the critic acts as a conditioned reinforcement signal for the actor, providing an immediate evaluation of performance even when the primary reward signal itself is considerably delayed. Algorithms that estimate action-value functions, such as Q-learning and Sarsa, similarly use TD learning principles to enable learning with delayed reinforcement by means of conditioned reinforcement. The close parallel between TD learning and the activity of dopamine producing neurons that we discuss in Chapter 15 lends additional support to links between reinforcement learning algorithms and this aspect of Hull's learning theory.

14.5 Cognitive Maps

Model-based reinforcement learning algorithms use environment models that have elements in common with what psychologists call *cognitive maps*. Recall from our discussion of planning and learning in Chapter 8 that by an environment model we mean anything an agent can use to predict how its environment will respond to its actions in terms of state transitions and rewards, and by planning we mean any process that computes a policy from such a model. Environment models consist of two parts: the state-transition part encodes knowledge about the effect of actions on state changes, and the reward-model part encodes knowledge about the reward signals expected for each state or each state-action pair. A model-based algorithm selects actions by using a model to predict the consequences of possible courses of action in terms of future states and the reward signals expected to arise from those states. The simplest kind of planning is to compare the predicted consequences of collections of "imagined" sequences of decisions.

Questions about whether or not animals use environment models, and if so, what are the models like and how are they learned, have played influential roles in the history of animal learning research. Some researchers challenged the then-prevailing stimulus-response (S-R) view of learning and behavior, which corresponds to the simplest model-free way of learning policies, by demonstrating *latent learning*. In the earliest latent learning experiment, two groups of rats were run in a maze. For the experimental group, there was no reward during the first stage of the experiment, but food was suddenly introduced into the goal box of the maze at the start of the second stage. For the control group, food was in the goal box throughout both stages. The question was whether or not rats in the experimental group would have learned anything during the first stage in the absence of food reward. Although the experimental rats did not *appear* to learn much during the first, unrewarded, stage, as soon as they discovered the food that was introduced in the second stage, they rapidly caught up with the rats in the control group. It was concluded that "during the non-reward period, the rats [in the experimental group] were developing a latent learning of the maze which they were able to utilize as soon as reward was introduced" (Blodgett, 1929).

Latent learning is most closely associated with the psychologist Edward Tolman, who interpreted this result, and others like it, as showing that animals could learn a “cognitive map of the environment” in the absence of rewards or penalties, and that they could use the map later when they were motivated to reach a goal (Tolman, 1948). A cognitive map could also allow a rat to plan a route to the goal that was different from the route the rat had used in its initial exploration. Explanations of results like these led to the enduring controversy lying at the heart of the behaviorist/cognitive dichotomy in psychology. In modern terms, cognitive maps are not restricted to models of spatial layouts but are more generally environment models, or models of an animal’s “task space” (e.g., Wilson, Takahashi, Schoenbaum, and Niv, 2014). The cognitive map explanation of latent learning experiments is analogous to the claim that animals use model-based algorithms, and that environment models can be learned even without explicit rewards or penalties. Models are then used for planning when the animal is motivated by the appearance of rewards or penalties.

Tolman’s account of how animals learn cognitive maps was that they learn stimulus-stimulus, or $S-S$, associations by experiencing successions of stimuli as they explore an environment. In psychology this is called *expectancy theory*: given $S-S$ associations, the occurrence of a stimulus generates an expectation about the stimulus to come next. This is much like what control engineers call *system identification*, in which a model of a system with unknown dynamics is learned from labeled training examples. In the simplest discrete-time versions, training examples are $S-S'$ pairs, where S is a state and S' , the subsequent state, is the label. When S is observed, the model creates the “expectation” that S' will be observed next. Models more useful for planning involve actions as well, so that examples look like $SA-S'$, where S' is expected when action A is executed in state S . It is also useful to learn how the environment generates rewards. In this case, examples are of the form $S-R$ or $SA-R$, where R is a reward signal associated with S or the SA pair. These are all forms of supervised learning by which an agent can acquire cognitive-like maps whether or not it receives any non-zero reward signals while exploring its environment.

14.6 Habitual and Goal-directed Behavior

The distinction between model-free and model-based reinforcement learning algorithms corresponds to the distinction psychologists make between *habitual* and *goal-directed* control of learned behavioral patterns. Habits are behavior patterns triggered by appropriate stimuli and then performed more-or-less automatically. Goal-directed behavior, according to how psychologists use the phrase, is purposeful in the sense that it is controlled by knowledge of the value of goals and the relationship between actions and their consequences. Habits are sometimes said to be controlled by antecedent stimuli, whereas goal-directed behavior is said to be controlled by its consequences (Dickinson, 1980, 1985). Goal-directed control has the advantage that it can rapidly change an animal’s behavior when the environment changes its way of reacting to the animal’s actions. While habitual behavior responds quickly to input from an accustomed environment, it is unable

to quickly adjust to changes in the environment. The development of goal-directed behavioral control was likely a major advance in the evolution of animal intelligence.

Figure 14.5 illustrates the difference between model-free and model-based decision strategies in a hypothetical task in which a rat has to navigate a maze that has distinctive goal boxes, each delivering an associated reward of the magnitude shown (Figure 14.5 top). Starting at S_1 , the rat has to first select left (L) or right (R) and then has to select L or R again at S_2 or S_3 to reach one of the goal boxes. The goal boxes are the terminal states of each episode of the rat's episodic task. A model-free strategy (Figure 14.5 lower left) relies on stored values for state-action pairs. These action values are estimates of the highest return the rat can expect for each action taken from each (nonterminal) state. They are obtained over many trials of running the maze from start to finish. When the action values have become good enough estimates of the optimal returns, the rat just has

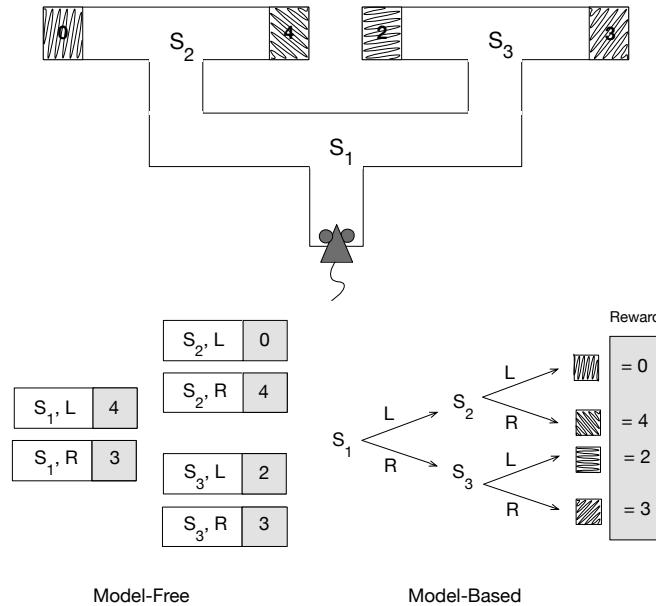


Figure 14.5: Model-based and model-free strategies to solve a hypothetical sequential action-selection problem. Top: a rat navigates a maze with distinctive goal boxes, each associated with a reward having the value shown. Lower left: a model-free strategy relies on stored action values for all the state-action pairs obtained over many learning trials. To make decisions the rat just has to select at each state the action with the largest action value for that state. Lower right: in a model-based strategy, the rat learns an environment model, consisting of knowledge of state-action-next-state transitions and a reward model consisting of knowledge of the reward associated with each distinctive goal box. The rat can decide which way to turn at each state by using the model to simulate sequences of action choices to find a path yielding the highest return. Adapted from *Trends in Cognitive Science*, volume 10, number 8, Y. Niv, D. Joel, and P. Dayan, A Normative Perspective on Motivation, p. 376, 2006, with permission from Elsevier.

to select at each state the action with the largest action value in order to make optimal decisions. In this case, when the action-value estimates become accurate enough, the rat selects L from S_1 and R from S_2 to obtain the maximum return of 4. A different model-free strategy might simply rely on a cached policy instead of action values, making direct links from S_1 to L and from S_2 to R. In neither of these strategies do decisions rely on an environment model. There is no need to consult a state-transition model, and no connection is required between the features of the goal boxes and the rewards they deliver.

Figure 14.5 (lower right) illustrates a model-based strategy. It uses an environment model consisting of a state-transition model and a reward model. The state-transition model is shown as a decision tree, and the reward model associates the distinctive features of the goal boxes with the rewards to be found in each. (The rewards associated with states S_1 , S_2 , and S_3 are also part of the reward model, but here they are zero and are not shown.) A model-based agent can decide which way to turn at each state by using the model to simulate sequences of action choices to find a path yielding the highest return. In this case the return is the reward obtained from the outcome at the end of the path. Here, with a sufficiently accurate model, the rat would select L and then R to obtain a return of 4. Comparing the predicted returns of simulated paths is a simple form of planning, which can be done in a variety of ways as discussed in Chapter 8.

When the environment of a model-free agent changes the way it reacts to the agent's actions, the agent has to acquire new experience in the changed environment during which it can update its policy and/or value function. In the model-free strategy shown in Figure 14.5 (lower left), for example, if one of the goal boxes were to somehow shift to delivering a different reward, the rat would have to traverse the maze, possibly many times, to experience the new reward upon reaching that goal box, all the while updating either its policy or its action-value function (or both) based on this experience. The key point is that for a model-free agent to change the action its policy specifies for a state, or to change an action value associated with a state, it has to move to that state, act from it, possibly many times, and experience the consequences of its actions.

A model-based agent can accommodate changes in its environment without this kind of 'personal experience' with the states and actions affected by the change. A change in its model automatically (through planning) changes its policy. Planning can determine the consequences of changes in the environment that have never been linked together in the agent's own experience. For example, again referring to the maze task of Figure 14.5, imagine that a rat with a previously learned transition and reward model is placed directly in the goal box to the right of S_2 to find that the reward available there now has value 1 instead of 4. The rat's reward model will change even though the action choices required to find that goal box in the maze were not involved. The planning process will bring knowledge of the new reward to bear on maze running without the need for additional experience in the maze; in this case changing the policy to right turns at both S_1 and S_3 to obtain a return of 3.

Exactly this logic is the basis of *outcome-devaluation experiments* with animals. Results from these experiments provide insight into whether an animal has learned a habit or if its behavior is under goal-directed control. Outcome-devaluation experiments are like latent-learning experiments in that the reward changes from one stage to the next. After

an initial rewarded stage of learning, the reward value of an outcome is changed, including being shifted to zero or even to a negative value.

An early important experiment of this type was conducted by Adams and Dickinson (1981). They trained rats via instrumental conditioning until the rats energetically pressed a lever for sucrose pellets in a training chamber. The rats were then placed in the same chamber with the lever retracted and allowed non-contingent food, meaning that pellets were made available to them independently of their actions. After 15-minutes of this free-access to the pellets, rats in one group were injected with the nausea-inducing poison lithium chloride. This was repeated for three sessions, in the last of which none of the injected rats consumed any of the non-contingent pellets, indicating that the reward value of the pellets had been decreased—the pellets had been devalued. In the next stage taking place a day later, the rats were again placed in the chamber and given a session of extinction training, meaning that the response lever was back in place but disconnected from the pellet dispenser so that pressing it did not release pellets. The question was whether the rats that had the reward value of the pellets decreased would lever-press less than rats that did not have the reward value of the pellets decreased, even without experiencing the devalued reward as a result of lever-pressing. It turned out that the injected rats had significantly lower response rates than the non-injected rats *right from the start of the extinction trials*.

Adams and Dickinson concluded that the injected rats associated lever pressing with consequent nausea by means of a cognitive map linking lever pressing to pellets, and pellets to nausea. Hence, in the extinction trials, the rats “knew” that the consequences of pressing the lever would be something they did not want, and so they reduced their lever-pressing right from the start. The important point is that they reduced lever-pressing without ever having experienced lever-pressing directly followed by being sick: no lever was present when they were made sick. They seemed able to combine knowledge of the outcome of a behavioral choice (pressing the lever will be followed by getting a pellet) with the reward value of the outcome (pellets are to be avoided) and hence could alter their behavior accordingly. Not every psychologist agrees with this “cognitive” account of this kind of experiment, and it is not the only possible way to explain these results, but the model-based planning explanation is widely accepted.

Nothing prevents an agent from using both model-free and model-based algorithms, and there are good reasons for using both. We know from our own experience that with enough repetition, goal-directed behavior tends to turn into habitual behavior. Experiments show that this happens for rats too. Adams (1982) conducted an experiment to see if extended training would convert goal-directed behavior into habitual behavior. He did this by comparing the effect of outcome devaluation on rats that experienced different amounts of training. If extended training made the rats less sensitive to devaluation compared to rats that received less training, this would be evidence that extended training made the behavior more habitual. Adams’ experiment closely followed the Adams and Dickinson (1981) experiment just described. Simplifying a bit, rats in one group were trained until they made 100 rewarded lever-presses, and rats in the other group—the overtrained group—were trained until they made 500 rewarded lever-presses. After this training, the reward value of the pellets was decreased (using lithium chloride injections) for rats

in both groups. Then both groups of rats were given a session of extinction training. Adams' question was whether devaluation would effect the rate of lever-pressing for the overtrained rats less than it would for the non-overtrained rats, which would be evidence that extended training reduces sensitivity to outcome devaluation. It turned out that devaluation strongly decreased the lever-pressing rate of the non-overtrained rats. For the overtrained rats, in contrast, devaluation had little effect on their lever-pressing; in fact, if anything, it made it more vigorous. (The full experiment included control groups showing that the different amounts of training did not by themselves significantly effect lever-pressing rates after learning.) This result suggested that while the non-overtrained rats were acting in a goal-directed manner sensitive to their knowledge of the outcome of their actions, the overtrained rats had developed a lever-pressing habit.

Viewing this and other results like it from a computational perspective provides insight as to why one might expect animals to behave habitually in some circumstances but in a goal-directed way in others, and why they shift from one mode of control to another as they continue to learn. While animals undoubtedly use algorithms that do not exactly match those we have presented in this book, one can gain insight into animal behavior by considering the tradeoffs that various reinforcement learning algorithms imply. An idea developed by computational neuroscientists Daw, Niv, and Dayan (2005) is that animals use both model-free and model-based processes. Each process proposes an action, and the action chosen for execution is the one proposed by the process judged to be the more trustworthy of the two as determined by measures of confidence that are maintained throughout learning. Early in learning the planning process of a model-based system is more trustworthy because it chains together short-term predictions which can become accurate with less experience than long-term predictions of the model-free process. But with continued experience, the model-free process becomes more trustworthy because planning is prone to making mistakes due to model inaccuracies and short-cuts necessary to make planning feasible, such as various forms of "tree-pruning": the removal of unpromising search tree branches. According to this idea one would expect a shift from goal-directed behavior to habitual behavior as more experience accumulates. Other ideas have been proposed for how animals arbitrate between goal-directed and habitual control, and both behavioral and neuroscience research continues to examine this and related questions.

The distinction between model-free and model-based algorithms is proving to be useful for this research. One can examine the computational implications of these types of algorithms in abstract settings that expose basic advantages and limitations of each type. This serves both to suggest and to sharpen questions that guide the design of experiments necessary for increasing psychologists' understanding of habitual and goal-directed behavioral control.

14.7 Summary

Our goal in this chapter has been to discuss correspondences between reinforcement learning and the experimental study of animal learning in psychology. We emphasized at the outset that reinforcement learning as described in this book is not intended

to model details of animal behavior. It is an abstract computational framework that explores idealized situations from the perspective of artificial intelligence and engineering. But many of the basic reinforcement learning algorithms were inspired by psychological theories, and in some cases, these algorithms have contributed to the development of new animal learning models. This chapter described the most conspicuous of these correspondences.

The distinction in reinforcement learning between algorithms for prediction and algorithms for control parallels animal learning theory's distinction between classical, or Pavlovian, conditioning and instrumental conditioning. The key difference between instrumental and classical conditioning experiments is that in the former the reinforcing stimulus is contingent upon the animal's behavior, whereas in the latter it is not. Learning to predict via a TD algorithm corresponds to classical conditioning, and we described the *TD model of classical conditioning* as one instance in which reinforcement learning principles account for some details of animal learning behavior. This model generalizes the influential Rescorla–Wagner model by including the temporal dimension where events within individual trials influence learning, and it provides an account of second-order conditioning, where predictors of reinforcing stimuli become reinforcing themselves. It also is the basis of an influential view of the activity of dopamine neurons in the brain, something we take up in Chapter 15.

Learning by trial and error is at the base of the control aspect of reinforcement learning. We presented some details about Thorndike's experiments with cats and other animals that led to his *Law of Effect*, which we discussed here and in Chapter 1 (page 15). We pointed out that in reinforcement learning, exploration does not have to be limited to “blind groping”; trials can be generated by sophisticated methods using innate and previously learned knowledge as long as there is *some* exploration. We discussed the training method B. F. Skinner called *shaping* in which reward contingencies are progressively altered to train an animal to successively approximate a desired behavior. Shaping is not only indispensable for animal training, it is also an effective tool for training reinforcement learning agents. There is also a connection to the idea of an animal's motivational state, which influences what an animal will approach or avoid and what events are rewarding or punishing for the animal.

The reinforcement learning algorithms presented in this book include two basic mechanisms for addressing the problem of delayed reinforcement: eligibility traces and value functions learned via TD algorithms. Both mechanisms have antecedents in theories of animal learning. Eligibility traces are similar to stimulus traces of early theories, and value functions correspond to the role of secondary reinforcement in providing nearly immediate evaluative feedback.

The next correspondence the chapter addressed is that between reinforcement learning's environment models and what psychologists call *cognitive maps*. Experiments conducted in the mid 20th century purported to demonstrate the ability of animals to learn cognitive maps as alternatives to, or as additions to, state–action associations, and later use them to guide behavior, especially when the environment changes unexpectedly. Environment models in reinforcement learning are like cognitive maps in that they can be learned by supervised learning methods without relying on reward signals, and then they can be used later to plan behavior.

Reinforcement learning's distinction between *model-free* and *model-based* algorithms corresponds to the distinction in psychology between *habitual* and *goal-directed* behavior. Model-free algorithms make decisions by accessing information that has been stored in a policy or an action-value function, whereas model-based methods select actions as the result of planning ahead using a model of the agent's environment. Outcome-devaluation experiments provide information about whether an animal's behavior is habitual or under goal-directed control. Reinforcement learning theory has helped clarify thinking about these issues.

Animal learning clearly informs reinforcement learning, but as a type of machine learning, reinforcement learning is directed toward designing and understanding effective learning algorithms, not toward replicating or explaining details of animal behavior. We focused on aspects of animal learning that relate in clear ways to methods for solving prediction and control problems, highlighting the fruitful two-way flow of ideas between reinforcement learning and psychology without venturing deeply into many of the behavioral details and controversies that have occupied the attention of animal learning researchers. Future development of reinforcement learning theory and algorithms will likely exploit links to many other features of animal learning as the computational utility of these features becomes better appreciated. We expect that a flow of ideas between reinforcement learning and psychology will continue to bear fruit for both disciplines.

Many connections between reinforcement learning and areas of psychology and other behavioral sciences are beyond the scope of this chapter. We largely omit discussing links to the psychology of decision making, which focuses on how actions are selected, or how decisions are made, *after* learning has taken place. We also do not discuss links to ecological and evolutionary aspects of behavior studied by ethologists and behavioral ecologists: how animals relate to one another and to their physical surroundings, and how their behavior contributes to evolutionary fitness. Optimization, MDPs, and dynamic programming figure prominently in these fields, and our emphasis on agent interaction with dynamic environments connects to the study of agent behavior in complex "ecologies." Multi-agent reinforcement learning, omitted in this book, has connections to social aspects of behavior. Despite the lack of treatment here, reinforcement learning should by no means be interpreted as dismissing evolutionary perspectives. Nothing about reinforcement learning implies a *tabula rasa* view of learning and behavior. Indeed, experience with engineering applications has highlighted the importance of building into reinforcement learning systems knowledge that is analogous to what evolution provides to animals.

Bibliographical and Historical Remarks

Ludvig, Bellemare, and Pearson (2011) and Shah (2012) review reinforcement learning in the contexts of psychology and neuroscience. These publications are useful companions to this chapter and the following chapter on reinforcement learning and neuroscience.

- 14.1** Dayan, Niv, Seymour, and Daw (2006) focused on interactions between classical and instrumental conditioning, particularly situations where classically-conditioned and instrumental responses are in conflict. They proposed a Q-learning framework for modeling aspects of this interaction. Modayil and Sutton (2014) used a mobile robot to demonstrate the effectiveness of a control method combining a fixed response with online prediction learning. Calling this *Pavlovian control*, they emphasized that it differs from the usual control methods of reinforcement learning, being based on predictively executing fixed responses and not on reward maximization. The electro-mechanical machine of Ross (1933) and especially the learning version of Walter's turtle (Walter, 1951) were very early illustrations of Pavlovian control.
- 14.2.1** Kamin (1968) first reported blocking, now commonly known as Kamin blocking, in classical conditioning. Moore and Schmajuk (2008) provide an excellent summary of the blocking phenomenon, the research it stimulated, and its lasting influence on animal learning theory. Gibbs, Cool, Land, Kehoe, and Gormezano (1991) describe second-order conditioning of the rabbit's nictitating membrane response and its relationship to conditioning with serial-compound stimuli. Finch and Culler (1934) reported obtaining fifth-order conditioning of a dog's foreleg withdrawal "when the *motivation* of the animal is maintained through the various orders."
- 14.2.2** The idea built into the Rescorla–Wagner model that learning occurs when animals are surprised is derived from Kamin (1969). Models of classical conditioning other than Rescorla and Wagner's include the models of Klopf (1988), Grossberg (1975), Mackintosh (1975), Moore and Stickney (1980), Pearce and Hall (1980), and Courville, Daw, and Touretzky (2006). Schmajuk (2008) reviews models of classical conditioning. Wagner (2008) provides a modern psychological perspective on the Rescorla-Wagner model and similar elemental theories of learning.
- 14.2.3** An early version of the TD model of classical conditioning appeared in Sutton and Barto (1981a), which also included the early model's prediction that temporal primacy overrides blocking, later shown by Kehoe, Schreurs, and Graham (1987) to occur in the rabbit nictitating membrane preparation. Sutton and Barto (1981a) contains the earliest recognition of the near identity between the Rescorla–Wagner model and the Least-Mean-Square (LMS), or Widrow-Hoff, learning rule (Widrow and Hoff, 1960). This early model was revised following Sutton's development of the TD algorithm (Sutton, 1984, 1988) and was first presented as the TD model in Sutton and Barto (1987) and more completely in Sutton and Barto (1990), upon which this section is largely based. Additional exploration

of the TD model and its possible neural implementation was conducted by Moore and colleagues (Moore, Desmond, Berthier, Blazis, Sutton, and Barto, 1986; Moore and Blazis, 1989; Moore, Choi, and Brunzell, 1998; Moore, Marks, Castagna, and Polewan, 2001). Klopf's (1988) drive-reinforcement theory of classical conditioning extends the TD model to address additional experimental details, such as the S-shape of acquisition curves. In some of these publications TD is taken to mean Time Derivative instead of Temporal Difference.

- 14.2.4** Ludvig, Sutton, and Kehoe (2012) evaluated the performance of the TD model in previously unexplored tasks involving classical conditioning and examined the influence of various stimulus representations, including the microstimulus representation that they introduced earlier (Ludvig, Sutton, and Kehoe, 2008). Earlier investigations of the influence of various stimulus representations and their possible neural implementations on response timing and topography in the context of the TD model are those of Moore and colleagues cited above. Although not in the context of the TD model, representations like the microstimulus representation of Ludvig et al. (2012) have been proposed and studied by Grossberg and Schmajuk (1989), Brown, Bullock, and Grossberg (1999), Buhusi and Schmajuk (1999), and Machado (1997). The figures on pages 353–355 are adapted from Sutton and Barto (1990).

- 14.3** Section 1.7 includes comments on the history of trial-and-error learning and the Law of Effect. The idea that Thorndike's cats might have been exploring according to an instinctual context-specific ordering over actions rather than by just selecting from a set of instinctual impulses was suggested by Peter Dayan (personal communication). Selfridge, Sutton, and Barto (1985) illustrated the effectiveness of shaping in a pole-balancing reinforcement learning task. Other examples of shaping in reinforcement learning are Gullapalli and Barto (1992), Mahadevan and Connell (1992), Mataric (1994), Dorigo and Colombette (1994), Saksida, Raymond, and Touretzky (1997), and Randløv and Alstrøm (1998). Ng (2003) and Ng, Harada, and Russell (1999) used the term shaping in a sense somewhat different from Skinner's, focusing on the problem of how to alter the reward signal without altering the set of optimal policies.

Dickinson and Balleine (2002) discuss the complexity of the interaction between learning and motivation. Wise (2004) provides an overview of reinforcement learning and its relation to motivation. Daw and Shohamy (2008) link motivation and learning to aspects of reinforcement learning theory. See also McClure, Daw, and Montague (2003), Niv, Joel, and Dayan (2006), Rangel, Camerer, and Montague (2008), and Dayan and Berridge (2014). McClure et al. (2003), Niv, Daw, and Dayan (2006), and Niv, Daw, Joel, and Dayan (2007) present theories of behavioral vigor related to the reinforcement learning framework.

- 14.4** Spence, Hull's student and collaborator at Yale, elaborated the role of higher-order reinforcement in addressing the problem of delayed reinforcement (Spence, 1947). Learning over very long delays, as in taste-aversion conditioning with

delays up to several hours, led to interference theories as alternatives to decaying-trace theories (e.g., Revusky and Garcia, 1970; Boakes and Costa, 2014). Other views of learning under delayed reinforcement invoke roles for awareness and working memory (e.g., Clark and Squire, 1998; Seo, Barraclough, and Lee, 2007).

- 14.5** Thistlthwaite (1951) provides an extensive review of latent learning experiments up to the time of its publication. Ljung (1998) provides an overview of model learning, or system identification, techniques in engineering. Gopnik, Glymour, Sobel, Schulz, Kushnir, and Danks (2004) present a Bayesian theory about how children learn models.
- 14.6** Connections between habitual and goal-directed behavior and model-free and model-based reinforcement learning were first proposed by Daw, Niv, and Dayan (2005). The hypothetical maze task used to explain habitual and goal-directed behavioral control is based on the explanation of Niv, Joel, and Dayan (2006). Dolan and Dayan (2013) review four generations of experimental research related to this issue and discuss how it can move forward on the basis of reinforcement learning's model-free/model-based distinction. Dickinson (1980, 1985) and Dickinson and Balleine (2002) discuss experimental evidence related to this distinction. Donahoe and Burgos (2000) alternatively argue that model-free processes can account for the results of outcome-devaluation experiments. Dayan and Berridge (2014) argue that classical conditioning involves model-based processes. Rangel, Camerer, and Montague (2008) review many of the outstanding issues involving habitual, goal-directed, and Pavlovian modes of control.

Comments on Terminology— The traditional meaning of *reinforcement* in psychology is the strengthening of a pattern of behavior (by increasing either its intensity or frequency) as a result of an animal receiving a stimulus (or experiencing the omission of a stimulus) in an appropriate temporal relationship with another stimulus or with a response. Reinforcement produces changes that remain in future behavior. Sometimes in psychology reinforcement refers to the process of producing lasting changes in behavior, whether the changes strengthen or weaken a behavior pattern (Mackintosh, 1983). Letting reinforcement refer to weakening in addition to strengthening is at odds with the everyday meaning of reinforce, and its traditional use in psychology, but it is a useful extension that we have adopted here. In either case, a stimulus considered to be the cause of the behavioral change is called a *reinforcer*.

Psychologists do not generally use the specific phrase *reinforcement learning* as we do. Animal learning pioneers probably regarded reinforcement and learning as being synonymous, so it would be redundant to use both words. Our use of the phrase follows its use in computational and engineering research, influenced mostly by Minsky (1961). But the phrase is lately gaining currency in psychology and neuroscience, likely because strong parallels have surfaced between reinforcement learning algorithms and animal learning—parallels described in this chapter and the next.

According to common usage, a *reward* is an object or event that an animal will approach and work for. A reward may be given to an animal in recognition of its ‘good’

behavior, or given in order to make the animal's behavior 'better.' Similarly, a *penalty* is an object or event that the animal usually avoids and that is given as a consequence of 'bad' behavior, usually in order to change that behavior. *Primary reward* is reward due to machinery built into an animal's nervous system by evolution to improve its chances of survival and reproduction, for example, reward produced by the taste of nourishing food, sexual contact, successful escape, and many other stimuli and events that predicted reproductive success over the animal's ancestral history. As explained in Section 14.2.1, *higher-order reward* is reward delivered by stimuli that predict primary reward, either directly or indirectly by predicting other stimuli that predict primary reward. Reward is *secondary* if its rewarding quality is the result of directly predicting primary reward.

In this book we call R_t the 'reward signal at time t ,' or sometimes just the 'reward at time t ,' but we do not think of it as an object or event in the agent's environment. Because R_t is a number—not an object or an event—it is more like a reward signal in neuroscience, which is a signal internal to the brain, like the activity of neurons, that influences decision making and learning. This signal might be triggered when the animal perceives an attractive (or an aversive) object, but it can also be triggered by things that do not physically exist in the animal's external environment, such as memories, ideas, or hallucinations. Because our R_t can be positive, negative, or zero, it might be better to call a negative R_t a penalty, and an R_t equal to zero a neutral signal, but for simplicity we generally avoid these terms.

In reinforcement learning, the process that generates all the R_t s defines the problem the agent is trying to solve. The agent's objective is to keep the magnitude of R_t as large as possible over time. In this respect, R_t is like primary reward for an animal if we think of the problem the animal faces as the problem of obtaining as much primary reward as possible over its lifetime (and thereby, through the prospective "wisdom" of evolution, improve its chances of solving its real problem, which is to pass its genes on to future generations). However, as we suggest in Chapter 15, it is unlikely that there is a single "master" reward signal like R_t in an animal's brain.

Not all reinforcers are rewards or penalties. Sometimes reinforcement is not the result of an animal receiving a stimulus that evaluates its behavior by labeling the behavior good or bad. A behavior pattern can be reinforced by a stimulus that arrives to an animal no matter how the animal behaved. As described in Section 14.1, whether the delivery of reinforcement depends, or does not depend, on preceding behavior is the defining difference between instrumental, or operant, conditioning experiments and classical, or Pavlovian, conditioning experiments. Reinforcement is at work in both types of experiments, but only in the former is it feedback that evaluates past behavior. (Though it has often been pointed out that even when the reinforcing US in a classical conditioning experiment is not contingent on the subject's preceding behavior, its reinforcing value can be influenced by this behavior, an example being that a closed eye makes an air puff to the eye less aversive.)

The distinction between reward signals and reinforcement signals is a crucial point when we discuss neural correlates of these signals in the next chapter. Like a reward signal, for us, the reinforcement signal at any specific time is a positive or negative number, or zero. A reinforcement signal is the major factor directing changes a learning algorithm

makes in an agent's policy, value estimates, or environment models. The definition that makes the most sense to us is that a reinforcement signal at any time is a number that multiplies (possibly along with some constants) a vector to determine parameter updates in some learning algorithm.

For some algorithms, the reward signal alone is the critical multiplier in the parameter-update equation. For these algorithms the reinforcement signal is the same as the reward signal. But for most of the algorithms we discuss in this book, reinforcement signals include terms in addition to the reward signal, an example being a TD error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, which is the reinforcement signal for TD state-value learning (and analogous TD errors for action-value learning). In this reinforcement signal, R_{t+1} is the *primary reinforcement* contribution, and the temporal difference in predicted values, $\gamma V(S_{t+1}) - V(S_t)$ (or an analogous temporal difference for action values), is the *conditioned reinforcement* contribution. Thus, whenever $\gamma V(S_{t+1}) - V(S_t) = 0$, δ_t signals 'pure' primary reinforcement; and whenever $R_{t+1} = 0$, it signals 'pure' conditioned reinforcement, but it often signals a mixture of these. Note as we mentioned in Section 6.1, this δ_t is not available until time $t + 1$. We therefore think of δ_t as the reinforcement signal at time $t + 1$, which is fitting because it reinforces predictions and/or actions made earlier at step t .

A possible source of confusion is the terminology used by the famous psychologist B. F. Skinner and his followers. For Skinner, positive reinforcement occurs when the consequences of an animal's behavior increase the frequency of that behavior; punishment occurs when the behavior's consequences decrease that behavior's frequency. Negative reinforcement occurs when behavior leads to the removal of an aversive stimulus (that is, a stimulus the animal does not like), thereby increasing the frequency of that behavior. Negative punishment, on the other hand, occurs when behavior leads to the removal of an appetitive stimulus (that is, a stimulus the animal likes), thereby decreasing the frequency of that behavior. We find no critical need for these distinctions because our approach is more abstract than this, with both reward and reinforcement signals allowed to take on both positive and negative values. (But note especially that when our reinforcement signal is negative, it is not the same as Skinner's negative reinforcement.)

On the other hand, it has often been pointed out that using a single number as a reward or a penalty signal, depending only on its sign, is at odds with the fact that animals' appetitive and aversive systems have qualitatively different properties and involve different brain mechanisms. This points to a direction in which the reinforcement learning framework might be developed in the future to exploit computational advantages of separate appetitive and aversive systems, but for now we are passing over these possibilities.

Another discrepancy in terminology is how we use the word *action*. To many cognitive scientists, an action is purposeful in the sense of being the result of an animal's knowledge about the relationship between the behavior in question and the consequences of that behavior. An action is goal-directed and the result of a decision, whereas a response is triggered by a stimulus and is the result of a reflex or a habit. We use the word action without differentiating among what others call actions, decisions, and responses. These are important distinctions, but for us they are encompassed by differences between

model-free and model-based reinforcement learning algorithms, which we discussed above in relation to habitual and goal-directed behavior in Section 14.6. Dickinson (1985) discusses the distinction between responses and actions.

A term used a lot in this book is *control*. What we mean by control is entirely different from what it means to animal learning psychologists. By control we mean that an agent influences its environment to bring about states or events that the agent prefers: the agent exerts control over its environment. This is the sense of control used by control engineers. In psychology, on the other hand, control typically means that an animal's behavior is influenced by—is controlled by—the stimuli the animal receives (stimulus control) or the reinforcement schedule it experiences. Here the environment is controlling the agent. Control in this sense is the basis of behavior modification therapy. Of course, both of these directions of control are at play when an agent interacts with its environment, but our focus is on the agent as controller, not the environment as controller. A view equivalent to ours, and perhaps more illuminating, is that the agent is actually controlling the input it receives from its environment (Powers, 1973). This is *not* what psychologists mean by stimulus control.

Sometimes reinforcement learning is understood to refer solely to learning policies directly from rewards (and penalties) without the involvement of value functions or environment models. This is what psychologists call stimulus-response, or S-R, learning. But for us, along with most of today's psychologists, reinforcement learning is much broader than this, including in addition to S-R learning, methods involving value functions, environment models, planning, and other processes that are commonly thought to belong to the more cognitive side of mental functioning.

Chapter 15

Neuroscience

Neuroscience is the multidisciplinary study of nervous systems: how they regulate bodily functions; control behavior; change over time as a result of development, learning, and aging; and how cellular and molecular mechanisms make these functions possible. One of the most exciting aspects of reinforcement learning is the mounting evidence from neuroscience that the nervous systems of humans and many other animals implement algorithms that correspond in striking ways to reinforcement learning algorithms. The main objective of this chapter is to explain these parallels and what they suggest about the neural basis of reward-related learning in animals.

The most remarkable point of contact between reinforcement learning and neuroscience involves dopamine, a chemical deeply involved in reward processing in the brains of mammals. Dopamine appears to convey temporal-difference (TD) errors to brain structures where learning and decision making take place. This parallel is expressed by the *reward prediction error hypothesis of dopamine neuron activity*, a hypothesis that resulted from the convergence of computational reinforcement learning and results of neuroscience experiments. In this chapter we discuss this hypothesis, the neuroscience findings that led to it, and why it is a significant contribution to understanding brain reward systems. We also discuss parallels between reinforcement learning and neuroscience that are less striking than this dopamine/TD-error parallel but that provide useful conceptual tools for thinking about reward-based learning in animals. Other elements of reinforcement learning have the potential to impact the study of nervous systems, but their connections to neuroscience are still relatively undeveloped. We discuss several of these evolving connections that we think will grow in importance over time.

As we outlined in the history section of this book's introductory chapter (Section 1.7), many aspects of reinforcement learning were influenced by neuroscience. A second objective of this chapter is to acquaint readers with ideas about brain function that have contributed to our approach to reinforcement learning. Some elements of reinforcement learning are easier to understand when seen in light of theories of brain function. This is particularly true for the idea of the eligibility trace, one of the basic mechanisms of reinforcement learning, that originated as a conjectured property of synapses, the structures by which nerve cells—neurons—communicate with one another.

In this chapter we do not delve very deeply into the enormous complexity of the neural systems underlying reward-based learning in animals: this chapter is too short, and we are not neuroscientists. We do not try to describe—or even to name—the very many brain structures and pathways, or any of the molecular mechanisms, believed to be involved in these processes. We also do not do justice to hypotheses and models that are alternatives to those that align so well with reinforcement learning. It should not be surprising that there are differing views among experts in the field. We can only provide a glimpse into this fascinating and developing story. We hope, though, that this chapter convinces you that a very fruitful channel has emerged connecting reinforcement learning and its theoretical underpinnings to the neuroscience of reward-based learning in animals.

Many excellent publications cover links between reinforcement learning and neuroscience, some of which we cite in this chapter's final section. Our treatment differs from most of these because we assume familiarity with reinforcement learning as presented in the earlier chapters of this book, but we do not assume knowledge of neuroscience. We begin with a brief introduction to the neuroscience concepts needed for a basic understanding of what is to follow.

15.1 Neuroscience Basics

Some basic information about nervous systems is helpful for following what we cover in this chapter. Terms that we refer to later are italicized. Skipping this section will not be a problem if you already have an elementary knowledge of neuroscience.

Neurons, the main components of nervous systems, are cells specialized for processing and transmitting information using electrical and chemical signals. They come in many forms, but a neuron typically has a cell body, *dendrites*, and a single *axon*. Dendrites are structures that branch from the cell body to receive input from other neurons (or to also receive external signals in the case of sensory neurons). A neuron's axon is a fiber that carries the neuron's output to other neurons (or to muscles or glands). A neuron's output consists of sequences of electrical pulses called *action potentials* that travel along the axon. Action potentials are also called *spikes*, and a neuron is said to *fire* when it generates a spike. In models of neural networks it is common to use real numbers to represent a neuron's *firing rate*, the average number of spikes per some unit of time.

A neuron's axon can branch widely so that the neuron's action potentials reach many targets. The branching structure of a neuron's axon is called the neuron's *axonal arbor*. Because the conduction of an action potential is an active process, not unlike the burning of a fuse, when an action potential reaches an axonal branch point it "lights up" action potentials on all of the outgoing branches (although propagation to a branch can sometimes fail). As a result, the activity of a neuron with a large axonal arbor can influence many target sites.

A *synapse* is a structure generally at the termination of an axon branch that mediates the communication of one neuron to another. A synapse transmits information from the *presynaptic* neuron's axon to a dendrite or cell body of the *postsynaptic* neuron. With a few exceptions, synapses release a chemical *neurotransmitter* upon the arrival of an action potential from the presynaptic neuron. (The exceptions are cases of direct electric coupling between neurons, but these will not concern us here.) Neurotransmitter molecules released from the presynaptic side of the synapse diffuse across the *synaptic cleft*, the very small space between the presynaptic ending and the postsynaptic neuron, and then bind to receptors on the surface of the postsynaptic neuron to excite or inhibit its spike-generating activity, or to modulate its behavior in other ways. A particular neurotransmitter may bind to several different types of receptors, with each producing a different effect on the postsynaptic neuron. For example, there are at least five different receptor types by which the neurotransmitter dopamine can affect a postsynaptic neuron. Many different chemicals have been identified as neurotransmitters in animal nervous systems.

A neuron's *background* activity is its level of activity, usually its firing rate, when the neuron does not appear to be driven by synaptic input related to the task of interest to the experimenter, for example, when the neuron's activity is not correlated with a stimulus delivered to a subject as part of an experiment. Background activity can be irregular due to input from the wider network, or due to noise within the neuron or its synapses. Sometimes background activity is the result of dynamic processes intrinsic to the neuron. A neuron's *phasic* activity, in contrast to its background activity, consists of bursts of spiking activity usually caused by synaptic input. Activity that varies slowly and often in a graded manner, whether as background activity or not, is called a neuron's *tonic* activity.

The strength or effectiveness by which the neurotransmitter released at a synapse influences the postsynaptic neuron is the synapse's *efficacy*. One way a nervous system can change through experience is through changes in synaptic efficacies as a result of combinations of the activities of the presynaptic and postsynaptic neurons, and sometimes by the presence of a *neuromodulator*, which is a neurotransmitter having effects other than, or in addition to, direct fast excitation or inhibition.

Brains contain several different neuromodulation systems consisting of clusters of neurons with widely branching axonal arbors, with each system using a different neurotransmitter. Neuromodulation can alter the function of neural circuits, mediate motivation, arousal, attention, memory, mood, emotion, sleep, and body temperature. Important here is that a neuromodulatory system can distribute something like a scalar signal, such as a reinforcement signal, to alter the operation of synapses in widely distributed sites critical for learning.

The ability of synaptic efficacies to change is called *synaptic plasticity*. It is one of the primary mechanisms responsible for learning. The parameters, or weights, adjusted by learning algorithms correspond to synaptic efficacies. As we detail below, modulation of synaptic plasticity via the neuromodulator dopamine is a plausible mechanism for how the brain might implement learning algorithms like many of those described in this book.

15.2 Reward Signals, Reinforcement Signals, Values, and Prediction Errors

Links between neuroscience and computational reinforcement learning begin as parallels between signals in the brain and signals playing prominent roles in reinforcement learning theory and algorithms. In Chapter 3 we said that any problem of learning goal-directed behavior can be reduced to the three signals representing actions, states, and rewards. However, to explain links that have been made between neuroscience and reinforcement learning, we have to be less abstract than this and consider other reinforcement learning signals that correspond, in certain ways, to signals in the brain. In addition to reward signals, these include reinforcement signals (which we argue are different from reward signals), value signals, and signals conveying prediction errors. When we label a signal by its function in this way, we are doing it in the context of reinforcement learning theory in which the signal corresponds to a term in an equation or an algorithm. On the other hand, when we refer to a signal in the brain, we mean a physiological event such as a burst of action potentials or the secretion of a neurotransmitter. Labeling a neural signal by its function, for example calling the phasic activity of a dopamine neuron a reinforcement signal, means that the neural signal behaves like, and is conjectured to function like, the corresponding theoretical signal.

Uncovering evidence for these correspondences involves many challenges. Neural activity related to reward processing can be found in nearly every part of the brain, and it is difficult to interpret results unambiguously because representations of different reward-related signals tend to be highly correlated with one another. Experiments need to be carefully designed to allow one type of reward-related signal to be distinguished with any degree of certainty from others—or from an abundance of other signals not related to reward processing. Despite these difficulties, many experiments have been conducted with the aim of reconciling aspects of reinforcement learning theory and algorithms with neural signals, and some compelling links have been established. To prepare for examining these links, in the rest of this section we remind the reader of what various reward-related signals mean according to reinforcement learning theory.

In our *Comments on Terminology* at the end of the previous chapter, we said that R_t is like a reward signal in an animal's brain and not an object or event in the animal's environment. In reinforcement learning, the reward signal (along with an agent's environment) defines the problem a reinforcement learning agent is trying to solve. In this respect, R_t is like a signal in an animal's brain that distributes primary reward to sites throughout the brain. But it is unlikely that a unitary master reward signal like R_t exists in an animal's brain. It is best to think of R_t as an abstraction summarizing the overall effect of a multitude of neural signals generated by many systems in the brain that assess the rewarding or punishing qualities of sensations and states.

Reinforcement signals in reinforcement learning are different from reward signals. The function of a reinforcement signal is to direct the changes a learning algorithm makes in an agent's policy, value estimates, or environment models. For a TD method, for instance, the reinforcement signal at time t is the TD error $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$.¹ The

¹As we mentioned in Section 6.1, δ_t in our notation is defined to be $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, so δ_t

reinforcement signal for some algorithms could be just the reward signal, but for most of the algorithms we consider the reinforcement signal is the reward signal adjusted by other information, such as the value estimates in TD errors.

Estimates of state values or of action values, that is, V or Q , specify what is good or bad for the agent over the long run. They are predictions of the total reward an agent can expect to accumulate over the future. Agents make good decisions by selecting actions leading to states with the largest estimated state values, or by selecting actions with the largest estimated action values.

Prediction errors measure discrepancies between expected and actual signals or sensations. Reward prediction errors (RPEs) specifically measure discrepancies between the expected and the received reward signal, being positive when the reward signal is greater than expected, and negative otherwise. TD errors like (6.5) are special kinds of RPEs that signal discrepancies between current and earlier expectations of reward over the long-term. When neuroscientists refer to RPEs they generally (though not always) mean TD RPEs, which we simply call TD errors throughout this chapter. Also in this chapter, a TD error is generally one that does not depend on actions, as opposed to TD errors used in learning action-values by algorithms like Sarsa and Q-learning. This is because the most well-known links to neuroscience are stated in terms of action-free TD errors, but we do not mean to rule out possible similar links involving action-dependent TD errors. (TD errors for predicting signals other than rewards are useful too, but that case will not concern us here. See, for example, Modayil, White, and Sutton, 2014.)

One can ask many questions about links between neuroscience data and these theoretically-defined signals. Is an observed signal more like a reward signal, a value signal, a prediction error, a reinforcement signal, or something altogether different? And if it is an error signal, is it an RPE, a TD error, or a simpler error like the Rescorla–Wagner error (14.3)? And if it is a TD error, does it depend on actions like the TD error of Q-learning or Sarsa? As indicated above, probing the brain to answer questions like these is extremely difficult. But experimental evidence suggests that one neurotransmitter, specifically the neurotransmitter dopamine, signals RPEs, and further, that the phasic activity of dopamine-producing neurons in fact conveys TD errors (see Section 15.1 for a definition of phasic activity). This evidence led to the *reward prediction error hypothesis of dopamine neuron activity*, which we describe next.

15.3 The Reward Prediction Error Hypothesis

The *reward prediction error hypothesis of dopamine neuron activity* proposes that one of the functions of the phasic activity of dopamine-producing neurons in mammals is to deliver an error between an old and a new estimate of expected future reward to target areas throughout the brain. This hypothesis (though not in these exact words) was first explicitly stated by Montague, Dayan, and Sejnowski (1996), who showed how the TD error concept from reinforcement learning accounts for many features of the phasic

is not available until time $t + 1$. The TD error *available* at t is actually $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$. Because we are thinking of time steps as very small, or even infinitesimal, time intervals, one should not attribute undue importance to this one-step time shift.

activity of dopamine neurons in mammals. The experiments that led to this hypothesis were performed in the 1980s and early 1990s in the laboratory of neuroscientist Wolfram Schultz. Section 15.5 describes these influential experiments, Section 15.6 explains how the results of these experiments align with TD errors, and the Bibliographical and Historical Remarks section at the end of this chapter includes a guide to the literature surrounding the development of this influential hypothesis.

Montague et al. (1996) compared the TD errors of the TD model of classical conditioning with the phasic activity of dopamine-producing neurons during classical conditioning experiments. Recall from Section 14.2 that the TD model of classical conditioning is basically the semi-gradient-descent $\text{TD}(\lambda)$ algorithm with linear function approximation. Montague et al. made several assumptions to set up this comparison. First, because a TD error can be negative but neurons cannot have a negative firing rate, they assumed that the quantity corresponding to dopamine neuron activity is $\delta_{t-1} + b_t$, where b_t is the background firing rate of the neuron. A negative TD error corresponds to a drop in a dopamine neuron's firing rate below its background rate.²

A second assumption was needed about the states visited in each classical conditioning trial and how they are represented as inputs to the learning algorithm. This is the same issue we discussed in Section 14.2.4 for the TD model. Montague et al. chose a complete serial compound (CSC) representation as shown in the left column of Figure 14.1, but where the sequence of short-duration internal signals continues until the onset of the US, which here is the arrival of a non-zero reward signal. This representation allows the TD error to mimic the fact that dopamine neuron activity not only predicts a future reward, but that it is also sensitive to *when* after a predictive cue that reward is expected to arrive. There has to be some way to keep track of the time between sensory cues and the arrival of reward. If a stimulus initiates a sequence of internal signals that continues after the stimulus ends, and if there is a different signal for each time step following the stimulus, then each time step after the stimulus is represented by a distinct state. Thus, the TD error, being state-dependent, can be sensitive to the timing of events within a trial.

In simulated trials with these assumptions about background firing rate and input representation, TD errors of the TD model are remarkably similar to dopamine neuron phasic activity. Previewing our description of details about these similarities in Section 15.5 below, the TD errors parallel the following features of dopamine neuron activity: (1) the phasic response of a dopamine neuron only occurs when a rewarding event is unpredicted; (2) early in learning, neutral cues that precede a reward do not cause substantial phasic dopamine responses, but with continued learning these cues gain predictive value and come to elicit phasic dopamine responses; (3) if an even earlier cue reliably precedes a cue that has already acquired predictive value, the phasic dopamine response shifts to the earlier cue, ceasing for the later cue; and (4) if after learning, the predicted rewarding event is omitted, a dopamine neuron's response decreases below its baseline level shortly after the expected time of the rewarding event.

²In the literature relating TD errors to the activity of dopamine neurons, their δ_t is the same as our $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$.

Although not every dopamine neuron monitored in the experiments of Schultz and colleagues behaved in all of these ways, the striking correspondence between the activities of most of the monitored neurons and TD errors lends strong support to the reward prediction error hypothesis. There are situations, however, in which predictions based on the hypothesis do not match what is observed in experiments. The choice of input representation is critical to how closely TD errors match some of the details of dopamine neuron activity, particularly details about the timing of dopamine neuron responses. Different ideas, some of which we discuss below, have been proposed about input representations and other features of TD learning to make the TD errors fit the data better, though the main parallels appear with the CSC representation that Montague et al. used. Overall, the reward prediction error hypothesis has received wide acceptance among neuroscientists studying reward-based learning, and it has proven to be remarkably resilient in the face of accumulating results from neuroscience experiments.

To prepare for our description of the neuroscience experiments supporting the reward prediction error hypothesis, and to provide some context so that the significance of the hypothesis can be appreciated, we next present some of what is known about dopamine, the brain structures it influences, and how it is involved in reward-based learning.

15.4 Dopamine

Dopamine is produced as a neurotransmitter by neurons whose cell bodies lie mainly in two clusters of neurons in the midbrain of mammals: the substantia nigra pars compacta (SNpc) and the ventral tegmental area (VTA). Dopamine plays essential roles in many processes in the mammalian brain. Prominent among these are motivation, learning, action-selection, most forms of addiction, and the disorders schizophrenia and Parkinson's disease. Dopamine is called a neuromodulator because it performs many functions other than direct fast excitation or inhibition of targeted neurons. Although much remains unknown about dopamine's functions and details of its cellular effects, it is clear that it is fundamental to reward processing in the mammalian brain. Dopamine is not the only neuromodulator involved in reward processing, and its role in aversive situations—punishment—remains controversial. Dopamine also can function differently in non-mammals. But no one doubts that dopamine is essential for reward-related processes in mammals, including humans.

An early, traditional view is that dopamine neurons broadcast a reward signal to multiple brain regions implicated in learning and motivation. This view followed from a famous 1954 paper by James Olds and Peter Milner that described the effects of electrical stimulation on certain areas of a rat's brain. They found that electrical stimulation to particular regions acted as a very powerful reward in controlling the rat's behavior: "...the control exercised over the animal's behavior by means of this reward is extreme, possibly exceeding that exercised by any other reward previously used in animal experimentation" (Olds and Milner, 1954). Later research revealed that the sites at which stimulation was most effective in producing this rewarding effect excited dopamine pathways, either directly or indirectly, that ordinarily are excited by natural rewarding stimuli. Effects similar to these were also observed with human subjects. These observations strongly suggested that dopamine neuron activity signals reward.

But if the reward prediction error hypothesis is correct—even if it accounts for only some features of a dopamine neuron’s activity—this traditional view of dopamine neuron activity is not entirely correct: phasic responses of dopamine neurons signal reward prediction errors, not reward itself. In reinforcement learning’s terms, a dopamine neuron’s phasic response at a time t corresponds to $\delta_{t-1} = R_t + \gamma V(S_t) - V(S_{t-1})$, not to R_t .

Reinforcement learning theory and algorithms help reconcile the reward-prediction-error view with the conventional notion that dopamine signals reward. In many of the algorithms we discuss in this book, δ functions as a reinforcement signal, meaning that it is the main driver of learning. For example, δ is the critical factor in the TD model of classical conditioning, and δ is the reinforcement signal for learning both a value function and a policy in an actor–critic architecture (Sections 13.5 and 15.7). Action-dependent forms of δ are reinforcement signals for Q-learning and Sarsa. The reward signal R_t is a crucial component of δ_{t-1} , but it is not the complete determinant of its reinforcing effect in these algorithms. The additional term $\gamma V(S_t) - V(S_{t-1})$ is the higher-order reinforcement part of δ_{t-1} , and even if reward occurs ($R_t \neq 0$), the TD error can be silent if the reward is fully predicted (which is fully explained in Section 15.6 below).

A closer look at Olds’ and Milner’s 1954 paper, in fact, reveals that it is mainly about the reinforcing effect of electrical stimulation in an instrumental conditioning task. Electrical stimulation not only energized the rats’ behavior—through dopamine’s effect on motivation—it also led to the rats quickly learning to stimulate themselves by pressing a lever, which they would do frequently for long periods of time. The activity of dopamine neurons triggered by electrical stimulation reinforced the rats’ lever pressing.

More recent experiments using optogenetic methods clinch the role of phasic responses of dopamine neurons as reinforcement signals. These methods allow neuroscientists to precisely control the activity of selected neuron types at a millisecond timescale in awake behaving animals. Optogenetic methods introduce light-sensitive proteins into selected neuron types so that these neurons can be activated or silenced by means of flashes of laser light. The first experiment using optogenetic methods to study dopamine neurons showed that optogenetic stimulation producing phasic activation of dopamine neurons in mice was enough to condition the mice to prefer the side of a chamber where they received this stimulation as compared to the chamber’s other side where they received no, or lower-frequency, stimulation (Tsai et al. 2009). In another example, Steinberg et al. (2013) used optogenetic activation of dopamine neurons to create artificial bursts of dopamine neuron activity in rats at the times when rewarding stimuli were expected but omitted—times when dopamine neuron activity normally pauses. With these pauses replaced by artificial bursts, responding was sustained when it would ordinarily decrease due to lack of reinforcement (in extinction trials), and learning was enabled when it would ordinarily be blocked due to the reward being already predicted (the blocking paradigm; Section 14.2.1).

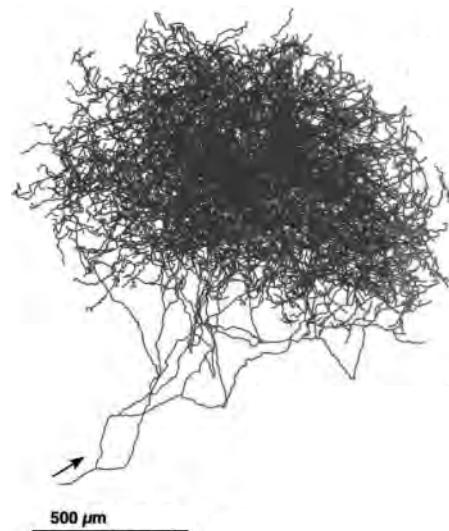
Additional evidence for the reinforcing function of dopamine comes from optogenetic experiments with fruit flies, except in these animals dopamine’s effect is the opposite of its effect in mammals: optically triggered bursts of dopamine neuron activity act just like electric foot shock in reinforcing avoidance behavior, at least for the population

of dopamine neurons activated (Claridge-Chang et al. 2009). Although none of these optogenetic experiments showed that phasic dopamine neuron activity is specifically like a TD error, they convincingly demonstrated that phasic dopamine neuron activity acts just like δ acts (or perhaps like *minus* δ acts in fruit flies) as the reinforcement signal in algorithms for both prediction (classical conditioning) and control (instrumental conditioning).

Dopamine neurons are particularly well suited to broadcasting a reinforcement signal to many areas of the brain. These neurons have huge axonal arbors, each releasing dopamine at 100 to 1,000 times more synaptic sites than reached by the axons of typical neurons. Shown to the right is the axonal arbor of a single dopamine neuron whose cell body is in the SNpc of a rat's brain. Each axon of a SNpc or VTA dopamine neuron makes roughly 500,000 synaptic contacts on the dendrites of neurons in targeted brain areas.

If dopamine neurons broadcast a reinforcement signal like reinforcement learning's δ , then because this is a scalar signal, i.e., a single number, all dopamine neurons in both the SNpc and VTA would be expected to activate more-or-less identically so that they would act in near synchrony to send the same signal to all of the sites their axons target. Although it has been a common belief that dopamine neurons do act together like this, modern evidence is pointing to the more complicated picture that different subpopulations of dopamine neurons respond to input differently depending on the structures to which they send their signals and the different ways these signals act on their target structures. Dopamine has functions other than signaling RPEs, and even for dopamine neurons that do signal RPEs, it can make sense to send different RPEs to different structures depending on the roles these structures play in producing reinforced behavior. This is beyond what we treat in any detail in this book, but vector-valued RPE signals make sense from the perspective of reinforcement learning when decisions can be decomposed into separate sub-decisions, or more generally, as a way to address the *structural* version of the credit assignment problem: How do you distribute credit for success (or blame for failure) of a decision among the many component structures that could have been involved in producing it? We say a bit more about this in Section 15.10 below.

The axons of most dopamine neurons make synaptic contact with neurons in the frontal cortex and the basal ganglia, areas of the brain involved in voluntary movement, decision making, learning, and cognitive functions such as planning. Because most ideas relating



Axonal arbor of a single neuron producing dopamine as a neurotransmitter. These axons make synaptic contacts with a huge number of dendrites of neurons in targeted brain areas.

Adapted from *The Journal of Neuroscience*, Matsuda, Furuta, Nakamura, Hioki, Fujiyama, Arai, and Kaneko, volume 29, 2009, page 451.

dopamine to reinforcement learning focus on the basal ganglia, and the connections from dopamine neurons are particularly dense there, we focus on the basal ganglia here. The basal ganglia are a collection of neuron groups, or nuclei, lying at the base of the forebrain. The main input structure of the basal ganglia is called the striatum. Essentially all of the cerebral cortex, among other structures, provides input to the striatum. The activity of cortical neurons conveys a wealth of information about sensory input, internal states, and motor activity. The axons of cortical neurons make synaptic contacts on the dendrites of the main input/output neurons of the striatum, called medium spiny neurons. Output from the striatum loops back via other basal ganglia nuclei and the thalamus to frontal areas of cortex, and to motor areas, making it possible for the striatum to influence movement, abstract decision processes, and reward processing. Two main subdivisions of the striatum are important for reinforcement learning: the dorsal striatum, primarily implicated in influencing action selection, and the ventral striatum, thought to be critical for different aspects of reward processing, including the assignment of affective value to sensations.

The dendrites of medium spiny neurons are covered with spines on whose tips the axons of neurons in the cortex make synaptic contact. Also making synaptic contact with these spines—in this case contacting the spine stems—are axons of dopamine neurons (Figure 15.1). This arrangement brings together presynaptic activity of cortical neurons,

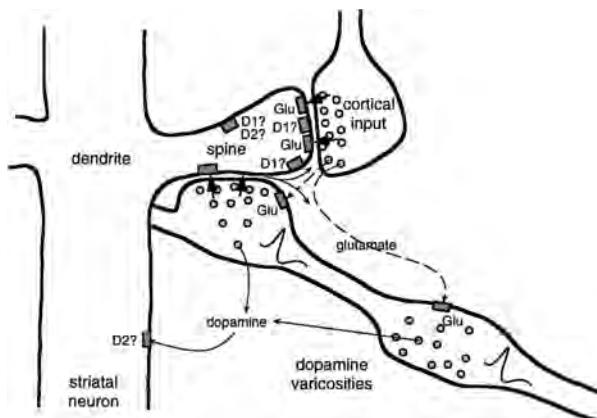


Figure 15.1: Spine of a striatal neuron showing input from both cortical and dopamine neurons. Axons of cortical neurons influence striatal neurons via corticostriatal synapses releasing the neurotransmitter glutamate at the tips of spines covering the dendrites of striatal neurons. An axon of a VTA or SNpc dopamine neuron is shown passing by the spine (from the lower right). “Dopamine varicosities” on this axon release dopamine at or near the spine stem, in an arrangement that brings together presynaptic input from cortex, postsynaptic activity of the striatal neuron, and dopamine, making it possible that several types of learning rules govern the plasticity of corticostriatal synapses. Each axon of a dopamine neuron makes synaptic contact with the stems of roughly 500,000 spines. Some of the complexity omitted from our discussion is shown here by other neurotransmitter pathways and multiple receptor types, such as D1 and D2 dopamine receptors by which dopamine can produce different effects at spines and other postsynaptic sites. From *Journal of Neurophysiology*, W. Schultz, vol. 80, 1998, page 10.

postsynaptic activity of medium spiny neurons, and input from dopamine neurons. What actually occurs at these spines is complex and not completely understood. Figure 15.1 hints at the complexity by showing two types of receptors for dopamine, receptors for glutamate—the neurotransmitter of the cortical inputs—and multiple ways that the various signals can interact. But evidence is mounting that changes in the efficacies of the synapses on the pathway from the cortex to the striatum, which neuroscientists call *corticostriatal synapses*, depend critically on appropriately-timed dopamine signals.

15.5 Experimental Support for the Reward Prediction Error Hypothesis

Dopamine neurons respond with bursts of activity to intense, novel, or unexpected visual and auditory stimuli that trigger eye and body movements, but very little of their activity is related to the movements themselves. This is surprising because degeneration of dopamine neurons is a cause of Parkinson’s disease, whose symptoms include motor disorders, particularly deficits in self-initiated movement. Motivated by the weak relationship between dopamine neuron activity and stimulus-triggered eye and body movements, Romo and Schultz (1990) and Schultz and Romo (1990) took the first steps toward the reward prediction error hypothesis by recording the activity of dopamine neurons and muscle activity while monkeys moved their arms.

They trained two monkeys to reach from a resting hand position into a bin containing a bit of apple, a piece of cookie, or a raisin, when the monkey saw and heard the bin’s door open. The monkey could then grab and bring the food to its mouth. After a monkey became good at this, it was trained on two additional tasks. The purpose of the first task was to see what dopamine neurons do when movements are self-initiated. The bin was left open but covered from above so that the monkey could not see inside but could reach in from below. No triggering stimuli were presented, and after the monkey reached for and ate the food morsel, the experimenter usually (though not always), silently and unseen by the monkey, replaced food in the bin by sticking it onto a rigid wire. Here too, the activity of the dopamine neurons Romo and Schultz monitored was not related to the monkey’s movements, but a large percentage of these neurons produced phasic responses whenever the monkey first touched a food morsel. These neurons did not respond when the monkey touched just the wire or explored the bin when no food was there. This was good evidence that the neurons were responding to the food and not to other aspects of the task.

The purpose of Romo and Schultz’s second task was to see what happens when movements are triggered by stimuli. This task used a different bin with a movable cover. The sight and sound of the bin opening triggered reaching movements to the bin. In this case, Romo and Schultz found that after some period of training, the dopamine neurons no longer responded to the touch of the food but instead responded to the sight and sound of the opening cover of the food bin. The phasic responses of these neurons had shifted from the reward itself to stimuli predicting the availability of the reward. In a followup study, Romo and Schultz found that most of the dopamine neurons whose activity they

monitored did not respond to the sight and sound of the bin opening outside the context of the behavioral task. These observations suggested that the dopamine neurons were responding neither to the initiation of a movement nor to the sensory properties of the stimuli, but were rather signaling an expectation of reward.

Schultz's group conducted many additional studies involving both SNpc and VTA dopamine neurons. A particular series of experiments was influential in suggesting that the phasic responses of dopamine neurons correspond to TD errors and not to simpler errors like those in the Rescorla–Wagner model (14.3). In the first of these experiments (Ljungberg, Apicella, and Schultz, 1992), monkeys were trained to depress a lever after a light was illuminated as a 'trigger cue' to obtain a drop of apple juice. As Romo and Schultz had observed earlier, many dopamine neurons initially responded to the reward—the drop of juice (Figure 15.2, top panel). But many of these neurons lost that reward response as training continued and developed responses instead to the illumination of the light that predicted the reward (Figure 15.2, middle panel). With continued training, lever pressing became faster while the number of dopamine neurons responding to the trigger cue decreased.

Following this study, the same monkeys were trained on a new task (Schultz, Apicella, and Ljungberg, 1993). Here the monkeys faced two levers, each with a light above it. Illuminating one of these lights was an 'instruction cue' indicating which of the two levers

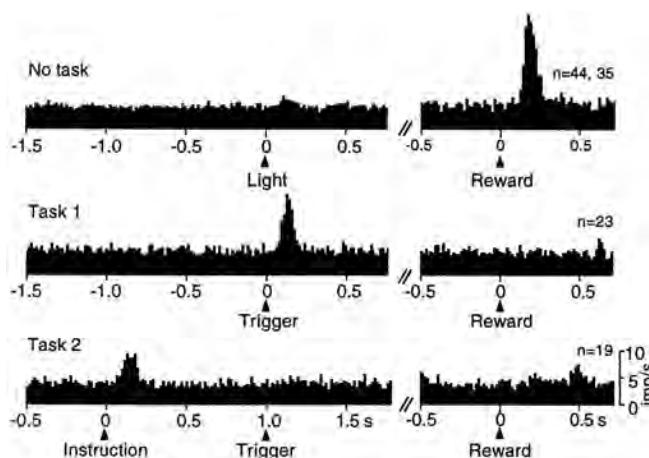


Figure 15.2: The response of dopamine neurons shifts from initial responses to primary reward to earlier predictive stimuli. These are plots of the number of action potentials produced by monitored dopamine neurons within small time intervals, averaged over all the monitored dopamine neurons (ranging from 23 to 44 neurons for these data). Top: dopamine neurons are activated by the unpredicted delivery of drop of apple juice. Middle: with learning, dopamine neurons developed responses to the reward-predicting trigger cue and lost responsiveness to the delivery of reward. Bottom: with the addition of an instruction cue preceding the trigger cue by 1 second, dopamine neurons shifted their responses from the trigger cue to the earlier instruction cue. From Schultz et al. (1995), MIT Press.

would produce a drop of apple juice. In this task, the instruction cue preceded the trigger cue of the previous task by a fixed interval of 1 second. The monkeys learned to withhold reaching until seeing the trigger cue, and dopamine neuron activity increased, but now the responses of the monitored dopamine neurons occurred almost exclusively to the earlier instruction cue and not to the trigger cue (Figure 15.2, bottom panel). Here again the number of dopamine neurons responding to the instruction cue was much reduced when the task was well learned. During learning across these tasks, dopamine neuron activity shifted from initially responding to the reward to responding to the earlier predictive stimuli, first progressing to the trigger stimulus then to the still earlier instruction cue. As responding moved earlier in time it disappeared from the later stimuli. This shifting of responses to earlier reward predictors, while losing responses to later predictors is a hallmark of TD learning (see, for example, Figure 14.2).

The task just described revealed another property of dopamine neuron activity shared with TD learning. The monkeys sometimes pressed the wrong key, that is, the key other than the instructed one, and consequently received no reward. In these trials, many of the dopamine neurons showed a sharp decrease in their firing rates below baseline shortly after the reward's usual time of delivery, and this happened without the availability of any external cue to mark the usual time of reward delivery (Figure 15.3). Somehow the

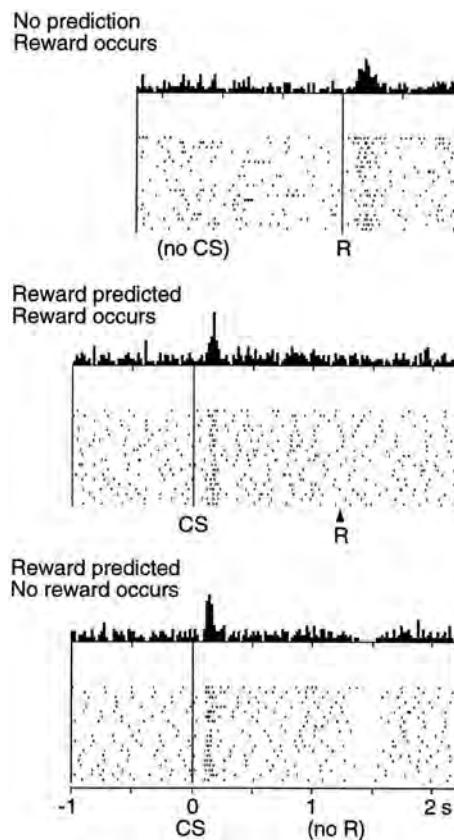


Figure 15.3: The response of dopamine neurons drops below baseline shortly after the time when an expected reward fails to occur. Top: dopamine neurons are activated by the unpredicted delivery of a drop of apple juice. Middle: dopamine neurons respond to a conditioned stimulus (CS) that predicts reward and do not respond to the reward itself. Bottom: when the reward predicted by the CS fails to occur, the activity of dopamine neurons drops below baseline shortly after the time the reward is expected to occur. At the top of each of these panels is shown the average number of action potentials produced by monitored dopamine neurons within small time intervals around the indicated times. The raster plots below show the activity patterns of the individual dopamine neurons that were monitored; each dot represents an action potential. From Schultz, Dayan, and Montague, A Neural Substrate of Prediction and Reward, *Science*, vol. 275, issue 5306, pages 1593-1598, March 14, 1997. Reprinted with permission from AAAS.

monkeys were internally keeping track of the timing of the reward. (Response timing is one area where the simplest version of TD learning needs to be modified to account for some of the details of the timing of dopamine neuron responses. We consider this issue in the following section.)

The observations from the studies described above led Schultz and his group to conclude that dopamine neurons respond to unpredicted rewards, to the earliest predictors of reward, and that dopamine neuron activity decreases below baseline if a reward, or a predictor of reward, does not occur at its expected time. Researchers familiar with reinforcement learning were quick to recognize that these results are strikingly similar to how the TD error behaves as the reinforcement signal in a TD algorithm. The next section explores this similarity by working through a specific example in detail.

15.6 TD Error/Dopamine Correspondence

This section explains the correspondence between the TD error δ and the phasic responses of dopamine neurons observed in the experiments just described. We examine how δ changes over the course of learning in a task something like the one described above where a monkey first sees an instruction cue and then a fixed time later has to respond correctly to a trigger cue in order to obtain reward. We use a simple idealized version of this task, but we go into a lot more detail than is usual because we want to emphasize the theoretical basis of the parallel between TD errors and dopamine neuron activity.

The first simplifying assumption is that the agent has already learned the actions required to obtain reward. Then its task is just to learn accurate predictions of future reward for the sequence of states it experiences. This is then a prediction task, or more technically, a policy-evaluation task: learning the value function for a fixed policy (Sections 4.1 and 6.1). The value function to be learned assigns to each state a value that predicts the return that will follow that state if the agent selects actions according to the given policy, where the return is the (possibly discounted) sum of all the future rewards. This is unrealistic as a model of the monkey’s situation because the monkey would likely learn these predictions at the same time that it is learning to act correctly (as would a reinforcement learning algorithm that learns policies as well as value functions, such as an actor–critic algorithm), but this scenario is simpler to describe than one in which a policy and a value function are learned simultaneously.

Now imagine that the agent’s experience divides into multiple trials, in each of which the same sequence of states repeats, with a distinct state occurring on each time step during the trial. Further imagine that the return being predicted is limited to the return over a trial, which makes a trial analogous to a reinforcement learning episode as we have defined it. In reality, of course, the returns being predicted are not confined to single trials, and the time interval between trials is an important factor in determining what an animal learns. This is true for TD learning as well, but here we assume that returns do not accumulate over multiple trials. Given this, then, a trial in experiments like those conducted by Schultz and colleagues is equivalent to an episode of reinforcement learning. (Though in this discussion, we will use the term trial instead of episode to relate better to the experiments.)

As usual, we also need to make an assumption about how states are represented as inputs to the learning algorithm, an assumption that influences how closely the TD error corresponds to dopamine neuron activity. We discuss this issue later, but for now we assume the same CSC representation used by Montague et al. (1996) in which there is a separate internal stimulus for each state visited at each time step in a trial. This reduces the process to the tabular case covered in the first part of this book. Finally, we assume that the agent uses $\text{TD}(0)$ to learn a value function, V , stored in a lookup table initialized to be zero for all the states. We also assume that this is a deterministic task and that the discount factor, γ , is very nearly one so that we can ignore it.

Figure 15.4 shows the time courses of R , V , and δ at several stages of learning in this policy-evaluation task. The time axes represent the time interval over which a sequence of states is visited in a trial (where for clarity we omit showing individual states). The reward signal is zero throughout each trial except when the agent reaches the rewarding state, shown near the right end of the time line, when the reward signal becomes some positive number, say R^* . The goal of TD learning is to predict the return for each state visited in a trial, which in this undiscounted case and given our assumption that predictions are confined to individual trials, is simply R^* for each state.

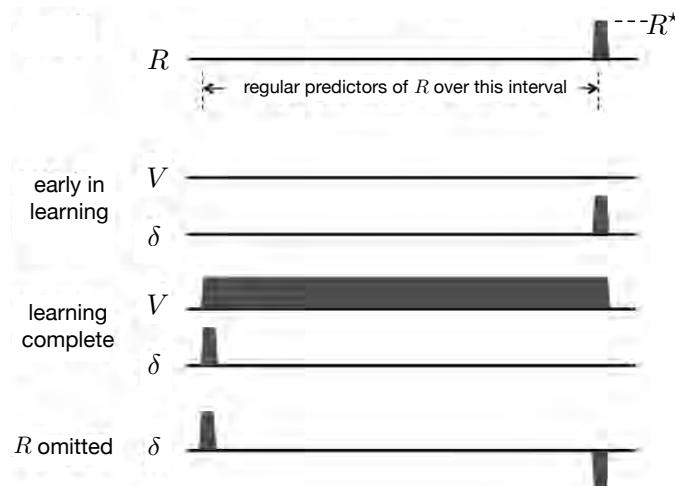


Figure 15.4: The behavior of the TD error δ during TD learning is consistent with features of the phasic activation of dopamine neurons. (Here δ is the TD error *available* at time t , i.e., δ_{t-1}). Top: a sequence of states, shown as an interval of regular predictors, is followed by a non-zero reward R^* . *Early in learning*: the initial value function, V , and initial δ , which at first is equal to R^* . *Learning complete*: the value function accurately predicts future reward, δ is positive at the earliest predictive state, and $\delta = 0$ at the time of the non-zero reward. *R^* omitted*: at the time the predicted reward is omitted, δ becomes negative. See text for a complete explanation of why this happens.

Preceding the rewarding state is a sequence of reward-predicting states, with the *earliest reward-predicting state* shown near the left end of the time line. This is like the state near the start of a trial, for example like the state marked by the instruction cue in a trial of the monkey experiment of Schultz et al. (1993) described above. It is the first state in a trial that reliably predicts that trial's reward. (Of course, in reality states visited on preceding trials are even earlier reward-predicting states, but because we are confining predictions to individual trials, these do not qualify as predictors of *this* trial's reward. Below we give a more satisfactory, though more abstract, description of an earliest reward-predicting state.) The *latest reward-predicting state* in a trial is the state immediately preceding the trial's rewarding state. This is the state near the far right end of the time line in Figure 15.4. Note that the rewarding state of a trial does not predict the return for that trial: the value of this state would come to predict the return over all the *following* trials, which here we are assuming to be zero in this episodic formulation.

Figure 15.4 shows the first-trial time courses of V and δ as the graphs labeled 'early in learning.' Because the reward signal is zero throughout the trial except when the rewarding state is reached, and all the V -values are zero, the TD error is also zero until it becomes R^* at the rewarding state. This follows because $\delta_{t-1} = R_t + V_t - V_{t-1} = R_t + 0 - 0 = R_t$, which is zero until it equals R^* when the reward occurs. Here V_t and V_{t-1} are respectively the estimated values of the states visited at times t and $t - 1$ in a trial. The TD error at this stage of learning is analogous to a dopamine neuron responding to an unpredicted reward (e.g., a drop of apple juice) at the start of training.

Throughout this first trial and all successive trials, TD(0) updates occur at each state transition as described in Chapter 6. This successively increases the values of the reward-predicting states, with the increases spreading backwards from the rewarding state, until the values converge to the correct return predictions. In this case (because we are assuming no discounting) the correct predictions are equal to R^* for all the reward-predicting states. This can be seen in Figure 15.4 as the graph of V labeled 'learning complete' where the values of all the states from the earliest to the latest reward-predicting states all equal R^* . The values of the states preceding the earliest reward-predicting state remain low (which Figure 15.4 shows as zero) because they are not reliable predictors of reward.

When learning is complete, that is, when V attains its correct values, the TD errors associated with transitions *from* any reward-predicting state are zero because the predictions are now accurate. This is because for a transition from a reward-predicting state to another reward-predicting state, we have $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + R^* - R^* = 0$, and for the transition from the latest reward-predicting state to the rewarding state, we have $\delta_{t-1} = R_t + V_t - V_{t-1} = R^* + 0 - R^* = 0$. On the other hand, the TD error on a transition from any state *to* the earliest reward-predicting state is positive because of the mismatch between this state's low value and the larger value of the following reward-predicting state. Indeed, if the value of a state preceding the earliest reward-predicting state were zero, then after the transition to the earliest reward-predicting state, we would have that $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + R^* - 0 = R^*$. The 'learning complete' graph of δ in Figure 15.4 shows this positive value at the earliest reward-predicting state, and zeros everywhere else.

The positive TD error upon transitioning to the earliest reward-predicting state is analogous to the persistence of dopamine responses to the earliest stimuli predicting reward. By the same token, when learning is complete, a transition from the latest reward-predicting state to the rewarding state produces a zero TD error because the latest reward-predicting state's value, being correct, cancels the reward. This parallels the observation that fewer dopamine neurons generate a phasic response to a fully predicted reward than to an unpredicted reward.

After learning, if the reward is suddenly omitted, the TD error goes negative at the usual time of reward because the value of the latest reward-predicting state is then too high: $\delta_{t-1} = R_t + V_t - V_{t-1} = 0 + 0 - R^* = -R^*$, as shown at the right end of the '*R omitted*' graph of δ in Figure 15.4. This is like dopamine neuron activity decreasing below baseline at the time an expected reward is omitted as seen in the experiment of Schultz et al. (1993) described above and shown in Figure 15.3.

The idea of an *earliest reward-predicting state* deserves more attention. In the scenario described above, because experience is divided into trials, and we assumed that predictions are confined to individual trials, the earliest reward-predicting state is always the first state of a trial. Clearly this is artificial. A more general way to think of an earliest reward-predicting state is that it is an *unpredicted predictor* of reward, and there can be many such states. In an animal's life, many different states may precede an earliest reward-predicting state. However, because these states are more often followed by *other* states that do not predict reward, their reward-predicting powers, that is, their values, remain low. A TD algorithm, if operating throughout the animal's life, would update the values of these states too, but the updates would not consistently accumulate because, by assumption, none of these states reliably precedes an earliest reward-predicting state. If any of them did, they would be reward-predicting states as well. This might explain why with overtraining, dopamine responses decrease to even the earliest reward-predicting stimulus in a trial. With overtraining one would expect that even a formerly-unpredicted predictor state would become predicted by stimuli associated with earlier states: the animal's interaction with its environment both inside and outside of an experimental task would become commonplace. Upon breaking this routine with the introduction of a new task, however, one would see TD errors reappear, as indeed is observed in dopamine neuron activity.

The example described above explains why the TD error shares key features with the phasic activity of dopamine neurons when the animal is learning in a task similar to the idealized task of our example. But not every property of the phasic activity of dopamine neurons coincides so neatly with properties of δ . One of the most troubling discrepancies involves what happens when a reward occurs *earlier* than expected. We have seen that the omission of an expected reward produces a negative prediction error at the reward's expected time, which corresponds to the activity of dopamine neurons decreasing below baseline when this happens. If the reward arrives later than expected, it is then an unexpected reward and generates a positive prediction error. This happens with both TD errors and dopamine neuron responses. But when reward arrives earlier than expected, dopamine neurons do not do what the TD error does—at least with the CSC representation used by Montague et al. (1996) and by us in our example. Dopamine

neurons do respond to the early reward, which is consistent with a positive TD error because the reward is not predicted to occur then. However, at the later time when the reward is expected but omitted, the TD error is negative whereas, in contrast to this prediction, dopamine neuron activity does not drop below baseline in the way the TD model predicts (Hollerman and Schultz, 1998). Something more complicated is going on in the animal's brain than simply TD learning with a CSC representation.

Some of the mismatches between the TD error and dopamine neuron activity can be addressed by selecting suitable parameter values for the TD algorithm and by using stimulus representations other than the CSC representation. For instance, to address the early-reward mismatch just described, Suri and Schultz (1999) proposed a CSC representation in which the sequences of internal signals initiated by earlier stimuli are cancelled by the occurrence of a reward. Another proposal by Daw, Courville, and Touretzky (2006) is that the brain's TD system uses representations produced by statistical modeling carried out in sensory cortex rather than simpler representations based on raw sensory input. Ludvig, Sutton, and Kehoe (2008) found that TD learning with a microstimulus (MS) representation (Figure 14.1) fits the activity of dopamine neurons in the early-reward and other situations better than when a CSC representation is used. Pan, Schmidt, Wickens, and Hyland (2005) found that even with the CSC representation, prolonged eligibility traces improve the fit of the TD error to some aspects of dopamine neuron activity. In general, many fine details of TD-error behavior depend on subtle interactions between eligibility traces, discounting, and stimulus representations. Findings like these elaborate and refine the reward prediction error hypothesis without refuting its core claim that the phasic activity of dopamine neurons is well characterized as signaling TD errors.

On the other hand, there are other discrepancies between the TD theory and experimental data that are not so easily accommodated by selecting parameter values and stimulus representations (we mention some of these discrepancies in the Bibliographical and Historical Remarks section at the end of this chapter), and more mismatches are likely to be discovered as neuroscientists conduct ever more refined experiments. But the reward prediction error hypothesis has been functioning very effectively as a catalyst for improving our understanding of how the brain's reward system works. Intricate experiments have been designed to validate or refute predictions derived from the hypothesis, and experimental results have, in turn, led to refinement and elaboration of the TD error/dopamine hypothesis.

A remarkable aspect of these developments is that the reinforcement learning algorithms and theory that connect so well with properties of the dopamine system were developed from a computational perspective in total absence of any knowledge about the relevant properties of dopamine neurons—remember, TD learning and its connections to optimal control and dynamic programming were developed many years before any of the experiments were conducted that revealed the TD-like nature of dopamine neuron activity. This unplanned correspondence, despite not being perfect, suggests that the TD error/dopamine parallel captures something significant about brain reward processes.

In addition to accounting for many features of the phasic activity of dopamine neurons, the reward prediction error hypothesis links neuroscience to other aspects of reinforcement

learning, in particular, to learning algorithms that use TD errors as reinforcement signals. Neuroscience is still far from reaching complete understanding of the circuits, molecular mechanisms, and functions of the phasic activity of dopamine neurons, but evidence supporting the reward prediction error hypothesis, along with evidence that phasic dopamine responses are reinforcement signals for learning, suggest that the brain might implement something like an actor–critic algorithm in which TD errors play critical roles. Other reinforcement learning algorithms are plausible candidates too, but actor–critic algorithms fit the anatomy and physiology of the mammalian brain particularly well, as we describe in the following two sections.

15.7 Neural Actor–Critic

Actor–critic algorithms learn both policies and value functions. The ‘actor’ is the component that learns policies, and the ‘critic’ is the component that learns about whatever policy is currently being followed by the actor in order to ‘criticize’ the actor’s action choices. The critic uses a TD algorithm to learn the state-value function for the actor’s current policy. The value function allows the critic to critique the actor’s action choices by sending TD errors, δ , to the actor. A positive δ means that the action was ‘good’ because it led to a state with a better-than-expected value; a negative δ means that the action was ‘bad’ because it led to a state with a worse-than-expected value. Based on these critiques, the actor continually updates its policy.

Two distinctive features of actor–critic algorithms are responsible for thinking that the brain might implement an algorithm like this. First, the two components of an actor–critic algorithm—the actor and the critic—suggest that two parts of the striatum—the dorsal and ventral subdivisions (Section 15.4), both critical for reward-based learning—may function respectively something like an actor and a critic. A second property of actor–critic algorithms that suggests a brain implementation is that the TD error has the dual role of being the reinforcement signal for both the actor and the critic, though it has a different influence on learning in each of these components. This fits well with several properties of the neural circuitry: axons of dopamine neurons target both the dorsal and ventral subdivisions of the striatum; dopamine appears to be critical for modulating synaptic plasticity in both structures; and how a neuromodulator such as dopamine acts on a target structure depends on properties of the target structure and not just on properties of the neuromodulator.

Section 13.5 presents actor–critic algorithms as policy gradient methods, but the actor–critic algorithm of Barto, Sutton, and Anderson (1983) was simpler and was presented as an artificial neural network (ANN). Here we describe an ANN implementation something like that of Barto et al., and we follow Takahashi, Schoenbaum, and Niv (2008) in giving a schematic proposal for how this ANN might be implemented by real neural networks in the brain. We postpone discussion of the actor and critic learning rules until Section 15.8, where we present them as special cases of the policy-gradient formulation and discuss what they suggest about how dopamine might modulate synaptic plasticity.

Figure 15.5a shows an implementation of an actor-critic algorithm as an ANN with component networks implementing the actor and the critic. The critic consists of a single neuron-like unit, V , whose output activity represents state values, and a component shown as the diamond labeled TD that computes TD errors by combining V 's output with reward signals and with previous state values (as suggested by the loop from the TD diamond to itself). The actor network has a single layer of k actor units labeled A_i , $i = 1, \dots, k$. The output of each actor unit is a component of a k -dimensional action vector. An alternative is that there are k separate actions, one commanded by each actor unit, that compete with one another to be executed, but here we will think of the entire A -vector as an action.

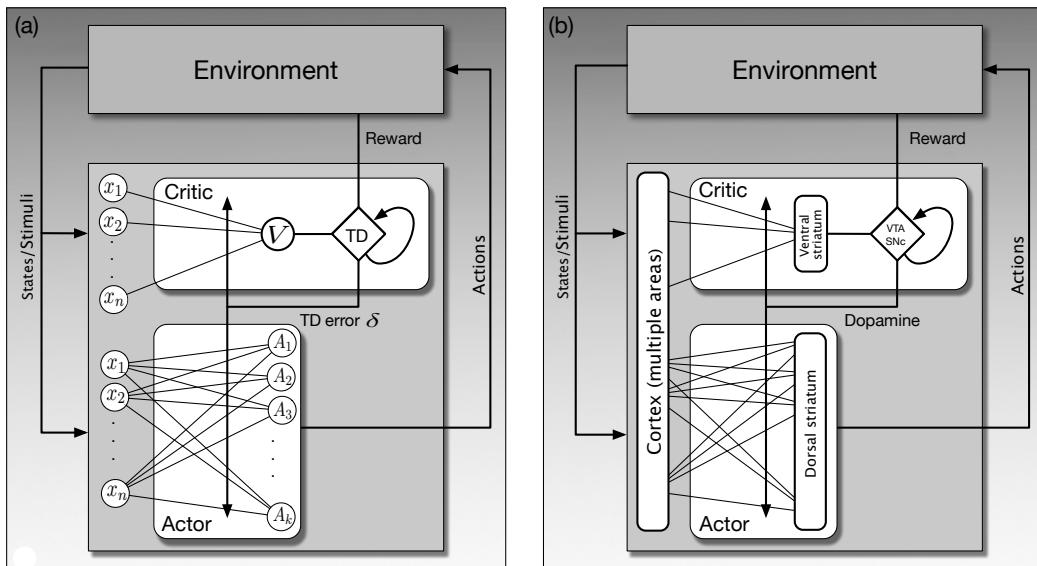


Figure 15.5: Actor-critic ANN and a hypothetical neural implementation. a) Actor-critic algorithm as an ANN. The actor adjusts a policy based on the TD error δ it receives from the critic; the critic adjusts state-value parameters using the same δ . The critic produces a TD error from the reward signal, R , and the current change in its estimate of state values. The actor does not have direct access to the reward signal, and the critic does not have direct access to the action. b) Hypothetical neural implementation of an actor-critic algorithm. The actor and the value-learning part of the critic are respectively placed in the dorsal and ventral subdivisions of the striatum. The TD error is transmitted by dopamine neurons located in the VTA and SNpc to modulate changes in synaptic efficacies of input from cortical areas to the ventral and dorsal striatum. Adapted from *Frontiers in Neuroscience*, vol. 2(1), 2008, Y. Takahashi, G. Schoenbaum, and Y. Niv, Silencing the critics: Understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an Actor/Critic model.

Both the critic and actor networks receive input consisting of multiple features representing the state of the agent’s environment. (Recall from Chapter 1 that the environment of a reinforcement learning agent includes components both inside and outside of the ‘organism’ containing the agent.) The figure shows these features as the circles labeled x_1, x_2, \dots, x_n , shown twice just to keep the figure simple. A weight representing the efficacy of a synapse is associated with each connection from each feature x_i to the critic unit, V , and to each of the action units, A_i . The weights in the critic network parameterize the value function, and the weights in the actor network parameterize the policy. The networks learn as these weights change according to the critic and actor learning rules that we describe in the following section.

The TD error produced by circuitry in the critic is the reinforcement signal for changing the weights in both the critic and the actor networks. This is shown in Figure 15.5a by the line labeled ‘TD error δ ’ extending across all of the connections in the critic and actor networks. This aspect of the network implementation, together with the reward prediction error hypothesis and the fact that the activity of dopamine neurons is so widely distributed by the extensive axonal arbors of these neurons, suggests that an actor–critic network something like this may not be too farfetched as a hypothesis about how reward-related learning might happen in the brain.

Figure 15.5b suggests—very schematically—how the ANN on the figure’s left might map onto structures in the brain according to the hypothesis of Takahashi et al. (2008). The hypothesis puts the actor and the value-learning part of the critic respectively in the dorsal and ventral subdivisions of the striatum, the input structure of the basal ganglia. Recall from Section 15.4 that the dorsal striatum is primarily implicated in influencing action selection, and the ventral striatum is thought to be critical for different aspects of reward processing, including the assignment of affective value to sensations. The cerebral cortex, along with other structures, sends input to the striatum conveying information about stimuli, internal states, and motor activity.

In this hypothetical actor–critic brain implementation, the ventral striatum sends value information to the VTA and SNpc, where dopamine neurons in these nuclei combine it with information about reward to generate activity corresponding to TD errors (though exactly how dopaminergic neurons calculate these errors is not yet understood). The ‘TD error δ ’ line in Figure 15.5a becomes the line labeled ‘Dopamine’ in Figure 15.5b, which represents the widely branching axons of dopamine neurons whose cell bodies are in the VTA and SNpc. Referring back to Figure 15.1, these axons make synaptic contact with the spines on the dendrites of medium spiny neurons, the main input/output neurons of both the dorsal and ventral divisions of the striatum. Axons of the cortical neurons that send input to the striatum make synaptic contact on the tips of these spines. According to the hypothesis, it is at these spines where changes in the efficacies of the synapses from cortical regions to the striatum are governed by learning rules that critically depend on a reinforcement signal supplied by dopamine.

An important implication of the hypothesis illustrated in Figure 15.5b is that the dopamine signal is not the ‘master’ reward signal like the scalar R_t of reinforcement learning. In fact, the hypothesis implies that one should not necessarily be able to probe the brain and record any signal like R_t in the activity of any single neuron.

Many interconnected neural systems generate reward-related information, with different structures being recruited depending on different types of rewards. Dopamine neurons receive information from many different brain areas, so the input to the SNpc and VTA labeled ‘Reward’ in Figure 15.5b should be thought of as vector of reward-related information arriving to neurons in these nuclei along multiple input channels. What the theoretical scalar reward signal R_t might correspond to, then, is the net contribution of all reward-related information to dopamine neuron activity. It is the result of a pattern of activity across many neurons in different areas of the brain.

Although the actor–critic neural implementation illustrated in Figure 15.5b may be correct on some counts, it clearly needs to be refined, extended, and modified to qualify as a full-fledged model of the function of the phasic activity of dopamine neurons. The Historical and Bibliographic Remarks section at the end of this chapter cites publications that discuss in more detail both empirical support for this hypothesis and places where it falls short. We now look in detail at what the actor and critic learning algorithms suggest about the rules governing changes in synaptic efficacies of corticostriatal synapses.

15.8 Actor and Critic Learning Rules

If the brain does implement something like the actor–critic algorithm—and assuming populations of dopamine neurons broadcast a common reinforcement signal to the corticostriatal synapses of both the dorsal and ventral striatum as illustrated in Figure 15.5b (which is likely an oversimplification as we mentioned above)—then this reinforcement signal affects the synapses of these two structures in different ways. The learning rules for the critic and the actor use the same reinforcement signal, the TD error δ , but its effect on learning is different for these two components. The TD error (combined with eligibility traces) tells the actor how to update action probabilities in order to reach higher-valued states. Learning by the actor is like instrumental conditioning using a Law-of-Effect-type learning rule (Section 1.7): the actor works to keep δ as positive as possible. On the other hand, the TD error (when combined with eligibility traces) tells the critic the direction and magnitude in which to change the parameters of the value function in order to improve its predictive accuracy. The critic works to reduce δ ’s magnitude to be as close to zero as possible using a learning rule like the TD model of classical conditioning (Section 14.2). The difference between the critic and actor learning rules is relatively simple, but this difference has a profound effect on learning and is essential to how the actor–critic algorithm works. The difference lies solely in the eligibility traces each type of learning rule uses.

More than one set of learning rules can be used in actor–critic neural networks like those in Figure 15.5b but, to be specific, here we focus on the actor–critic algorithm for continuing problems with eligibility traces presented in Section 13.6. On each transition from state S_t to state S_{t+1} , taking action A_t and receiving reward R_{t+1} , that algorithm computes the TD error (δ) and then updates the eligibility trace vectors (\mathbf{z}_t^w and \mathbf{z}_t^θ) and

the parameters for the critic and actor (\mathbf{w} and $\boldsymbol{\theta}$), according to

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}), \\ \mathbf{z}_t^{\mathbf{w}} &= \gamma \lambda^{\mathbf{w}} \mathbf{z}_{t-1}^{\mathbf{w}} + \nabla \hat{v}(S_t, \mathbf{w}), \\ \mathbf{z}_t^{\boldsymbol{\theta}} &= \gamma \lambda^{\boldsymbol{\theta}} \mathbf{z}_{t-1}^{\boldsymbol{\theta}} + \nabla \ln \pi(A_t | S_t, \boldsymbol{\theta}), \\ \mathbf{w} &\leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta_t \mathbf{z}_t^{\mathbf{w}}, \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta_t \mathbf{z}_t^{\boldsymbol{\theta}},\end{aligned}$$

where $\gamma \in [0, 1]$ is a discount-rate parameter, $\lambda^{\mathbf{w}} \in [0, 1]$ and $\lambda^{\boldsymbol{\theta}} \in [0, 1]$ are bootstrapping parameters for the critic and the actor respectively, and $\alpha^{\mathbf{w}} > 0$ and $\alpha^{\boldsymbol{\theta}} > 0$ are analogous step-size parameters.

Think of the approximate value function \hat{v} as the output of a single linear neuron-like unit, called the *critic unit* and labeled V in Figure 15.5a. Then the value function is a linear function of the feature-vector representation of state s , $\mathbf{x}(s) = (x_1(s), \dots, x_n(s))^{\top}$, parameterized by a weight vector $\mathbf{w} = (w_1, \dots, w_n)^{\top}$:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^{\top} \mathbf{x}(s). \quad (15.1)$$

Each $x_i(s)$ is like the presynaptic signal to a neuron's synapse whose efficacy is w_i . The weights of the critic are incremented according to the rule above by $\alpha^{\mathbf{w}} \delta_t \mathbf{z}_t^{\mathbf{w}}$, where the reinforcement signal, δ_t , corresponds to a dopamine signal being broadcast to all of the critic unit's synapses. The eligibility trace vector, $\mathbf{z}_t^{\mathbf{w}}$, for the critic unit is a trace (average of recent values) of $\nabla \hat{v}(S_t, \mathbf{w})$. Because $\hat{v}(s, \mathbf{w})$ is linear in the weights, $\nabla \hat{v}(S_t, \mathbf{w}) = \mathbf{x}(S_t)$.

In neural terms, this means that each synapse has its own eligibility trace, which is one component of the vector $\mathbf{z}_t^{\mathbf{w}}$. A synapse's eligibility trace accumulates according to the level of activity arriving at that synapse, that is, the level of presynaptic activity, represented here by the component of the feature vector $\mathbf{x}(S_t)$ arriving at that synapse. The trace otherwise decays toward zero at a rate governed by the fraction $\lambda^{\mathbf{w}}$. A synapse is *eligible for modification* as long as its eligibility trace is non-zero. How the synapse's efficacy is actually modified depends on the reinforcement signals that arrive while the synapse is eligible. We call eligibility traces like these of the critic unit's synapses *non-contingent eligibility traces* because they only depend on presynaptic activity and are not contingent in any way on postsynaptic activity.

The non-contingent eligibility traces of the critic unit's synapses mean that the critic unit's learning rule is essentially the TD model of classical conditioning described in Section 14.2. With the definition we have given above of the critic unit and its learning rule, the critic in Figure 15.5a is the same as the critic in the ANN actor-critic of Barto et al. (1983). Clearly, a critic like this consisting of just one linear neuron-like unit is the simplest starting point; this critic unit is a proxy for a more complicated neural network able to learn value functions of greater complexity.

The actor in Figure 15.5a is a one-layer network of k neuron-like actor units, each receiving at time t the same feature vector, $\mathbf{x}(S_t)$, that the critic unit receives. Each actor unit j , $j = 1, \dots, k$, has its own weight vector, $\boldsymbol{\theta}_j$, but because the actor units are all identical, we describe just one of the units and omit the subscript. One way for these

units to follow the actor–critic algorithm given in the equations above is for each to be a *Bernoulli-logistic unit*. This means that the output of each actor unit at each time is a random variable, A_t , taking value 0 or 1. Think of value 1 as the neuron firing, that is, emitting an action potential. The weighted sum, $\boldsymbol{\theta}^\top \mathbf{x}(S_t)$, of a unit’s input vector determines the unit’s action probabilities via the exponential soft-max distribution (13.2), which for two actions is the logistic function:

$$\pi(1|s, \boldsymbol{\theta}) = 1 - \pi(0|s, \boldsymbol{\theta}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^\top \mathbf{x}(s))}. \quad (15.2)$$

The weights of each actor unit are incremented, as above, by: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta_t \mathbf{z}_t^{\boldsymbol{\theta}}$, where δ again corresponds to the dopamine signal: the same reinforcement signal that is sent to all the critic unit’s synapses. Figure 15.5a shows δ_t being broadcast to all the synapses of all the actor units (which makes this actor network a *team* of reinforcement learning agents, something we discuss in Section 15.10 below). The actor eligibility trace vector $\mathbf{z}_t^{\boldsymbol{\theta}}$ is a trace (average of recent values) of $\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$. To understand this eligibility trace refer to Exercise 13.5, which defines this kind of unit and asks you to give a learning rule for it. That exercise asked you to express $\nabla \ln \pi(a|s, \boldsymbol{\theta})$ in terms of a , $\mathbf{x}(s)$, and $\pi(a|s, \boldsymbol{\theta})$ (for arbitrary state s and action a) by calculating the gradient. For the action and state actually occurring at time t , the answer is

$$\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}) = (A_t - \pi(1|S_t, \boldsymbol{\theta})) \mathbf{x}(S_t). \quad (15.3)$$

Unlike the non-contingent eligibility trace of a critic synapse that only accumulates the presynaptic activity $\mathbf{x}(S_t)$, the eligibility trace of an actor unit’s synapse in addition depends on the activity of the actor unit itself. We call this a *contingent eligibility trace* because it is contingent on this postsynaptic activity. The eligibility trace at each synapse continually decays, but increments or decrements depending on the activity of the presynaptic neuron *and* whether or not the postsynaptic neuron fires. The factor $A_t - \pi(1|S_t, \boldsymbol{\theta})$ in (15.3) is positive when $A_t = 1$ and negative otherwise. *The postsynaptic contingency in the eligibility traces of actor units is the only difference between the critic and actor learning rules.* By keeping information about what actions were taken in what states, contingent eligibility traces allow credit for reward (positive δ), or blame for punishment (negative δ), to be apportioned among the policy parameters (the efficacies of the actor units’ synapses) according to the contributions these parameters made to the units’ outputs that could have influenced later values of δ . Contingent eligibility traces mark the synapses as to how they should be modified to alter the units’ future responses to favor positive values of δ .

What do the critic and actor learning rules suggest about how efficacies of corticostriatal synapses change? Both learning rules are related to Donald Hebb’s classic proposal that whenever a presynaptic signal participates in activating the postsynaptic neuron, the synapse’s efficacy increases (Hebb, 1949). The critic and actor learning rules share with Hebb’s proposal the idea that changes in a synapse’s efficacy depend on the interaction of several factors. In the critic learning rule the interaction is between the reinforcement signal δ and eligibility traces that depend only on presynaptic signals. Neuroscientists call this a *two-factor learning rule* because the interaction is between two signals or

quantities. The actor learning rule, on the other hand, is a *three-factor learning rule* because, in addition to depending on δ , its eligibility traces depend on both presynaptic and postsynaptic activity. Unlike Hebb's proposal, however, the relative timing of the factors is critical to how synaptic efficacies change, with eligibility traces intervening to allow the reinforcement signal to affect synapses that were active in the recent past.

Some subtleties about signal timing for the actor and critic learning rules deserve closer attention. In defining the neuron-like actor and critic units, we ignored the small amount of time it takes synaptic input to effect the firing of a real neuron. When an action potential from the presynaptic neuron arrives at a synapse, neurotransmitter molecules are released that diffuse across the synaptic cleft to the postsynaptic neuron, where they bind to receptors on the postsynaptic neuron's surface; this activates molecular machinery that causes the postsynaptic neuron to fire (or to inhibit its firing in the case of inhibitory synaptic input). This process can take several tens of milliseconds. According to (15.1) and (15.2), though, the input to a critic and actor unit instantaneously produces the unit's output. Ignoring activation time like this is common in abstract models of Hebbian-style plasticity in which synaptic efficacies change according to a simple product of simultaneous pre- and postsynaptic activity. More realistic models must take activation time into account.

Activation time is especially important for a more realistic actor unit because it influences how contingent eligibility traces have to work in order to properly apportion credit for reinforcement to the appropriate synapses. The expression $(A_t - \pi(1|S_t, \theta))\mathbf{x}(S_t)$ defining contingent eligibility traces for the actor unit's learning rule given above includes the postsynaptic factor $(A_t - \pi(1|S_t, \theta))$ and the presynaptic factor $\mathbf{x}(S_t)$. This works because by ignoring activation time, the presynaptic activity $\mathbf{x}(S_t)$ participates in *causing* the postsynaptic activity appearing in $(A_t - \pi(1|S_t, \theta))$. To assign credit for reinforcement correctly, the presynaptic factor defining the eligibility trace must be a cause of the postsynaptic factor that also defines the trace. Contingent eligibility traces for a more realistic actor unit would have to take activation time into account. (Activation time should not be confused with the time required for a neuron to receive a reinforcement signal influenced by that neuron's activity. The function of eligibility traces is to span this time interval which is generally much longer than the activation time. We discuss this further in the following section.)

There are hints from neuroscience for how this process might work in the brain. Neuroscientists have discovered a form of Hebbian plasticity called *spike-timing-dependent plasticity* (STDP) that lends plausibility to the existence of actor-like synaptic plasticity in the brain. STDP is a Hebbian-style plasticity, but changes in a synapse's efficacy depend on the relative timing of presynaptic and postsynaptic action potentials. The dependence can take different forms, but in the one most studied, a synapse increases in strength if spikes incoming via that synapse arrive shortly before the postsynaptic neuron fires. If the timing relation is reversed, with a presynaptic spike arriving shortly after the postsynaptic neuron fires, then the strength of the synapse decreases. STDP is a type of Hebbian plasticity that takes the activation time of a neuron into account, which is one of the ingredients needed for actor-like learning.

The discovery of STDP has led neuroscientists to investigate the possibility of a three-factor form of STDP in which neuromodulatory input must follow appropriately-timed pre- and postsynaptic spikes. This form of synaptic plasticity, called *reward-modulated STDP*, is much like the actor learning rule discussed here. Synaptic changes that would be produced by regular STDP only occur if there is neuromodulatory input within a time window after a presynaptic spike is closely followed by a postsynaptic spike. Evidence is accumulating that reward-modulated STDP occurs at the spines of medium spiny neurons of the dorsal striatum, with dopamine providing the neuromodulatory factor—the sites where actor learning takes place in the hypothetical neural implementation of an actor–critic algorithm illustrated in Figure 15.5b. Experiments have demonstrated reward-modulated STDP in which lasting changes in the efficacies of corticostriatal synapses occur only if a neuromodulatory pulse arrives within a time window that can last up to 10 seconds after a presynaptic spike is closely followed by a postsynaptic spike (Yagishita et al. 2014). Although the evidence is indirect, these experiments point to the existence of contingent eligibility traces having prolonged time courses. The molecular mechanisms producing these traces, as well as the much shorter traces that likely underly STDP, are not yet understood, but research focusing on time-dependent and neuromodulator-dependent synaptic plasticity is continuing.

The neuron-like actor unit that we have described here, with its Law-of-Effect-style learning rule, appeared in somewhat simpler form in the actor–critic network of Barto et al. (1983). That network was inspired by the “hedonistic neuron” hypothesis proposed by physiologist A. H. Klopff (1972, 1982). Not all the details of Klopff’s hypothesis are consistent with what has been learned about synaptic plasticity, but the discovery of STDP and the growing evidence for a reward-modulated form of STDP suggest that Klopff’s ideas may not have been far off the mark. We discuss Klopff’s hedonistic neuron hypothesis next.

15.9 Hedonistic Neurons

In his hedonistic neuron hypothesis, Klopff (1972, 1982) conjectured that individual neurons seek to maximize the difference between synaptic input treated as rewarding and synaptic input treated as punishing by adjusting the efficacies of their synapses on the basis of rewarding or punishing consequences of their own action potentials. In other words, individual neurons can be trained with response-contingent reinforcement like an animal can be trained in an instrumental conditioning task. His hypothesis included the idea that rewards and punishments are conveyed to a neuron via the same synaptic input that excites or inhibits the neuron’s spike-generating activity. (Had Klopff known what we know today about neuromodulatory systems, he might have assigned the reinforcing role to neuromodulatory input, but he wanted to avoid any centralized source of training information.) Synaptically-local traces of past pre- and postsynaptic activity had the key function in Klopff’s hypothesis of making synapses *eligible*—the term he introduced—for modification by later reward or punishment. He conjectured that these traces are implemented by molecular mechanisms local to each synapse and therefore different from the electrical activity of both the pre- and the postsynaptic neurons. In

the Bibliographical and Historical Remarks section of this chapter we bring attention to some similar proposals made by others.

Klopf specifically conjectured that synaptic efficacies change in the following way. When a neuron fires an action potential, all of its synapses that were active in contributing to that action potential become eligible to undergo changes in their efficacies. If the action potential is followed within an appropriate time period by an increase of reward, the efficacies of all the eligible synapses increase. Symmetrically, if the action potential is followed within an appropriate time period by an increase of punishment, the efficacies of eligible synapses decrease. This is implemented by triggering an eligibility trace at a synapse upon a coincidence of presynaptic and postsynaptic activity (or more exactly, upon pairing of presynaptic activity with the postsynaptic activity that that presynaptic activity participates in causing)—what we call a contingent eligibility trace. This is essentially the three-factor learning rule of an actor unit described in the previous section.

The shape and time course of an eligibility trace in Klopf's theory reflects the durations of the many feedback loops in which the neuron is embedded, some of which lie entirely within the brain and body of the organism, while others extend out through the organism's external environment as mediated by its motor and sensory systems. His idea was that the shape of a synaptic eligibility trace is like a histogram of the durations of the feedback loops in which the neuron is embedded. The peak of an eligibility trace would then occur at the duration of the most prevalent feedback loops in which that neuron participates. The eligibility traces used by algorithms described in this book are simplified versions of Klopf's original idea, being exponentially (or geometrically) decreasing functions controlled by the parameters λ and γ . This simplifies simulations as well as theory, but we regard these simple eligibility traces as a placeholders for traces closer to Klopf's original conception, which would have computational advantages in complex reinforcement learning systems by refining the credit-assignment process.

Klopf's hedonistic neuron hypothesis is not as implausible as it may at first appear. A well-studied example of a single cell that seeks some stimuli and avoids others is the bacterium *Escherichia coli*. The movement of this single-cell organism is influenced by chemical stimuli in its environment, behavior known as chemotaxis. It swims in its liquid environment by rotating hairlike structures called flagella attached to its surface. (Yes, it rotates them!) Molecules in the bacterium's environment bind to receptors on its surface. Binding events modulate the frequency with which the bacterium reverses flagellar rotation. Each reversal causes the bacterium to tumble in place and then head off in a random new direction. A little chemical memory and computation causes the frequency of flagellar reversal to decrease when the bacterium swims toward higher concentrations of molecules it needs to survive (attractants) and increase when the bacterium swims toward higher concentrations of molecules that are harmful (repellants). The result is that the bacterium tends to persist in swimming up attractant gradients and tends to avoid swimming up repellent gradients.

The chemotactic behavior just described is called klinokinesis. It is a kind of trial-and-error behavior, although it is unlikely that learning is involved: the bacterium needs a modicum of short-term memory to detect molecular concentration gradients, but it probably does not maintain long-term memories. Artificial intelligence pioneer Oliver

Selfridge called this strategy “run and twiddle,” pointing out its utility as a basic adaptive strategy: “keep going in the same way if things are getting better, and otherwise move around” (Selfridge, 1978, 1984). Similarly, one might think of a neuron “swimming” (not literally of course) in a medium composed of the complex collection of feedback loops in which it is embedded, acting to obtain one type of input signal and to avoid others. Unlike the bacterium, however, the neuron’s synaptic strengths retain information about its past trial-and-error behavior. If this view of the behavior of a neuron (or just one type of neuron) is plausible, then the closed-loop nature of how the neuron interacts with its environment is important for understanding its behavior, where the neuron’s environment consists of the rest of the animal together with the environment with which the animal as a whole interacts.

Klopf’s hedonistic neuron hypothesis extended beyond the idea that individual neurons are reinforcement learning agents. He argued that many aspects of intelligent behavior can be understood as the result of the collective behavior of a population of self-interested hedonistic neurons interacting with one another in an immense society or economic system making up an animal’s nervous system. Whether or not this view of nervous systems is useful, the collective behavior of reinforcement learning agents has implications for neuroscience. We take up this subject next.

15.10 Collective Reinforcement Learning

The behavior of populations of reinforcement learning agents is deeply relevant to the study of social and economic systems, and if anything like Klopf’s hedonistic neuron hypothesis is correct, to neuroscience as well. The hypothesis described above about how an actor–critic algorithm might be implemented in the brain only narrowly addresses the implications of the fact that the dorsal and ventral subdivisions of the striatum, the respective locations of the actor and the critic according to the hypothesis, each contain millions of medium spiny neurons whose synapses undergo change modulated by phasic bursts of dopamine neuron activity.

The actor in Figure 15.5a is a single-layer network of k actor units. The actions produced by this network are vectors $(A_1, A_2, \dots, A_k)^\top$ presumed to drive the animal’s behavior. Changes in the efficacies of the synapses of all of these units depend on the reinforcement signal δ . Because actor units attempt to make δ as large as possible, δ effectively acts as a reward signal for them (so in this case reinforcement is the same as reward). Thus, each actor unit is itself a reinforcement learning agent—a hedonistic neuron if you will. Now, to make the situation as simple as possible, assume that each of these units receives the same reward signal at the same time (although, as indicated above, the assumption that dopamine is released at all the corticostriatal synapses under the same conditions and at the same times is likely an oversimplification).

What can reinforcement learning theory tell us about what happens when all members of a population of reinforcement learning agents learn according to a common reward signal? The field of *multi-agent reinforcement learning* considers many aspects of learning by populations of reinforcement learning agents. Although this field is beyond the scope of this book, we believe that some of its basic concepts and results are relevant to thinking

about the brain's diffuse neuromodulatory systems. In multi-agent reinforcement learning (and in game theory), the scenario in which all the agents try to maximize a common reward signal that they simultaneously receive is known as a *cooperative game* or a *team problem*.

What makes a team problem interesting and challenging is that the common reward signal sent to each agent evaluates the *pattern* of activity produced by the entire population, that is, it evaluates the *collective action* of the team members. This means that any individual agent has only limited ability to affect the reward signal because any single agent contributes just one component of the collective action evaluated by the common reward signal. Effective learning in this scenario requires addressing a *structural credit assignment problem*: which team members, or groups of team members, deserve credit for a favorable reward signal, or blame for an unfavorable reward signal? It is a *cooperative game*, or a *team problem*, because the agents are united in seeking to increase the same reward signal: there are no conflicts of interest among the agents. The scenario would be a *competitive game* if different agents receive different reward signals, where each reward signal again evaluates the collective action of the population, and the objective of each agent is to increase its own reward signal. In this case there might be conflicts of interest among the agents, meaning that actions that are good for some agents are bad for others. Even deciding what the best collective action should be is a non-trivial aspect of game theory. This competitive setting might be relevant to neuroscience too (for example, to account for heterogeneity of dopamine neuron activity), but here we focus only on the cooperative, or team, case.

How can each reinforcement learning agent in a team learn to “do the right thing” so that the collective action of the team is highly rewarded? An interesting result is that if each agent can learn effectively despite its reward signal being corrupted by a large amount of noise, and despite its lack of access to complete state information, then the population as a whole will learn to produce collective actions that improve as evaluated by the common reward signal, even when the agents cannot communicate with one another. Each agent faces its own reinforcement learning task in which its influence on the reward signal is deeply buried in the noise created by the influences of other agents. In fact, for any agent, all the other agents are part of its environment because its input, both the part conveying state information and the reward part, depends on how all the other agents are behaving. Furthermore, lacking access to the actions of the other agents, indeed lacking access to the parameters determining their policies, each agent can only partially observe the state of its environment. This makes each team member’s learning task very difficult, but if each uses a reinforcement learning algorithm able to increase a reward signal even under these difficult conditions, teams of reinforcement learning agents can learn to produce collective actions that improve over time as evaluated by the team’s common reward signal.

If the team members are neuron-like units, then each unit has to have the goal of increasing the amount of reward it receives over time, as the actor unit does that we described in Section 15.8. Each unit’s learning algorithm has to have two essential features. First, it has to use contingent eligibility traces. Recall that a contingent eligibility trace, in neural terms, is initiated (or increased) at a synapse when its presynaptic input

participates in causing the postsynaptic neuron to fire. A non-contingent eligibility trace, in contrast, is initiated or increased by presynaptic input independently of what the postsynaptic neuron does. As explained in Section 15.8, by keeping information about what actions were taken in what states, contingent eligibility traces allow credit for reward, or blame for punishment, to be apportioned to an agent's policy parameters according to the contribution the values of these parameters made in determining the agent's action. By similar reasoning, a team member must remember its recent action so that it can either increase or decrease the likelihood of producing that action according to the reward signal that is subsequently received. The action component of a contingent eligibility trace implements this action memory. Because of the complexity of the learning task, however, contingent eligibility is merely a preliminary step in the credit assignment process: the relationship between a single team member's action and changes in the team's reward signal is a statistical correlation that has to be estimated over many trials. Contingent eligibility is an essential but preliminary step in this process.

Learning with non-contingent eligibility traces does not work at all in the team setting because it does not provide a way to correlate actions with consequent changes in the reward signal. Non-contingent eligibility traces are adequate for learning to predict, as the critic component of the actor–critic algorithm does, but they do not support learning to control, as the actor component must do. The members of a population of critic-like agents may still receive a common reinforcement signal, but they would all learn to predict the same quantity (which in the case of an actor–critic method, would be the expected return for the current policy). How successful each member of the population would be in learning to predict the expected return would depend on the information it receives, which could be very different for different members of the population. There would be no need for the population to produce differentiated patterns of activity. This is not a team problem as defined here.

A second requirement for collective learning in a team problem is that there has to be variability in the actions of the team members in order for the team to explore the space of collective actions. The simplest way for a team of reinforcement learning agents to do this is for each member to independently explore its own action space through persistent variability in its output. This will cause the team as a whole to vary its collective actions. For example, a team of the actor units described in Section 15.8 explores the space of collective actions because the output of each unit, being a Bernoulli-logistic unit, probabilistically depends on the weighted sum of its input vector's components. The weighted sum biases firing probability up or down, but there is always variability. Because each unit uses a REINFORCE policy gradient algorithm (Chapter 13), each unit adjusts its weights with the goal of maximizing the average reward rate it experiences while stochastically exploring its own action space. One can show, as Williams (1992) did, that a team of Bernoulli-logistic REINFORCE units implements a policy gradient algorithm as a *whole* with respect to average rate of the team's common reward signal, where the actions are the collective actions of the team.

Further, Williams (1992) showed that a team of Bernoulli-logistic units using REINFORCE ascends the average reward gradient when the units in the team are interconnected to form a multilayer ANN. In this case, the reward signal is broadcast to all the units in

the network, though reward may depend only on the collective actions of the network's output units. This means that a multilayer team of Bernoulli-logistic REINFORCE units learns like a multilayer network trained by the widely-used error backpropagation method, but in this case the backpropagation process is replaced by the broadcasted reward signal. In practice, the error backpropagation method is considerably faster, but the reinforcement learning team method is more plausible as a neural mechanism, especially in light of what is being learned about reward-modulated STDP as discussed in Section 15.8.

Exploration through independent exploration by team members is only the simplest way for a team to explore; more sophisticated methods are possible if the team members coordinate their actions to focus on particular parts of the collective action space, either by communicating with one another or by responding to common inputs. There are also mechanisms more sophisticated than contingent eligibility traces for addressing structural credit assignment, which is easier in a team problem when the set of possible collective actions is restricted in some way. An extreme case is a winner-take-all arrangement (for example, the result of lateral inhibition in the brain) that restricts collective actions to those to which only one, or a few, team members contribute. In this case the winners get the credit or blame for resulting reward or punishment.

Details of learning in cooperative games (or team problems) and non-cooperative game problems are beyond the scope of this book. The Bibliographical and Historical Remarks section at the end of this chapter cites a selection of the relevant publications, including extensive references to research on implications for neuroscience of collective reinforcement learning.

15.11 Model-based Methods in the Brain

Reinforcement learning's distinction between model-free and model-based algorithms is proving to be useful for thinking about animal learning and decision processes. Section 14.6 discusses how this distinction aligns with that between habitual and goal-directed animal behavior. The hypothesis discussed above about how the brain might implement an actor-critic algorithm is relevant only to an animal's habitual mode of behavior because the basic actor-critic method is model-free. What neural mechanisms are responsible for producing goal-directed behavior, and how do they interact with those underlying habitual behavior?

One way to investigate questions about the brain structures involved in these modes of behavior is to inactivate an area of a rat's brain and then observe what the rat does in an outcome-devaluation experiment (Section 14.6). Results from experiments like these indicate that the actor-critic hypothesis described above is too simple in placing the actor in the dorsal striatum. Inactivating one part of the dorsal striatum, the dorsolateral striatum (DLS), impairs habit learning, causing the animal to rely more on goal-directed processes. On the other hand, inactivating the dorsomedial striatum (DMS) impairs goal-directed processes, requiring the animal to rely more on habit learning. Results like these support the view that the DLS in rodents is more involved in model-free processes, whereas their DMS is more involved in model-based processes. Results of

studies with human subjects in similar experiments using functional neuroimaging, and with non-human primates, support the view that the analogous structures in the primate brain are differentially involved in habitual and goal-directed modes of behavior.

Other studies identify activity associated with model-based processes in the prefrontal cortex of the human brain, the front-most part of the frontal cortex implicated in executive function, including planning and decision making. Specifically implicated is the orbitofrontal cortex (OFC), the part of the prefrontal cortex immediately above the eyes. Functional neuroimaging in humans, and also recordings of the activities of single neurons in monkeys, reveals strong activity in the OFC related to the subjective reward value of biologically significant stimuli, as well as activity related to the reward expected as a consequence of actions. Although not free of controversy, these results suggest significant involvement of the OFC in goal-directed choice. It may be critical for the reward part of an animal's environment model.

Another structure involved in model-based behavior is the hippocampus, a structure critical for memory and spatial navigation. A rat's hippocampus plays a critical role in the rat's ability to navigate a maze in the goal-directed manner that led Tolman to the idea that animals use models, or cognitive maps, in selecting actions (Section 14.5). The hippocampus may also be a critical component of our human ability to imagine new experiences (Hassabis and Maguire, 2007; Ólafsdóttir, Barry, Saleem, Hassabis, and Spiers, 2015).

The findings that most directly implicate the hippocampus in planning—the process needed to enlist an environment model in making decisions—come from experiments that decode the activity of neurons in the hippocampus to determine what part of space hippocampal activity is representing on a moment-to-moment basis. When a rat pauses at a choice point in a maze, the representation of space in the hippocampus sweeps forward (and not backwards) along the possible paths the animal can take from that point (Johnson and Redish, 2007). Furthermore, the spatial trajectories represented by these sweeps closely correspond to the rat's subsequent navigational behavior (Pfeiffer and Foster, 2013). These results suggest that the hippocampus is critical for the state-transition part of an animal's environment model, and that it is part of a system that uses the model to simulate possible future state sequences to assess the consequences of possible courses of action: a form of planning.

The results described above add to a voluminous literature on neural mechanisms underlying goal-directed, or model-based, learning and decision making, but many questions remain unanswered. For example, how can areas as structurally similar as the DLS and DMS be essential components of modes of learning and behavior that are as different as model-free and model-based algorithms? Are separate structures responsible for (what we call) the transition and reward components of an environment model? Is all planning conducted at decision time via simulations of possible future courses of action as the forward sweeping activity in the hippocampus suggests? In other words, is all planning something like a rollout algorithm (Section 8.10)? Or are models sometimes engaged in the background to refine or recompute value information as illustrated by the Dyna architecture (Section 8.2)? How does the brain arbitrate between the use of the habit and goal-directed systems? Is there, in fact, a clear separation between the neural substrates of these systems?

The evidence is not pointing to a positive answer to this last question. Summarizing the situation, Doll, Simon, and Daw (2012) wrote that “model-based influences appear ubiquitous more or less wherever the brain processes reward information,” and this is true even in the regions thought to be critical for model-free learning. This includes the dopamine signals themselves, which can exhibit the influence of model-based information in addition to the reward prediction errors thought to be the basis of model-free processes.

Continuing neuroscience research informed by reinforcement learning’s model-free and model-based distinction has the potential to sharpen our understanding of habitual and goal-directed processes in the brain. A better grasp of these neural mechanisms may lead to algorithms combining model-free and model-based methods in ways that have not yet been explored in computational reinforcement learning.

15.12 Addiction

Understanding the neural basis of drug abuse is a high-priority goal of neuroscience with the potential to produce new treatments for this serious public health problem. One view is that drug craving is the result of the same motivation and learning processes that lead us to seek natural rewarding experiences that serve our biological needs. Addictive substances, by being intensely reinforcing, effectively co-opt our natural mechanisms of learning and decision making. This is plausible given that many—though not all—drugs of abuse increase levels of dopamine either directly or indirectly in regions around terminals of dopamine neuron axons in the striatum, a brain structure firmly implicated in normal reward-based learning (Section 15.7). But the self-destructive behavior associated with drug addiction is not characteristic of normal learning. What is different about dopamine-mediated learning when the reward is the result of an addictive drug? Is addiction the result of normal learning in response to substances that were largely unavailable throughout our evolutionary history, so that evolution could not select against their damaging effects? Or do addictive substances somehow interfere with normal dopamine-mediated learning?

The reward prediction error hypothesis of dopamine neuron activity and its connection to TD learning are the basis of a model due to Redish (2004) of some—but certainly not all—features of addiction. The model is based on the observation that administration of cocaine and some other addictive drugs produces a transient increase in dopamine. In the model, this dopamine surge is assumed to increase the TD error, δ , in a way that cannot be cancelled out by changes in the value function. In other words, whereas δ is reduced to the degree that a normal reward is predicted by antecedent events (Section 15.6), the contribution to δ due to an addictive stimulus does not decrease as the reward signal becomes predicted: drug rewards cannot be “predicted away.” The model does this by preventing δ from ever becoming negative when the reward signal is due to an addictive drug, thus eliminating the error-correcting feature of TD learning for states associated with administration of the drug. The result is that the values of these states increase without bound, making actions leading to these states preferred above all others.

Addictive behavior is much more complicated than this result from Redish's model, but the model's main idea may be a piece of the puzzle. Or the model might be misleading. Dopamine appears not to play a critical role in all forms of addiction, and not everyone is equally susceptible to developing addictive behavior. Moreover, the model does not include the changes in many circuits and brain regions that accompany chronic drug taking, for example, changes that lead to a drug's diminishing effect with repeated use. It is also likely that addiction involves model-based processes. Still, Redish's model illustrates how reinforcement learning theory can be enlisted in the effort to understand a major health problem. In a similar manner, reinforcement learning theory has been influential in the development of the new field of computational psychiatry, which aims to improve understanding of mental disorders through mathematical and computational methods.

15.13 Summary

The neural pathways involved in the brain's reward system are complex and incompletely understood, but neuroscience research directed toward understanding these pathways and their roles in behavior is progressing rapidly. This research is revealing striking correspondences between the brain's reward system and the theory of reinforcement learning as presented in this book.

The *reward prediction error hypothesis of dopamine neuron activity* was proposed by scientists who recognized striking parallels between the behavior of TD errors and the activity of neurons that produce dopamine, a neurotransmitter essential in mammals for reward-related learning and behavior. Experiments conducted in the late 1980s and 1990s in the laboratory of neuroscientist Wolfram Schultz showed that dopamine neurons respond to rewarding events with substantial bursts of activity, called phasic responses, only if the animal does not expect those events, suggesting that dopamine neurons are signaling reward prediction errors instead of reward itself. Further, these experiments showed that as an animal learns to predict a rewarding event on the basis of preceding sensory cues, the phasic activity of dopamine neurons shifts to earlier predictive cues while decreasing to later predictive cues. This parallels the backing-up effect of the TD error as a reinforcement learning agent learns to predict reward.

Other experimental results firmly establish that the phasic activity of dopamine neurons is a reinforcement signal for learning that reaches multiple areas of the brain by means of profusely branching axons of dopamine producing neurons. These results are consistent with the distinction we make between a reward signal, R_t , and a reinforcement signal, which is the TD error δ_t in most of the algorithms we present. Phasic responses of dopamine neurons are reinforcement signals, not reward signals.

A prominent hypothesis is that the brain implements something like an actor–critic algorithm. Two structures in the brain (the dorsal and ventral subdivisions of the striatum), both of which play critical roles in reward-based learning, may function respectively like an actor and a critic. That the TD error is the reinforcement signal for both the actor and the critic fits well with the facts that dopamine neuron axons target both the dorsal and ventral subdivisions of the striatum; that dopamine appears to be

critical for modulating synaptic plasticity in both structures; and that the effect on a target structure of a neuromodulator such as dopamine depends on properties of the target structure and not just on properties of the neuromodulator.

The actor and the critic can be implemented by ANNs consisting of neuron-like units having learning rules based on the policy-gradient actor–critic method described in Section 13.5. Each connection in these networks is like a synapse between neurons in the brain, and the learning rules correspond to rules governing how synaptic efficacies change as functions of the activities of the presynaptic and the postsynaptic neurons, together with neuromodulatory input corresponding to input from dopamine neurons. In this setting, each synapse has its own eligibility trace that records past activity involving that synapse. The only difference between the actor and critic learning rules is that they use different kinds of eligibility traces: the critic unit’s traces are *non-contingent* because they do not involve the critic unit’s output, whereas the actor unit’s traces are *contingent* because in addition to the actor unit’s input, they depend on the actor unit’s output. In the hypothetical implementation of an actor–critic system in the brain, these learning rules respectively correspond to rules governing plasticity of corticostriatal synapses that convey signals from the cortex to the principal neurons in the dorsal and ventral striatal subdivisions, synapses that also receive inputs from dopamine neurons.

The learning rule of an actor unit in the actor–critic network closely corresponds to *reward-modulated spike-timing-dependent plasticity*. In spike-timing-dependent plasticity (STDP), the relative timing of pre- and postsynaptic activity determines the direction of synaptic change. In reward-modulated STDP, changes in synapses in addition depend on a neuromodulator, such as dopamine, arriving within a time window that can last up to 10 seconds after the conditions for STDP are met. Evidence is accumulating that reward-modulated STDP occurs at corticostriatal synapses, where the actor’s learning takes place in the hypothetical neural implementation of an actor–critic system, adds to the plausibility of the hypothesis that something like an actor–critic system exists in the brains of some animals.

The idea of synaptic eligibility and basic features of the actor learning rule derive from Klopf’s hypothesis of the “hedonistic neuron” (Klopf, 1972, 1981). He conjectured that individual neurons seek to obtain reward and to avoid punishment by adjusting the efficacies of their synapses on the basis of rewarding or punishing consequences of their action potentials. A neuron’s activity can affect its later input because the neuron is embedded in many feedback loops, some within the animal’s nervous system and body and others passing through the animal’s external environment. Klopf’s idea of eligibility is that synapses are temporarily marked as eligible for modification if they participated in the neuron’s firing (making this the contingent form of eligibility trace). A synapse’s efficacy is modified if a reinforcing signal arrives while the synapse is eligible. We alluded to the chemotactic behavior of a bacterium as an example of a single cell that directs its movements in order to seek some molecules and to avoid others.

A conspicuous feature of the dopamine system is that fibers releasing dopamine project widely to multiple parts of the brain. Although it is likely that only some populations of dopamine neurons broadcast the same reinforcement signal, if this signal reaches the synapses of many neurons involved in actor-type learning, then the situation can

be modeled as a *team problem*. In this type of problem, each agent in a collection of reinforcement learning agents receives the same reinforcement signal, where that signal depends on the activities of all members of the collection, or team. If each team member uses a sufficiently capable learning algorithm, the team can learn collectively to improve performance of the entire team as evaluated by the globally-broadcast reinforcement signal, even if the team members do not directly communicate with one another. This is consistent with the wide dispersion of dopamine signals in the brain and provides a neurally plausible alternative to the widely-used error-backpropagation method for training multilayer networks.

The distinction between model-free and model-based reinforcement learning is helping neuroscientists investigate the neural bases of habitual and goal-directed learning and decision making. Research so far points to their being some brain regions more involved in one type of process than the other, but the picture remains unclear because model-free and model-based processes do not appear to be neatly separated in the brain. Many questions remain unanswered. Perhaps most intriguing is evidence that the hippocampus, a structure traditionally associated with spatial navigation and memory, appears to be involved in simulating possible future courses of action as part of an animal's decision-making process. This suggests that it is part of a system that uses an environment model for planning.

Reinforcement learning theory is also influencing thinking about neural processes underlying drug abuse. A model of some features of drug addiction is based on the reward prediction error hypothesis. It proposes that an addicting stimulant, such as cocaine, destabilizes TD learning to produce unbounded growth in the values of actions associated with drug intake. This is far from a complete model of addiction, but it illustrates how a computational perspective suggests theories that can be tested with further research. The new field of computational psychiatry similarly focuses on the use of computational models, some derived from reinforcement learning, to better understand mental disorders.

This chapter only touched the surface of how the neuroscience of reinforcement learning and the development of reinforcement learning in computer science and engineering have influenced one another. Most features of reinforcement learning algorithms owe their design to purely computational considerations, but some have been influenced by hypotheses about neural learning mechanisms. Remarkably, as experimental data has accumulated about the brain's reward processes, many of the purely computationally-motivated features of reinforcement learning algorithms are turning out to be consistent with neuroscience data. Other features of computational reinforcement learning, such as eligibility traces and the ability of teams of reinforcement learning agents to learn to act collectively under the influence of a globally-broadcast reinforcement signal, may also turn out to parallel experimental data as neuroscientists continue to unravel the neural basis of reward-based animal learning and behavior.

Bibliographical and Historical Remarks

The number of publications treating parallels between the neuroscience of learning and decision making and the approach to reinforcement learning presented in this book is enormous. We can cite only a small selection. Niv (2009), Dayan and Niv (2008), Glimcher (2011), Ludvig, Bellemare, and Pearson (2011), and Shah (2012) are good places to start.

Together with economics, evolutionary biology, and mathematical psychology, reinforcement learning theory is helping to formulate quantitative models of the neural mechanisms of choice in humans and non-human primates. With its focus on learning, this chapter only lightly touches upon the neuroscience of decision making. Glimcher (2003) introduced the field of “neuroeconomics,” in which reinforcement learning contributes to the study of the neural basis of decision making from an economics perspective. See also Glimcher and Fehr (2013). The text on computational and mathematical modeling in neuroscience by Dayan and Abbott (2001) includes reinforcement learning’s role in these approaches. Sterling and Laughlin (2015) examined the neural basis of learning in terms of general design principles that enable efficient adaptive behavior.

- 15.1** There are many good expositions of basic neuroscience. Kandel, Schwartz, Jessell, Siegelbaum, and Hudspeth (2013) is an authoritative and very comprehensive source.
- 15.2** Berridge and Kringelbach (2008) reviewed the neural basis of reward and pleasure, pointing out that reward processing has many dimensions and involves many neural systems. Space prevents discussion of the influential research of Berridge and Robinson (1998), who distinguish between the hedonic impact of a stimulus, which they call “liking,” and the motivational effect, which they call “wanting.” Hare, O’Doherty, Camerer, Schultz, and Rangel (2008) examined the neural basis of value-related signals from an economic perspective, distinguishing between goal values, decision values, and prediction errors. Decision value is goal value minus action cost. See also Rangel, Camerer, and Montague (2008), Rangel and Hare (2010), and Peters and Büchel (2010).
- 15.3** The reward prediction error hypothesis of dopamine neuron activity is most prominently discussed by Schultz, Dayan, and Montague (1997). The hypothesis was first explicitly put forward by Montague, Dayan, and Sejnowski (1996). As they stated the hypothesis, it referred to reward prediction errors (RPEs) but not specifically to TD errors; however, their development of the hypothesis made it clear that they were referring to TD errors. The earliest recognition of the TD-error/dopamine connection of which we are aware is that of Montague, Dayan, Nowlan, Pouget, and Sejnowski (1993), who proposed a TD-error-modulated Hebbian learning rule motivated by results on dopamine signaling from Schultz’s group. The connection was also pointed out in an abstract by Quartz, Dayan, Montague, and Sejnowski (1992). Montague and Sejnowski (1994) emphasized the importance of prediction in the brain and outlined how predictive Hebbian learning modulated by TD errors could be implemented via a diffuse neuromodulatory system, such as the dopamine system. Friston, Tononi, Reike, Sporns,

and Edelman (1994) presented a model of value-dependent learning in the brain in which synaptic changes are mediated by a TD-like error provided by a global neuromodulatory signal (although they did not single out dopamine). Montague, Dayan, Person, and Sejnowski (1995) presented a model of honeybee foraging using the TD error. The model is based on research by Hammer, Menzel, and colleagues (Hammer and Menzel, 1995; Hammer, 1997) showing that the neuromodulator octopamine acts as a reinforcement signal in the honeybee. Montague et al. (1995) pointed out that dopamine likely plays a similar role in the vertebrate brain. Barto (1995a) related the actor–critic architecture to basal-ganglionic circuits and discussed the relationship between TD learning and the main results from Schultz’s group. Houk, Adams, and Barto (1995) suggested how TD learning and the actor–critic architecture might map onto the anatomy, physiology, and molecular mechanism of the basal ganglia. Doya and Sejnowski (1998) extended their earlier paper on a model of birdsong learning (Doya and Sejnowski, 1995) by including a TD-like error identified with dopamine to reinforce the selection of auditory input to be memorized. O’Reilly and Frank (2006) and O’Reilly, Frank, Hazy, and Watz (2007) argued that phasic dopamine signals are RPEs but not TD errors. In support of their theory they cited results with variable interstimulus intervals that do not match predictions of a simple TD model, as well as the observation that higher-order conditioning beyond second-order conditioning is rarely observed, while TD learning is not so limited. Dayan and Niv (2008) discussed “the good, the bad, and the ugly” of how reinforcement learning theory and the reward prediction error hypothesis align with experimental data. Glimcher (2011) reviewed the empirical findings that support the reward prediction error hypothesis and emphasized the significance of the hypothesis for contemporary neuroscience.

- 15.4** Graybiel (2000) is a brief primer on the basal ganglia. The experiments mentioned that involve optogenetic activation of dopamine neurons were conducted by Tsai, Zhang, Adamantidis, Stuber, Bonci, de Lecea, and Deisseroth (2009), Steinberg, Keiflin, Boivin, Witten, Deisseroth, and Janak (2013), and Claridge-Chang, Roorda, Vrontou, Sjulson, Li, Hirsh, and Miesenböck (2009). Fiorillo, Yun, and Song (2013), Lammel, Lim, and Malenka (2014), and Saddoris, Cacciapaglia, Wightman, and Carelli (2015) are among studies showing that the signaling properties of dopamine neurons are specialized for different target regions. RPE-signaling neurons may belong to one among multiple populations of dopamine neurons having different targets and subserving different functions. Eshel, Tian, Bukwicz, and Uchida (2016) found homogeneity of reward prediction error responses of dopamine neurons in the lateral VTA during classical conditioning in mice, though their results do not rule out response diversity across wider areas. Gershman, Pesaran, and Daw (2009) studied reinforcement learning tasks that can be decomposed into independent components with separate reward signals, finding evidence in human neuroimaging data suggesting that the brain exploits this kind of structure.

- 15.5** Schultz's 1998 survey article is a good entrée into the very extensive literature on reward predicting signaling of dopamine neurons. Berns, McClure, Pagnoni, and Montague (2001), Breiter, Aharon, Kahneman, Dale, and Shizgal (2001), Pagnoni, Zink, Montague, and Berns (2002), and O'Doherty, Dayan, Friston, Critchley, and Dolan (2003) described functional brain imaging studies supporting the existence of signals like TD errors in the human brain.
- 15.6** This section roughly follows Barto (1995a) in explaining how TD errors mimic the main results from Schultz's group on the phasic responses of dopamine neurons.
- 15.7** This section is largely based on Takahashi, Schoenbaum, and Niv (2008) and Niv (2009). To the best of our knowledge, Barto (1995a) and Houk, Adams, and Barto (1995) first speculated about possible implementations of actor–critic algorithms in the basal ganglia. On the basis of functional magnetic resonance imaging of human subjects while engaged in instrumental conditioning, O'Doherty, Dayan, Schultz, Deichmann, Friston, and Dolan (2004) suggested that the actor and the critic are most likely located respectively in the dorsal and ventral striatum. Gershman, Moustafa, and Ludvig (2014) focused on how time is represented in reinforcement learning models of the basal ganglia, discussing evidence for, and implications of, various computational approaches to time representation.
The hypothetical neural implementation of the actor–critic architecture described in this section includes very little detail about known basal ganglia anatomy and physiology. In addition to the more detailed hypothesis of Houk, Adams, and Barto (1995), a number of other hypotheses include more specific connections to anatomy and physiology and are claimed to explain additional data. These include hypotheses proposed by Suri and Schultz (1998, 1999), Brown, Bullock, and Grossberg (1999), Contreras-Vidal and Schultz (1999), Suri, Bargas, and Arbib (2001), O'Reilly and Frank (2006), and O'Reilly, Frank, Hazy, and Watz (2007). Joel, Niv, and Ruppin (2002) critically evaluated the anatomical plausibility of several of these models and present an alternative intended to accommodate some neglected features of basal ganglionic circuitry.
- 15.8** The actor learning rule discussed here is more complicated than the one in the early actor–critic network of Barto et al. (1983). Actor-unit eligibility traces in that network were traces of just $A_t \times \mathbf{x}(S_t)$ instead of the full $(A_t - \pi(1|S_t, \theta))\mathbf{x}(S_t)$. That work did not benefit from the policy-gradient theory presented in Chapter 13 or the contributions of Williams (1986, 1992), who showed how an ANN of Bernoulli-logistic units could implement a policy-gradient method.
Reynolds and Wickens (2002) proposed a three-factor rule for synaptic plasticity in the corticostriatal pathway in which dopamine modulates changes in corticostriatal synaptic efficacy. They discussed the experimental support for this kind of learning rule and its possible molecular basis. The definitive demonstration of spike-timing-dependent plasticity (STDP) is attributed to Markram, Lübke, Frotscher, and Sakmann (1997), with evidence from earlier experiments

by Levy and Steward (1983) and others that the relative timing of pre- and postsynaptic spikes is critical for inducing changes in synaptic efficacy. Rao and Sejnowski (2001) suggested how STDP could be the result of a TD-like mechanism at synapses with non-contingent eligibility traces lasting about 10 milliseconds. Dayan (2002) commented that this would require an error as in Sutton and Barto's (1981a) early model of classical conditioning and not a true TD error. Representative publications from the extensive literature on reward-modulated STDP are Wickens (1990), Reynolds and Wickens (2002), and Calabresi, Picconi, Tozzi and Di Filippo (2007). Pawlak and Kerr (2008) showed that dopamine is necessary to induce STDP at the corticostriatal synapses of medium spiny neurons. See also Pawlak, Wickens, Kirkwood, and Kerr (2010). Yagishita, Hayashi-Takagi, Ellis-Davies, Urakubo, Ishii, and Kasai (2014) found that dopamine promotes spine enlargement of the medium spiny neurons of mice only during a time window of from 0.3 to 2 seconds after STDP stimulation. Izhikevich (2007) proposed and explored the idea of using STDP timing conditions to trigger contingent eligibility traces. Frémaux, Sprekeler, and Gerstner (2010) proposed theoretical conditions for successful learning by rules based on reward-modulated STDP.

15.9

Klopf's hedonistic neuron hypothesis (Klopf 1972, 1982) inspired our actor–critic algorithm implemented as an ANN with a single neuron-like unit, called the actor unit, implementing a Law-of-Effect-like learning rule (Barto, Sutton, and Anderson, 1983). Ideas related to Klopf's synaptically-local eligibility have been proposed by others. Crow (1968) proposed that changes in the synapses of cortical neurons are sensitive to the consequences of neural activity. Emphasizing the need to address the time delay between neural activity and its consequences in a reward-modulated form of synaptic plasticity, he proposed a contingent form of eligibility, but associated with entire neurons instead of individual synapses. According to his hypothesis, a wave of neuronal activity

leads to a short-term change in the cells involved in the wave such that they are picked out from a background of cells not so activated. ... such cells are rendered sensitive by the short-term change to a reward signal ... in such a way that if such a signal occurs before the end of the decay time of the change the synaptic connexions between the cells are made more effective.
(Crow, 1968)

Crow argued against previous proposals that reverberating neural circuits play this role by pointing out that the effect of a reward signal on such a circuit would "...establish the synaptic connexions leading to the reverberation (that is to say, those involved in activity at the time of the reward signal) and not those on the path which led to the adaptive motor output." Crow further postulated that reward signals are delivered via a "distinct neural fiber system," presumably the one into which Olds and Milner (1954) tapped, that would transform synaptic connections "from a short into a long-term form."

In another farsighted hypothesis, Miller proposed a Law-of-Effect-like learning rule that includes synaptically-local contingent eligibility traces:

... it is envisaged that in a particular sensory situation neurone B, by chance, fires a ‘meaningful burst’ of activity, which is then translated into motor acts, which then change the situation. It must be supposed that the meaningful burst has an influence, *at the neuronal level*, on all of its own synapses which are active at the time ... thereby making a preliminary selection of the synapses to be strengthened, though not yet actually strengthening them. ...The strengthening signal ... makes the final selection ... and accomplishes the definitive change in the appropriate synapses. (Miller, 1981, p. 81)

Miller’s hypothesis also included a critic-like mechanism, which he called a “sensory analyzer unit,” that worked according to classical conditioning principles to provide reinforcement signals to neurons so that they would learn to move from lower- to higher-valued states, thus anticipating the use of the TD error as a reinforcement signal in the actor–critic architecture. Miller’s idea not only parallels Klopff’s (with the exception of its explicit invocation of a distinct “strengthening signal”), it also anticipated the general features of reward-modulated STDP.

A related though different idea, which Seung (2003) called the “hedonistic synapse,” is that synapses individually adjust the probability that they release neurotransmitter in the manner of the Law of Effect: if reward follows release, the release probability increases, and decreases if reward follows failure to release. This is essentially the same as the learning scheme Minsky used in his 1954 Princeton PhD dissertation, where he called the synapse-like learning element a SNARC (Stochastic Neural-Analog Reinforcement Calculator). Contingent eligibility is involved in these ideas too, although it is contingent on the activity of an individual synapse instead of the postsynaptic neuron. Also related is the proposal of Unnikrishnan and Venugopal (1994) that uses the correlation-based method of Harth and Tzanakou (1974) to adjust ANN weights.

Frey and Morris (1997) proposed the idea of a “synaptic tag” for the induction of long-lasting strengthening of synaptic efficacy. Though not unlike Klopff’s eligibility, their tag was hypothesized to consist of a temporary strengthening of a synapse that could be transformed into a long-lasting strengthening by subsequent neuron activation. The model of O’Reilly and Frank (2006) and O’Reilly, Frank, Hazy, and Watz (2007) uses working memory to bridge temporal intervals instead of eligibility traces. Wickens and Kotter (1995) discuss possible mechanisms for synaptic eligibility. He, Huertas, Hong, Tie, Hell, Shouval, Kirkwood (2015) provide evidence supporting the existence of contingent eligibility traces in synapses of cortical neurons with time courses like those of the eligibility traces Klopff postulated.

The metaphor of a neuron using a learning rule related to bacterial chemotaxis was discussed by Barto (1989). Koshland’s extensive study of bacterial chemotaxis was in part motivated by similarities between features of bacteria and features of neurons (Koshland, 1980). See also Berg (1975). Shimansky (2009) proposed a

synaptic learning rule somewhat similar to Seung’s mentioned above in which each synapse individually acts like a chemotactic bacterium. In this case a collection of synapses “swims” toward attractants in the high-dimensional space of synaptic weight values. Montague, Dayan, Person, and Sejnowski (1995) proposed a chemotactic-like model of the bee’s foraging behavior involving the neuromodulator octopamine.

- 15.10** Research on the behavior of reinforcement learning agents in team and game problems has a long history roughly occurring in three phases. To the best of our knowledge, the first phase began with investigations by the Russian mathematician and physicist M. L. Tsetlin. A collection of his work was published as Tsetlin (1973) after his death in 1966. Our Sections 1.7 and 4.8 refer to his study of learning automata in connection to bandit problems. The Tsetlin collection also includes studies of learning automata in team and game problems, which led to later work in this area using stochastic learning automata as described by Narendra and Thathachar (1974, 1989), Viswanathan and Narendra (1974), Lakshmivarahan and Narendra (1982), Narendra and Wheeler (1983), and Thathachar and Sastry (2002). Thathachar and Sastry (2011) is a more recent comprehensive account. These studies were mostly restricted to non-associative learning automata, meaning that they did not address associative, or contextual, bandit problems (Section 2.9).

The second phase began with the extension of learning automata to the associative, or contextual, case. Barto, Sutton, and Brouwer (1981) and Barto and Sutton (1981b) experimented with associative stochastic learning automata in single-layer ANNs to which a global reinforcement signal was broadcast. The learning algorithm was an associative extension of the Alopex algorithm of Harth and Tzanakou (1974). Barto et al. called neuron-like elements implementing this kind of learning *associative search elements* (ASEs). Barto and Anandan (1985) introduced an associative reinforcement learning algorithm called the *associative reward-penalty* (A_{R-P}) algorithm. They proved a convergence result by combining theory of stochastic learning automata with theory of pattern classification. Barto (1985, 1986) and Barto and Jordan (1987) described results with teams of A_{R-P} units connected into multi-layer ANNs, showing that they could learn nonlinear functions, such as XOR and others, with a globally-broadcast reinforcement signal. Barto (1985) extensively discussed this approach to ANNs and how this type of learning rule is related to others in the literature at that time. Williams (1992) mathematically analyzed and broadened this class of learning rules and related their use to the error backpropagation method for training multilayer ANNs. Williams (1988) described several ways that backpropagation and reinforcement learning can be combined for training ANNs. Williams (1992) showed that a special case of the A_{R-P} algorithm is a REINFORCE algorithm, although better results were obtained with the general A_{R-P} algorithm (Barto, 1985).

The third phase of interest in teams of reinforcement learning agents was influenced by increased understanding of the role of dopamine as a widely broadcast neuromodulator and speculation about the existence of reward-modulated STDP. Much more so than earlier research, this research considers details of synaptic plasticity and other constraints from neuroscience. Publications include the following (chronologically and alphabetically): Bartlett and Baxter (1999, 2000), Xie and Seung (2004), Baras and Meir (2007), Farries and Fairhall (2007), Florian (2007), Izhikevich (2007), Pecevski, Maass, and Legenstein (2008), Legenstein, Pecevski, and Maass (2008), Kolodziejksi, Porr, and Wörgötter (2009), Urbanczik and Senn (2009), and Vasilaki, Frémaux, Urbanczik, Senn, and Gerstner (2009). Nowé, Vrancx, and De Hauwere (2012) reviewed more recent developments in the wider field of multi-agent reinforcement learning

- 15.11** Yin and Knowlton (2006) reviewed findings from outcome-devaluation experiments with rodents supporting the view that habitual and goal-directed behavior (as psychologists use the phrase) are respectively most associated with processing in the dorsolateral striatum (DLS) and the dorsomedial striatum (DMS). Results of functional imaging experiments with human subjects in the outcome-devaluation setting by Valentini, Dickinson, and O'Doherty (2007) suggest that the orbitofrontal cortex (OFC) is an important component of goal-directed choice. Single unit recordings in monkeys by Padoa-Schioppa and Assad (2006) support the role of the OFC in encoding values guiding choice behavior. Rangel, Camerer, and Montague (2008) and Rangel and Hare (2010) reviewed findings from the perspective of neuroeconomics about how the brain makes goal-directed decisions. Pezzulo, van der Meer, Lansink, and Pennartz (2014) reviewed the neuroscience of internally generated sequences and presented a model of how these mechanisms might be components of model-based planning. Daw and Shohamy (2008) proposed that while dopamine signaling connects well to habitual, or model-free, behavior, other processes are involved in goal-directed, or model-based, behavior. Data from experiments by Bromberg-Martin, Matsumoto, Hong, and Hikosaka (2010) indicate that dopamine signals contain information pertinent to both habitual and goal-directed behavior. Doll, Simon, and Daw (2012) argued that there may not a clear separation in the brain between mechanisms that subserve habitual and goal-directed learning and choice.
- 15.12** Keiflin and Janak (2015) reviewed connections between TD errors and addiction. Nutt, Lingford-Hughes, Erritzoe, and Stokes (2015) critically evaluated the hypothesis that addiction is due to a disorder of the dopamine system. Montague, Dolan, Friston, and Dayan (2012) outlined the goals and early efforts in the field of computational psychiatry, and Adams, Huys, and Roiser (2015) reviewed more recent progress.

Chapter 16

Applications and Case Studies

In this chapter we present a few case studies of reinforcement learning. Several of these are substantial applications of potential economic significance. One, Samuel’s checkers player, is primarily of historical interest. Our presentations are intended to illustrate some of the trade-offs and issues that arise in real applications. For example, we emphasize how domain knowledge is incorporated into the formulation and solution of the problem. We also highlight the representation issues that are so often critical to successful applications. The algorithms used in some of these case studies are substantially more complex than those we have presented in the rest of the book. Applications of reinforcement learning are still far from routine and typically require as much art as science. Making applications easier and more straightforward is one of the goals of current research in reinforcement learning.

16.1 TD-Gammon

One of the most impressive applications of reinforcement learning to date is that by Gerald Tesauro to the game of backgammon (Tesauro, 1992, 1994, 1995, 2002). Tesauro’s program, *TD-Gammon*, required little backgammon knowledge, yet learned to play extremely well, near the level of the world’s strongest grandmasters. The learning algorithm in TD-Gammon was a straightforward combination of the $\text{TD}(\lambda)$ algorithm and nonlinear function approximation using a multilayer artificial neural network (ANN) trained by backpropagating TD errors.

Backgammon is a major game in the sense that it is played throughout the world, with numerous tournaments and regular world championship matches. It is in part a game of chance, and it is a popular vehicle for waging significant sums of money. There are probably more professional backgammon players than there are professional chess players. The game is played with 15 white and 15 black pieces on a board of 24 locations, called *points*. To the right on the next page is shown a typical position early in the game, seen from the perspective of the white player. White here has just rolled the dice and obtained a 5 and a 2. This means that he can move one of his pieces 5 steps and one

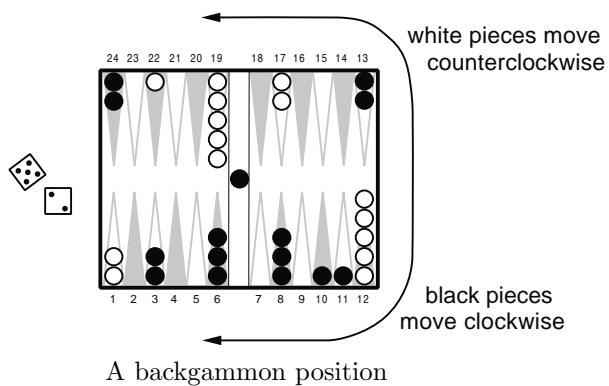
(possibly the same piece) 2 steps. For example, he could move two pieces from the 12 point, one to the 17 point, and one to the 14 point. White's objective is to advance all of his pieces into the last quadrant (points 19–24) and then off the board. The first player to remove all his pieces wins. One complication is that the pieces interact as they pass each other going in different directions. For example, if it were black's move, he could use the dice roll of 2 to move a piece from the 24 point to the 22 point,

“hitting” the white piece there. Pieces that have been hit are placed on the “bar” in the middle of the board (where we already see one previously hit black piece), from whence they reenter the race from the start. However, if there are two pieces on a point, then the opponent cannot move to that point; the pieces are protected from being hit. Thus, white cannot use his 5–2 dice roll to move either of his pieces on the 1 point, because their possible resulting points are occupied by groups of black pieces. Forming contiguous blocks of occupied points to block the opponent is one of the elementary strategies of the game.

Backgammon involves several further complications, but the above description gives the basic idea. With 30 pieces and 24 possible locations (26, counting the bar and off-the-board) it should be clear that the number of possible backgammon positions is enormous, far more than the number of memory elements one could have in any physically realizable computer. The number of moves possible from each position is also large. For a typical dice roll there might be 20 different ways of playing. In considering future moves, such as the response of the opponent, one must consider the possible dice rolls as well. The result is that the game tree has an effective branching factor of about 400. This is far too large to permit effective use of the conventional heuristic search methods that have proved so effective in games like chess and checkers.

On the other hand, the game is a good match to the capabilities of TD learning methods. Although the game is highly stochastic, a complete description of the game's state is available at all times. The game evolves over a sequence of moves and positions until finally ending in a win for one player or the other, ending the game. The outcome can be interpreted as a final reward to be predicted. On the other hand, the theoretical results we have described so far cannot be usefully applied to this task. The number of states is so large that a lookup table cannot be used, and the opponent is a source of uncertainty and time variation.

TD-Gammon used a nonlinear form of $\text{TD}(\lambda)$. The estimated value, $\hat{v}(s, w)$, of any state (board position) s was meant to estimate the probability of winning starting from state s . To achieve this, rewards were defined as zero for all time steps except those on which the game is won. To implement the value function, TD-Gammon used a standard multilayer ANN, much like that shown to the right on the next page. (The real



A backgammon position

network had two additional units in its final layer to estimate the probability of each player's winning in a special way called a "gammon" or "backgammon.") The network consisted of a layer of input units, a layer of hidden units, and a final output unit. The input to the network was a representation of a backgammon position, and the output was an estimate of the value of that position.

In the first version of TD-Gammon, TD-Gammon 0.0, backgammon positions were represented to the network in a relatively direct way that involved little backgammon knowledge. It did, however, involve substantial knowledge of how ANNs work and how information is best presented to them. It is instructive to note the exact representation Tesauro chose. There were a total of 198 input units to the network. For each point on the backgammon board, four units indicated the number of white pieces on the point. If there were no white pieces, then all four units took on the value zero. If there was one piece, then the first unit took on the value 1. This encoded the elementary concept of a "blot," i.e., a piece that can be hit by the opponent. If there were two or more pieces, then the second unit was set to 1. This encoded the basic concept of a "made point" on which the opponent cannot land. If there were exactly three pieces on the point, then the third unit was set to 1. This encoded the basic concept of a "single spare," i.e., an extra piece in addition to the two pieces that made the point. Finally, if there were more than three pieces, the fourth unit was set to a value proportionate to the number of additional pieces beyond three. Letting n denote the total number of pieces on the point, if $n > 3$, then the fourth unit took on the value $(n - 3)/2$. This encoded a linear representation of "multiple spares" at the given point.

With four units for white and four for black at each of the 24 points, that made a total of 192 units. Two additional units encoded the number of white and black pieces on the bar (each took the value $n/2$, where n is the number of pieces on the bar), and two more encoded the number of black and white pieces already successfully removed from the board (these took the value $n/15$, where n is the number of pieces already borne off). Finally, two units indicated in a binary fashion whether it was white's or black's turn to move. The general logic behind these choices should be clear. Basically, Tesauro tried to represent the position in a straightforward way, while keeping the number of units relatively small. He provided one unit for each conceptually distinct possibility that seemed likely to be relevant, and he scaled them to roughly the same range, in this case between 0 and 1.

Given a representation of a backgammon position, the network computed its estimated value in the standard way. Corresponding to each connection from an input unit to a hidden unit was a real-valued weight. Signals from each input unit were multiplied by

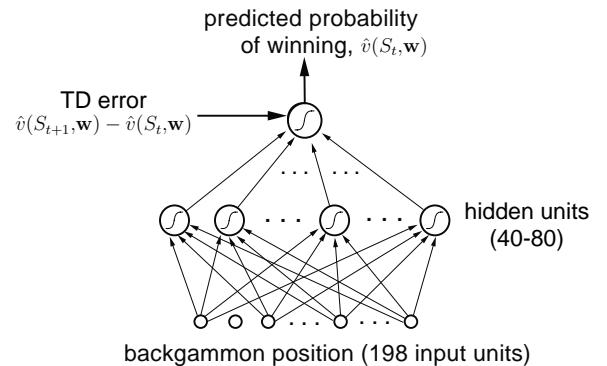


Figure 16.1: The TD-Gammon ANN

their corresponding weights and summed at the hidden unit. The output, $h(j)$, of hidden unit j was a nonlinear sigmoid function of the weighted sum:

$$h(j) = \sigma \left(\sum_i w_{ij} x_i \right) = \frac{1}{1 + e^{-\sum_i w_{ij} x_i}},$$

where x_i is the value of the i th input unit and w_{ij} is the weight of its connection to the j th hidden unit (all the weights in the network together make up the parameter vector \mathbf{w}). The output of the sigmoid is always between 0 and 1, and has a natural interpretation as a probability based on a summation of evidence. The computation from hidden units to the output unit was entirely analogous. Each connection from a hidden unit to the output unit had a separate weight. The output unit formed the weighted sum and then passed it through the same sigmoid nonlinearity.

TD-Gammon used the semi-gradient form of the TD(λ) algorithm described in Section 12.2, with the gradients computed by the error backpropagation algorithm (Rumelhart, Hinton, and Williams, 1986). Recall that the general update rule for this case is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)] \mathbf{z}_t, \quad (16.1)$$

where \mathbf{w}_t is the vector of all modifiable parameters (in this case, the weights of the network) and \mathbf{z}_t is a vector of eligibility traces, one for each component of \mathbf{w}_t , updated by

$$\mathbf{z}_t \doteq \gamma \lambda \mathbf{z}_{t-1} + \nabla \hat{v}(S_t, \mathbf{w}_t),$$

with $\mathbf{z}_0 \doteq \mathbf{0}$. The gradient in this equation can be computed efficiently by the backpropagation procedure. For the backgammon application, in which $\gamma = 1$ and the reward is always zero except upon winning, the TD error portion of the learning rule is usually just $\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$, as suggested in Figure 16.1.

To apply the learning rule we need a source of backgammon games. Tesauro obtained an unending sequence of games by playing his learning backgammon player against itself. To choose its moves, TD-Gammon considered each of the 20 or so ways it could play its dice roll and the corresponding positions that would result. The resulting positions are *afterstates* as discussed in Section 6.8. The network was consulted to estimate each of their values. The move was then selected that would lead to the position with the highest estimated value. Continuing in this way, with TD-Gammon making the moves for both sides, it was possible to easily generate large numbers of backgammon games. Each game was treated as an episode, with the sequence of positions acting as the states, S_0, S_1, S_2, \dots . Tesauro applied the nonlinear TD rule (16.1) fully incrementally, that is, after each individual move.

The weights of the network were set initially to small random values. The initial evaluations were thus entirely arbitrary. Because the moves were selected on the basis of these evaluations, the initial moves were inevitably poor, and the initial games often lasted hundreds or thousands of moves before one side or the other won, almost by accident. After a few dozen games however, performance improved rapidly.

After playing about 300,000 games against itself, TD-Gammon 0.0 as described above learned to play approximately as well as the best previous backgammon computer

programs. This was a striking result because all the previous high-performance computer programs had used extensive backgammon knowledge. For example, the reigning champion program at the time was, arguably, *Neurogammon*, another program written by Tesauro that used an ANN but not TD learning. Neurogammon's network was trained on a large training corpus of exemplary moves provided by backgammon experts, and, in addition, started with a set of features specially crafted for backgammon. Neurogammon was a highly tuned, highly effective backgammon program that decisively won the World Backgammon Olympiad in 1989. TD-Gammon 0.0, on the other hand, was constructed with essentially zero backgammon knowledge. That it was able to do as well as Neurogammon and all other approaches is striking testimony to the potential of self-play learning methods.

The tournament success of TD-Gammon 0.0 with zero expert backgammon knowledge suggested an obvious modification: add the specialized backgammon features but keep the self-play TD learning method. This produced TD-Gammon 1.0. TD-Gammon 1.0 was clearly substantially better than all previous backgammon programs and found serious competition only among human experts. Later versions of the program, TD-Gammon 2.0 (40 hidden units) and TD-Gammon 2.1 (80 hidden units), were augmented with a selective two-ply search procedure. To select moves, these programs looked ahead not just to the positions that would immediately result, but also to the opponent's possible dice rolls and moves. Assuming the opponent always took the move that appeared immediately best for him, the expected value of each candidate move was computed and the best was selected. To save computer time, the second ply of search was conducted only for candidate moves that were ranked highly after the first ply, about four or five moves on average. Two-ply search affected only the moves selected; the learning process proceeded exactly as before. The final versions of the program, TD-Gammon 3.0 and 3.1, used 160 hidden units and a selective three-ply search. TD-Gammon illustrates the combination of learned value functions and decision-time search as in heuristic search and MCTS methods. In follow-on work, Tesauro and Galperin (1997) explored trajectory sampling methods as an alternative to full-width search, which reduced the error rate of live play by large numerical factors (4x–6x) while keeping the think time reasonable at ~5–10 seconds per move.

During the 1990s, Tesauro was able to play his programs in a significant number of games against world-class human players. A summary of the results is given in Table 16.1.

Program	Hidden Units	Training Games	Opponents	Results
TD-Gammon 0.0	40	300,000	other programs	tied for best
TD-Gammon 1.0	80	300,000	Robertie, Magriel, ...	-13 pts / 51 games
TD-Gammon 2.0	40	800,000	various Grandmasters	-7 pts / 38 games
TD-Gammon 2.1	80	1,500,000	Robertie	-1 pt / 40 games
TD-Gammon 3.0	80	1,500,000	Kazaros	+6 pts / 20 games

Table 16.1: Summary of TD-Gammon Results

Based on these results and analyses by backgammon grandmasters (Robertie, 1992; see Tesauro, 1995), TD-Gammon 3.0 appeared to play at close to, or possibly better than, the playing strength of the best human players in the world. Tesauro reported in a subsequent article (Tesauro, 2002) the results of an extensive rollout analysis of the move decisions and doubling decisions of TD-Gammon relative to top human players. The conclusion was that TD-Gammon 3.1 had a “lopsided advantage” in piece-movement decisions, and a “slight edge” in doubling decisions, over top humans.

TD-Gammon had a significant impact on the way the best human players play the game. For example, it learned to play certain opening positions differently than was the convention among the best human players. Based on TD-Gammon’s success and further analysis, the best human players now play these positions as TD-Gammon does (Tesauro, 1995). The impact on human play was greatly accelerated when several other self-teaching ANN backgammon programs inspired by TD-Gammon, such as Jellyfish, Snowie, and GNUMBackgammon, became widely available. These programs enabled wide dissemination of new knowledge generated by the ANNs, resulting in great improvements in the overall caliber of human tournament play (Tesauro, 2002).

16.2 Samuel’s Checkers Player

An important precursor to Tesauro’s TD-Gammon was the seminal work of Arthur Samuel (1959, 1967) in constructing programs for learning to play checkers. Samuel was one of the first to make effective use of heuristic search methods and of what we would now call temporal-difference learning. His checkers players are instructive case studies in addition to being of historical interest. We emphasize the relationship of Samuel’s methods to modern reinforcement learning methods and try to convey some of Samuel’s motivation for using them.

Samuel first wrote a checkers-playing program for the IBM 701 in 1952. His first *learning* program was completed in 1955 and was demonstrated on television in 1956. Later versions of the program achieved good, though not expert, playing skill. Samuel was attracted to game-playing as a domain for studying machine learning because games are less complicated than problems “taken from life” while still allowing fruitful study of how heuristic procedures and learning can be used together. He chose to study checkers instead of chess because its relative simplicity made it possible to focus more strongly on learning.

Samuel’s programs played by performing a lookahead search from each current position. They used what we now call heuristic search methods to determine how to expand the search tree and when to stop searching. The terminal board positions of each search were evaluated, or “scored,” by a value function, or “scoring polynomial,” using linear function approximation. In this and other respects Samuel’s work seems to have been inspired by the suggestions of Shannon (1950). In particular, Samuel’s program was based on Shannon’s minimax procedure to find the best move from the current position. Working backward through the search tree from the scored terminal positions, each position was given the score of the position that would result from the best move, assuming that the machine would always try to maximize the score, while the opponent would always try to

minimize it. Samuel called this the “backed-up score” of the position. When the minimax procedure reached the search tree’s root—the current position—it yielded the best move under the assumption that the opponent would be using the same evaluation criterion, shifted to its point of view. Some versions of Samuel’s programs used sophisticated search control methods analogous to what are known as “alpha-beta” cutoffs (e.g., see Pearl, 1984).

Samuel used two main learning methods, the simplest of which he called *rote learning*. It consisted simply of saving a description of each board position encountered during play together with its backed-up value determined by the minimax procedure. The result was that if a position that had already been encountered were to occur again as a terminal position of a search tree, the depth of the search was effectively amplified because this position’s stored value cached the results of one or more searches conducted earlier. One initial problem was that the program was not encouraged to move along the most direct path to a win. Samuel gave it a “a sense of direction” by decreasing a position’s value a small amount each time it was backed up a level (called a ply) during the minimax analysis. “If the program is now faced with a choice of board positions whose scores differ only by the ply number, it will automatically make the most advantageous choice, choosing a low-ply alternative if winning and a high-ply alternative if losing” (Samuel, 1959, p. 80). Samuel found this discounting-like technique essential to successful learning. Rote learning produced slow but continual improvement that was most effective for opening and endgame play. His program became a “better-than-average novice” after learning from many games against itself, a variety of human opponents, and from book games in a supervised learning mode.

Rote learning and other aspects of Samuel’s work strongly suggest the essential idea of temporal-difference learning—that the value of a state should equal the value of likely following states. Samuel came closest to this idea in his second learning method, his “learning by generalization” procedure for modifying the parameters of the value function. Samuel’s method was the same in concept as that used much later by Tesauro in TD-Gammon. He played his program many games against another version of itself and performed an update after each move. The idea of Samuel’s update is suggested by the backup diagram in Figure 16.2. Each open circle represents a position where the program moves next, an *on-move* position, and each solid circle represents a position where the opponent moves next. An update was made to the value of each on-move position after a move by each side, resulting in a second on-move position. The update was toward the minimax value of a search launched from the second on-move position. Thus, the overall effect was that of a backing-up over one full move of real events and then a search over possible events, as suggested by Figure 16.2. Samuel’s actual algorithm was significantly more complex than this for computational reasons, but this was the basic idea.

Samuel did not include explicit rewards. Instead, he fixed the weight of the most important feature, the *piece advantage* feature, which measured the number of pieces the program had relative to how many its opponent had, giving higher weight to kings, and including refinements so that it was better to trade pieces when winning than when losing. Thus, the goal of Samuel’s program was to improve its piece advantage, which in checkers is highly correlated with winning.

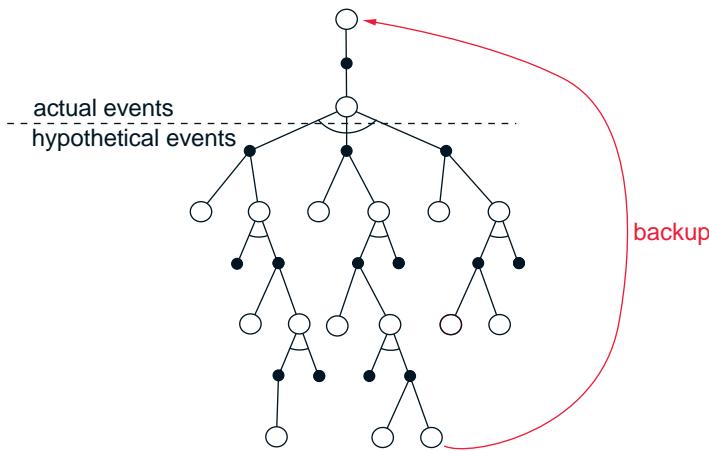


Figure 16.2: The backup diagram for Samuel’s checkers player.

However, Samuel’s learning method may have been missing an essential part of a sound temporal-difference algorithm. Temporal-difference learning can be viewed as a way of making a value function consistent with itself, and this we can clearly see in Samuel’s method. But also needed is a way of tying the value function to the true value of the states. We have enforced this via rewards and by discounting or giving a fixed value to the terminal state. But Samuel’s method included no rewards and no special treatment of the terminal positions of games. As Samuel himself pointed out, his value function could have become consistent merely by giving a constant value to all positions. He hoped to discourage such solutions by giving his piece-advantage term a large, nonmodifiable weight. But although this may decrease the likelihood of finding useless evaluation functions, it does not prohibit them. For example, a constant function could still be attained by setting the modifiable weights so as to cancel the effect of the nonmodifiable one.

Because Samuel’s learning procedure was not constrained to find useful evaluation functions, it should have been possible for it to become worse with experience. In fact, Samuel reported observing this during extensive self-play training sessions. To get the program improving again, Samuel had to intervene and set the weight with the largest absolute value back to zero. His interpretation was that this drastic intervention jarred the program out of local optima, but another possibility is that it jarred the program out of evaluation functions that were consistent but had little to do with winning or losing the game.

Despite these potential problems, Samuel’s checkers player using the generalization learning method approached “better-than-average” play. Fairly good amateur opponents characterized it as “tricky but beatable” (Samuel, 1959). In contrast to the rote-learning version, this version was able to develop a good middle game but remained weak in opening and endgame play. This program also included an ability to search through sets of features to find those that were most useful in forming the value function. A later version (Samuel, 1967) included refinements in its search procedure, such as alpha-beta pruning,

extensive use of a supervised learning mode called “book learning,” and hierarchical lookup tables called signature tables (Griffith, 1966) to represent the value function instead of linear function approximation. This version learned to play much better than the 1959 program, though still not at a master level. Samuel’s checkers-playing program was widely recognized as a significant achievement in artificial intelligence and machine learning.

16.3 Watson’s Daily-Double Wagering

IBM WATSON¹ is the system developed by a team of IBM researchers to play the popular TV quiz show *Jeopardy!*.² It gained fame in 2011 by winning first prize in an exhibition match against human champions. Although the main technical achievement demonstrated by WATSON was its ability to quickly and accurately answer natural language questions over broad areas of general knowledge, its winning *Jeopardy!* performance also relied on sophisticated decision-making strategies for critical parts of the game. Tesauro, Gondek, Lechner, Fan, and Prager (2012, 2013) adapted Tesauro’s TD-Gammon system described above to create the strategy used by WATSON in “Daily-Double” (DD) wagering in its celebrated winning performance against human champions. These authors report that the effectiveness of this wagering strategy went well beyond what human players are able to do in live game play, and that it, along with other advanced strategies, was an important contributor to WATSON’s impressive winning performance. Here we focus only on DD wagering because it is the component of WATSON that owes the most to reinforcement learning.

Jeopardy! is played by three contestants who face a board showing 30 squares, each of which hides a clue and has a dollar value. The squares are arranged in six columns, each corresponding to a different category. A contestant selects a square, the host reads the square’s clue, and each contestant may choose to respond to the clue by sounding a buzzer (“buzzing in”). The first contestant to buzz in gets to try responding to the clue. If this contestant’s response is correct, their score increases by the dollar value of the square; if their response is not correct, or if they do not respond within five seconds, their score decreases by that amount, and the other contestants get a chance to buzz in to respond to the same clue. One or two squares (depending on the game’s current round) are special DD squares. A contestant who selects one of these gets an exclusive opportunity to respond to the square’s clue and has to decide—before the clue is revealed—on how much to wager, or bet. The bet has to be greater than five dollars but not greater than the contestant’s current score. If the contestant responds correctly to the DD clue, their score increases by the bet amount; otherwise it decreases by the bet amount. At the end of each game is a “Final Jeopardy” (FJ) round in which each contestant writes down a sealed bet and then writes an answer after the clue is read. The contestant with the highest score after three rounds of play (where a round consists of revealing all 30 clues) is the winner. The game has many other details, but these are enough to appreciate

¹Registered trademark of IBM Corp.

²Registered trademark of Jeopardy Productions Inc.

the importance of DD wagering. Winning or losing often depends on a contestant's DD wagering strategy.

Whenever WATSON selected a DD square, it chose its bet by comparing action values, $\hat{q}(s, \text{bet})$, that estimated the probability of a win from the current game state, s , for each round-dollar legal bet. Except for some risk-abatement measures described below, WATSON selected the bet with the maximum action value. Action values were computed whenever a betting decision was needed by using two types of estimates that were learned before any live game play took place. The first were estimated values of the afterstates (Section 6.8) that would result from selecting each legal bet. These estimates were obtained from a state-value function, $\hat{v}(\cdot, \mathbf{w})$, defined by parameters \mathbf{w} , that gave estimates of the probability of a win for WATSON from any game state. The second estimates used to compute action values gave the "in-category DD confidence," p_{DD} , which estimated the likelihood that WATSON would respond correctly to the as-yet unrevealed DD clue.

Tesauro et al. used the reinforcement learning approach of TD-Gammon described above to learn $\hat{v}(\cdot, \mathbf{w})$: a straightforward combination of nonlinear TD(λ) using a multilayer ANN with weights \mathbf{w} trained by backpropagating TD errors during many simulated games. States were represented to the network by feature vectors specifically designed for *Jeopardy!*. Features included the current scores of the three players, how many DDs remained, the total dollar value of the remaining clues, and other information related to the amount of play left in the game. Unlike TD-Gammon, which learned by self-play, WATSON's \hat{v} was learned over millions of simulated games against carefully-crafted models of human players. In-category confidence estimates were conditioned on the number of right responses r and wrong responses w that WATSON gave in previously-played clues in the current category. The dependencies on (r, w) were estimated from WATSON's actual accuracies over many thousands of historical categories.

With the previously learned value function \hat{v} and in-category DD confidence p_{DD} , WATSON computed $\hat{q}(s, \text{bet})$ for each legal round-dollar bet as follows:

$$\hat{q}(s, \text{bet}) = p_{DD} \times \hat{v}(S_W + \text{bet}, \dots) + (1 - p_{DD}) \times \hat{v}(S_W - \text{bet}, \dots), \quad (16.2)$$

where S_W is WATSON's current score, and \hat{v} gives the estimated value for the game state after WATSON's response to the DD clue, which is either correct or incorrect. Computing an action value this way corresponds to the insight from Exercise 3.19 that an action value is the expected next state value given the action (except that here it is the expected next *afterstate* value because the full next state of the entire game depends on the next square selection).

Tesauro et al. found that selecting bets by maximizing action values incurred "a frightening amount of risk," meaning that if WATSON's response to the clue happened to be wrong, the loss could be disastrous for its chances of winning. To decrease the downside risk of a wrong answer, Tesauro et al. adjusted (16.2) by subtracting a small fraction of the standard deviation over WATSON's correct/incorrect afterstate evaluations. They further reduced risk by prohibiting bets that would cause the wrong-answer afterstate value to decrease below a certain limit. These measures slightly reduced WATSON's expectation of winning, but they significantly reduced downside risk, not only in terms of average risk per DD bet, but even more so in extreme-risk scenarios where a risk-neutral WATSON would bet most or all of its bankroll.

Why was the TD-Gammon method of self-play not used to learn the critical value function \hat{v} ? Learning from self-play in *Jeopardy!* would not have worked very well because WATSON was so different from any human contestant. Self-play would have led to exploration of state space regions that are not typical for play against human opponents, particularly human champions. In addition, unlike backgammon, *Jeopardy!* is a game of imperfect information because contestants do not have access to all the information influencing their opponents' play. In particular, *Jeopardy!* contestants do not know how much confidence their opponents have for responding to clues in the various categories. Self-play would have been something like playing poker with someone who is holding the same cards that you hold.

As a result of these complications, much of the effort in developing WATSON's DD-wagering strategy was devoted to creating good models of human opponents. The models did not address the natural language aspect of the game, but were instead stochastic process models of events that can occur during play. Statistics were extracted from an extensive fan-created archive of game information from the beginning of the show to the present day. The archive includes information such as the ordering of the clues, right and wrong contestant answers, DD locations, and DD and FJ bets for nearly 300,000 clues. Three models were constructed: an Average Contestant model (based on all the data), a Champion model (based on statistics from games with the 100 best players), and a Grand Champion model (based on statistics from games with the 10 best players). In addition to serving as opponents during learning, the models were used to assess the benefits produced by the learned DD-wagering strategy. WATSON's win rate in simulation when it used a baseline heuristic DD-wagering strategy was 61%; when it used the learned values and a default confidence value, its win rate increased to 64%; and with live in-category confidence, it was 67%. Tesauro et al. regarded this as a significant improvement, given that the DD wagering was needed only about 1.5 to 2 times in each game.

Because WATSON had only a few seconds to bet, as well as to select squares and decide whether or not to buzz in, the computation time needed to make these decisions was a critical factor. The ANN implementation of \hat{v} allowed DD bets to be made quickly enough to meet the time constraints of live play. However, once games could be simulated fast enough through improvements in the simulation software, near the end of a game it was feasible to estimate the value of bets by averaging over many Monte-Carlo trials in which the consequence of each bet was determined by simulating play to the game's end. Selecting endgame DD bets in live play based on Monte-Carlo trials instead of the ANN significantly improved WATSON's performance because errors in value estimates in endgames could seriously affect its chances of winning. Making all the decisions via Monte-Carlo trials might have led to better wagering decisions, but this was simply impossible given the complexity of the game and the time constraints of live play.

Although its ability to quickly and accurately answer natural language questions stands out as WATSON's major achievement, all of its sophisticated decision strategies contributed to its impressive defeat of human champions. According to Tesauro et al. (2012):

... it is plainly evident that our strategy algorithms achieve a level of quantitative precision and real-time performance that exceeds human capabilities.

This is particularly true in the cases of DD wagering and endgame buzzing, where humans simply cannot come close to matching the precise equity and confidence estimates and complex decision calculations performed by Watson.

16.4 Optimizing Memory Control

Most computers use dynamic random access memory (DRAM) as their main memory because of its low cost and high capacity. The job of a DRAM memory controller is to efficiently use the interface between the processor chip and an off-chip DRAM system to provide the high-bandwidth and low-latency data transfer necessary for high-speed program execution. A memory controller needs to deal with dynamically changing patterns of read/write requests while adhering to a large number of timing and resource constraints required by the hardware. This is a formidable scheduling problem, especially with modern processors with multiple cores sharing the same DRAM.

İpek, Mutlu, Martínez, and Caruana (2008) (also Martínez and İpek, 2009) designed a reinforcement learning memory controller and demonstrated that it can significantly improve the speed of program execution over what was possible with conventional controllers at the time of their research. They were motivated by limitations of existing state-of-the-art controllers that used policies that did not take advantage of past scheduling experience and did not account for long-term consequences of scheduling decisions. İpek et al.’s project was carried out by means of simulation, but they designed the controller at the detailed level of the hardware needed to implement it—including the learning algorithm—directly on a processor chip.

Accessing DRAM involves a number of steps that have to be done according to strict time constraints. DRAM systems consist of multiple DRAM chips, each containing multiple rectangular arrays of storage cells arranged in rows and columns. Each cell stores a bit as the charge on a capacitor. Because the charge decreases over time, each DRAM cell needs to be recharged—refreshed—every few milliseconds to prevent memory content from being lost. This need to refresh the cells is why DRAM is called “dynamic.”

Each cell array has a row buffer that holds a row of bits that can be transferred into or out of one of the array’s rows. An *activate* command “opens a row,” which means moving the contents of the row whose address is indicated by the command into the row buffer. With a row open, the controller can issue *read* and *write* commands to the cell array. Each read command transfers a word (a short sequence of consecutive bits) in the row buffer to the external data bus, and each write command transfers a word in the external data bus to the row buffer. Before a different row can be opened, a *precharge* command must be issued which transfers the (possibly updated) data in the row buffer back into the addressed row of the cell array. After this, another activate command can open a new row to be accessed. Read and write commands are *column commands* because they sequentially transfer bits into or out of columns of the row buffer; multiple bits can be transferred without re-opening the row. Read and write commands to the currently-open row can be carried out more quickly than accessing a different row, which would involve additional *row commands*: precharge and activate; this is sometimes

referred to as “row locality.” A memory controller maintains a *memory transaction queue* that stores memory-access requests from the processors sharing the memory system. The controller has to process requests by issuing commands to the memory system while adhering to a large number of timing constraints.

A controller’s policy for scheduling access requests can have a large effect on the performance of the memory system, such as the average latency with which requests can be satisfied and the throughput the system is capable of achieving. The simplest scheduling strategy handles access requests in the order in which they arrive by issuing all the commands required by the request before beginning to service the next one. But if the system is not ready for one of these commands, or executing a command would result in resources being underutilized (e.g., due to timing constraints arising from servicing that one command), it makes sense to begin servicing a newer request before finishing the older one. Policies can gain efficiency by reordering requests, for example, by giving priority to read requests over write requests, or by giving priority to read/write commands to already open rows. The policy called First-Ready, First-Come-First-Serve (FR-FCFS), gives priority to column commands (read and write) over row commands (activate and precharge), and in case of a tie gives priority to the oldest command. FR-FCFS was shown to outperform other scheduling policies in terms of average memory-access latency under conditions commonly encountered (Rixner, 2004).

Figure 16.3 is a high-level view of İpek et al.’s reinforcement learning memory controller. They modeled the DRAM access process as an MDP whose states are the contents of the transaction queue and whose actions are commands to the DRAM system: *precharge*, *activate*, *read*, *write*, and *NoOp*. The reward signal is 1 whenever the action is *read* or *write*, and otherwise it is 0. State transitions were considered to be stochastic because the next state of the system not only depends on the scheduler’s command, but also on aspects of the system’s behavior that the scheduler cannot control, such as the workloads of the processor cores accessing the DRAM system.

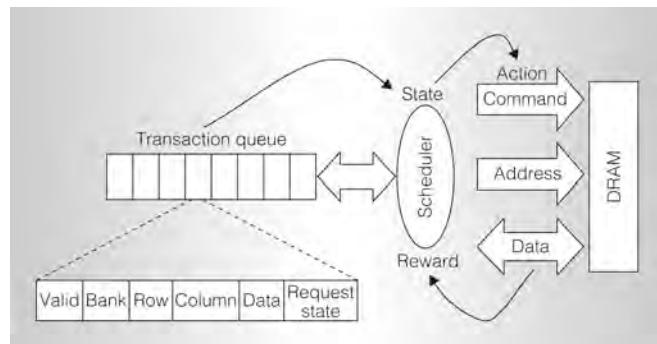


Figure 16.3: High-level view of the reinforcement learning DRAM controller. The scheduler is the reinforcement learning agent. Its environment is represented by features of the transaction queue, and its actions are commands to the DRAM system. ©2009 IEEE. Reprinted, with permission, from J. F. Martínez and E. İpek, Dynamic multicore resource management: A machine learning approach, *Micro, IEEE*, 29(5), p. 12.

Critical to this MDP are constraints on the actions available in each state. Recall from Chapter 3 that the set of available actions can depend on the state: $A_t \in \mathcal{A}(S_t)$, where A_t is the action at time step t and $\mathcal{A}(S_t)$ is the set of actions available in state S_t . In this application, the integrity of the DRAM system was assured by not allowing actions that would violate timing or resource constraints. Although İpek et al. did not make it explicit, they effectively accomplished this by pre-defining the sets $\mathcal{A}(S_t)$ for all possible states S_t .

These constraints explain why the MDP has a *NoOp* action and why the reward signal is 0 except when a *read* or *write* command is issued. *NoOp* is issued when it is the sole legal action in a state. To maximize utilization of the memory system, the controller’s task is to drive the system to states in which either a *read* or a *write* action can be selected: only these actions result in sending data over the external data bus, so it is only these that contribute to the throughput of the system. Although *precharge* and *activate* produce no immediate reward, the agent needs to select these actions to make it possible to later select the rewarded *read* and *write* actions.

The scheduling agent used Sarsa (Section 6.4) to learn an action-value function. States were represented by six integer-valued features. To approximate the action-value function, the algorithm used linear function approximation implemented by tile coding with hashing (Section 9.5.4). The tile coding had 32 tilings, each storing 256 action values as 16-bit fixed point numbers. Exploration was ε -greedy with $\varepsilon = 0.05$.

State features included the number of read requests in the transaction queue, the number of write requests in the transaction queue, the number of write requests in the transaction queue waiting for their row to be opened, and the number of read requests in the transaction queue waiting for their row to be opened that are the oldest issued by their requesting processors. (The other features depended on how the DRAM interacts with cache memory, details we omit here.) The selection of the state features was based on İpek et al.’s understanding of factors that impact DRAM performance. For example, balancing the rate of servicing reads and writes based on how many of each are in the transaction queue can help avoid stalling the DRAM system’s interaction with cache memory. The authors in fact generated a relatively long list of potential features, and then pared them down to a handful using simulations guided by stepwise feature selection.

An interesting aspect of this formulation of the scheduling problem as an MDP is that the features input to the tile coding for defining the action-value function were different from the features used to specify the action-constraint sets $\mathcal{A}(S_t)$. Whereas the tile coding input was derived from the contents of the transaction queue, the constraint sets depended on a host of other features related to timing and resource constraints that had to be satisfied by the hardware implementation of the entire system. In this way, the action constraints ensured that the learning algorithm’s exploration could not endanger the integrity of the physical system, while learning was effectively limited to a “safe” region of the much larger state space of the hardware implementation.

Because an objective of this work was that the learning controller could be implemented on a chip so that learning could occur online while a computer is running, hardware implementation details were important considerations. The design included two five-stage pipelines to calculate and compare two action values at every processor clock cycle, and

to update the appropriate action value. This included accessing the tile coding which was stored on-chip in static RAM. For the configuration İpek et al. simulated, which was a 4GHz 4-core chip typical of high-end workstations at the time of their research, there were 10 processor cycles for every DRAM cycle. Considering the cycles needed to fill the pipes, up to 12 actions could be evaluated in each DRAM cycle. İpek et al. found that the number of legal commands for any state was rarely greater than this, and that performance loss was negligible if enough time was not always available to consider all legal commands. These and other clever design details made it feasible to implement the complete controller and learning algorithm on a multi-processor chip.

İpek et al. evaluated their learning controller in simulation by comparing it with three other controllers: (1) the FR-FCFS controller mentioned above that produces the best on-average performance, (2) a conventional controller that processes each request in order, and (3) an unrealizable ideal controller, called the Optimistic controller, able to sustain 100% DRAM throughput if given enough demand by ignoring all timing and resource constraints, but otherwise modeling DRAM latency (as row buffer hits) and bandwidth. They simulated nine memory-intensive parallel workloads consisting of scientific and data-mining applications. Figure 16.4 shows the performance (the inverse of execution time normalized to the performance of FR-FCFS) of each controller for the nine applications, together with the geometric mean of their performances over the applications. The learning controller, labeled RL in the figure, improved over that of FR-FCFS by from 7% to 33% over the nine applications, with an average improvement of 19%. Of course, no realizable controller can match the performance of Optimistic, which ignores all timing and resource constraints, but the learning controller's performance closed the gap with Optimistic's upper bound by an impressive 27%.

Because the rationale for on-chip implementation of the learning algorithm was to allow the scheduling policy to adapt online to changing workloads, İpek et al. analyzed the impact of online learning compared to a previously-learned fixed policy. They trained

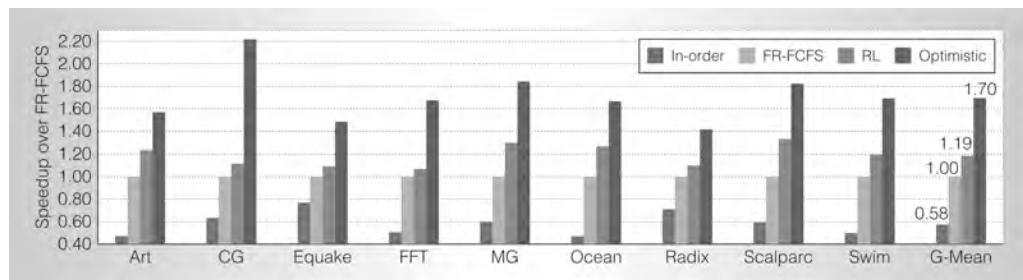


Figure 16.4: Performances of four controllers over a suite of 9 simulated benchmark applications. The controllers are: the simplest ‘in-order’ controller, FR-FCFS, the learning controller RL, and the unrealizable Optimistic controller which ignores all timing and resource constraints to provide a performance upper bound. Performance, normalized to that of FR-FCFS, is the inverse of execution time. At far right is the geometric mean of performances over the 9 benchmark applications for each controller. Controller RL comes closest to the ideal performance. ©2009 IEEE. Reprinted, with permission, from J. F. Martínez and E. İpek, Dynamic multicore resource management: A machine learning approach, *Micro, IEEE*, 29(5), p. 13.

their controller with data from all nine benchmark applications and then held the resulting action values fixed throughout the simulated execution of the applications. They found that the average performance of the controller that learned online was 8% better than that of the controller using the fixed policy, leading them to conclude that online learning is an important feature of their approach.

This learning memory controller was never committed to physical hardware because of the large cost of fabrication. Nevertheless, İpek et al. could convincingly argue on the basis of their simulation results that a memory controller that learns online via reinforcement learning has the potential to improve performance to levels that would otherwise require more complex and more expensive memory systems, while removing from human designers some of the burden required to manually design efficient scheduling policies. Mukundan and Martínez (2012) took this project forward by investigating learning controllers with additional actions, other performance criteria, and more complex reward functions derived using genetic algorithms. They considered additional performance criteria related to energy efficiency. The results of these studies surpassed the earlier results described above and significantly surpassed the 2012 state-of-the-art for all of the performance criteria they considered. The approach is especially promising for developing sophisticated power-aware DRAM interfaces.

16.5 Human-level Video Game Play

One of the greatest challenges in applying reinforcement learning to real-world problems is deciding how to represent and store value functions and/or policies. Unless the state set is finite and small enough to allow exhaustive representation by a lookup table—as in many of our illustrative examples—one must use a parameterized function approximation scheme. Whether linear or nonlinear, function approximation relies on features that have to be readily accessible to the learning system and able to convey the information necessary for skilled performance. Most successful applications of reinforcement learning owe much to sets of features carefully handcrafted based on human knowledge and intuition about the specific problem to be tackled.

A team of researchers at Google DeepMind developed an impressive demonstration that a deep multi-layer ANN can automate the feature design process (Mnih et al., 2013, 2015). Multi-layer ANNs have been used for function approximation in reinforcement learning ever since the 1986 popularization of the backpropagation algorithm as a method for learning internal representations (Rumelhart, Hinton, and Williams, 1986; see Section 9.7). Striking results have been obtained by coupling reinforcement learning with backpropagation. The results obtained by Tesauro and colleagues with TD-Gammon and WATSON discussed above are notable examples. These and other applications benefited from the ability of multi-layer ANNs to learn task-relevant features. However, in all the examples of which we are aware, the most impressive demonstrations required the network’s input to be represented in terms of specialized features handcrafted for the given problem. This is vividly apparent in the TD-Gammon results. TD-Gammon 0.0, whose network input was essentially a “raw” representation of the backgammon board, meaning that it involved very little knowledge of backgammon, learned to play approximately as well as

the best previous backgammon computer programs. Adding specialized backgammon features produced TD-Gammon 1.0 which was substantially better than all previous backgammon programs and competed well against human experts.

Mnih et al. developed a reinforcement learning agent called *deep Q-network* (DQN) that combined Q-learning with a *deep convolutional ANN*, a many-layered, or deep, ANN specialized for processing spatial arrays of data such as images. We describe deep convolutional ANNs in Section 9.7. By the time of Mnih et al.’s work with DQN, deep ANNs, including deep convolutional ANNs, had produced impressive results in many applications, but they had not been widely used in reinforcement learning.

Mnih et al. used DQN to show how a reinforcement learning agent can achieve a high level of performance on any of a collection of different problems without having to use different problem-specific feature sets. To demonstrate this, they let DQN learn to play 49 different Atari 2600 video games by interacting with a game emulator. DQN learned a different policy for each of the 49 games (because the weights of its ANN were reset to random values before learning on each game), but it used the same raw input, network architecture, and parameter values (e.g., step size, discount rate, exploration parameters, and many more specific to the implementation) for all the games. DQN achieved levels of play at or beyond human level on a large fraction of these games. Although the games were alike in being played by watching streams of video images, they varied widely in other respects. Their actions had different effects, they had different state-transition dynamics, and they needed different policies for learning high scores. The deep convolutional ANN learned to transform the raw input common to all the games into features specialized for representing the action values required for playing at the high level DQN achieved for most of the games.

The Atari 2600 is a home video game console that was sold in various versions by Atari Inc. from 1977 to 1992. It introduced or popularized many arcade video games that are now considered classics, such as Pong, Breakout, Space Invaders, and Asteroids. Although much simpler than modern video games, Atari 2600 games are still entertaining and challenging for human players, and they have been attractive as testbeds for developing and evaluating reinforcement learning methods (Diuk, Cohen, Littman, 2008; Naddaf, 2010; Cobo, Zang, Isbell, and Thomaz, 2011; Bellemare, Veness, and Bowling, 2013). Bellemare, Naddaf, Veness, and Bowling (2012) developed the publicly available Arcade Learning Environment (ALE) to encourage and simplify using Atari 2600 games to study learning and planning algorithms.

These previous studies and the availability of ALE made the Atari 2600 game collection a good choice for Mnih et al.’s demonstration, which was also influenced by the impressive human-level performance that TD-Gammon was able to achieve in backgammon. DQN is similar to TD-Gammon in using a multi-layer ANN as the function approximation method for a semi-gradient form of a TD algorithm, with the gradients computed by the backpropagation algorithm. However, instead of using $\text{TD}(\lambda)$ as TD-Gammon did, DQN used the semi-gradient form of Q-learning. TD-Gammon estimated the values of afterstates, which were easily obtained from the rules for making backgammon moves. To use the same algorithm for the Atari games would have required generating the next states for each possible action (which would not have been afterstates in that case). This could have been done by using the game emulator to run single-step simulations

for all the possible actions (which ALE makes possible). Or a model of each game’s state-transition function could have been learned and used to predict next states (Oh, Guo, Lee, Lewis, and Singh, 2015). While these methods might have produced results comparable to DQN’s, they would have been more complicated to implement and would have significantly increased the time needed for learning. Another motivation for using Q-learning was that DQN used the *experience replay* method, described below, which requires an off-policy algorithm. Being model-free and off-policy made Q-learning a natural choice.

Before describing the details of DQN and how the experiments were conducted, we look at the skill levels DQN was able to achieve. Mnih et al. compared the scores of DQN with the scores of the best performing learning system in the literature at the time, the scores of a professional human games tester, and the scores of an agent that selected actions at random. The best system from the literature used linear function approximation with features designed using some knowledge about Atari 2600 games (Bellemare, Naddaf, Veness, and Bowling, 2013). DQN learned on each game by interacting with the game emulator for 50 million frames, which corresponds to about 38 days of experience with the game. At the start of learning on each game, the weights of DQN’s network were reset to random values. To evaluate DQN’s skill level after learning, its score was averaged over 30 sessions on each game, each lasting up to 5 minutes and beginning with a random initial game state. The professional human tester played using the same emulator (with the sound turned off to remove any possible advantage over DQN which did not process audio). After 2 hours of practice, the human played about 20 episodes of each game for up to 5 minutes each and was not allowed to take any break during this time. DQN learned to play better than the best previous reinforcement learning systems on all but 6 of the games, and played better than the human player on 22 of the games. By considering any performance that scored at or above 75% of the human score to be comparable to, or better than, human-level play, Mnih et al. concluded that the levels of play DQN learned reached or exceeded human level on 29 of the 46 games. See Mnih et al. (2015) for a more detailed account of these results.

For an artificial learning system to achieve these levels of play would be impressive enough, but what makes these results remarkable—and what many at the time considered to be breakthrough results for artificial intelligence—is that the very same learning system achieved these levels of play on widely varying games without relying on any game-specific modifications.

A human playing any of these 49 Atari games sees 210×160 pixel image frames with 128 colors at 60Hz. In principle, exactly these images could have formed the raw input to DQN, but to reduce memory and processing requirements, Mnih et al. preprocessed each frame to produce an 84×84 array of luminance values. Because the full states of many of the Atari games are not completely observable from the image frames, Mnih et al. “stacked” the four most recent frames so that the inputs to the network had dimension $84 \times 84 \times 4$. This did not eliminate partial observability for all of the games, but it was helpful in making many of them more Markovian.

An essential point here is that these preprocessing steps were exactly the same for all 46 games. No game-specific prior knowledge was involved beyond the general understanding that it should still be possible to learn good policies with this reduced dimension and

that stacking adjacent frames should help with the partial observability of some of the games. Because no game-specific prior knowledge beyond this minimal amount was used in preprocessing the image frames, we can think of the $84 \times 84 \times 4$ input vectors as being “raw” input to DQN.

The basic architecture of DQN is similar to the deep convolutional ANN illustrated in Figure 9.15 (though unlike that network, subsampling in DQN is treated as part of each convolutional layer, with feature maps consisting of units having only a selection of the possible receptive fields). DQN has three hidden convolutional layers, followed by one fully connected hidden layer, followed by the output layer. The three successive hidden convolutional layers of DQN produce 32 20×20 feature maps, 64 9×9 feature maps, and 64 7×7 feature maps. The activation function of the units of each feature map is a rectifier nonlinearity ($\max(0, x)$). The 3,136 ($64 \times 7 \times 7$) units in this third convolutional layer all connect to each of 512 units in the fully connected hidden layer, which then each connect to all 18 units in the output layer, one for each possible action in an Atari game.

The activation levels of DQN’s output units were the estimated optimal action values of the corresponding state–action pairs, for the state represented by the network’s input. The assignment of output units to a game’s actions varied from game to game, and because the number of valid actions varied between 4 and 18 for the games, not all output units had functional roles in all of the games. It helps to think of the network as if it were 18 separate networks, one for estimating the optimal action value of each possible action. In reality, these networks shared their initial layers, but the output units learned to use the features extracted by these layers in different ways.

DQN’s reward signal indicated how a game’s score changed from one time step to the next: +1 whenever it increased, -1 whenever it decreased, and 0 otherwise. This standardized the reward signal across the games and made a single step-size parameter work well for all the games despite their varying ranges of scores. DQN used an ε -greedy policy, with ε decreasing linearly over the first million frames and remaining at a low value for the rest of the learning session. The values of the various other parameters, such as the learning step size, discount rate, and others specific to the implementation, were selected by performing informal searches to see which values worked best for a small selection of the games. These values were then held fixed for all of the games.

After DQN selected an action, the action was executed by the game emulator, which returned a reward and the next video frame. The frame was preprocessed and added to the four-frame stack that became the next input to the network. Skipping for the moment the changes to the basic Q-learning procedure made by Mnih et al., DQN used the following semi-gradient form of Q-learning to update the network’s weights:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \quad (16.3)$$

where \mathbf{w}_t is the vector of the network’s weights, A_t is the action selected at time step t , and S_t and S_{t+1} are respectively the preprocessed image stacks input to the network at time steps t and $t+1$.

The gradient in (16.3) was computed by backpropagation. Imagining again that there was a separate network for each action, for the update at time step t , backpropagation was applied only to the network corresponding to A_t . Mnih et al. took advantage of techniques shown to improve the basic backpropagation algorithm when applied to large

networks. They used a *mini-batch method* that updated weights only after accumulating gradient information over a small batch of images (here after 32 images). This yielded smoother sample gradients compared to the usual procedure that updates weights after each action. They also used a gradient-ascent algorithm called RMSProp (Tieleman and Hinton, 2012) that accelerates learning by adjusting the step-size parameter for each weight based on a running average of the magnitudes of recent gradients for that weight.

Mnih et al. modified the basic Q-learning procedure in three ways. First, they used a method called *experience replay* first studied by Lin (1992). This method stores the agent’s experience at each time step in a replay memory that is accessed to perform the weight updates. It worked like this in DQN. After the game emulator executed action A_t in a state represented by the image stack S_t , and returned reward R_{t+1} and image stack S_{t+1} , it added the tuple $(S_t, A_t, R_{t+1}, S_{t+1})$ to the replay memory. This memory accumulated experiences over many plays of the same game. At each time step multiple Q-learning updates—a mini-batch—were performed based on experiences sampled uniformly at random from the replay memory. Instead of S_{t+1} becoming the new S_t for the next update as it would in the usual form of Q-learning, a new unconnected experience was drawn from the replay memory to supply data for the next update. Because Q-learning is an off-policy algorithm, it does not need to be applied along connected trajectories.

Q-learning with experience replay provided several advantages over the usual form of Q-learning. The ability to use each stored experience for many updates allowed DQN to learn more efficiently from its experiences. Experience replay reduced the variance of the updates because successive updates were not correlated with one another as they would be with standard Q-learning. And by removing the dependence of successive experiences on the current weights, experience replay eliminated one source of instability.

Mnih et al. modified standard Q-learning in a second way to improve its stability. As in other methods that bootstrap, the target for a Q-learning update depends on the current action-value function estimate. When a parameterized function approximation method is used to represent action values, the target is a function of the same parameters that are being updated. For example, the target in the update given by (16.3) is $\gamma \max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$. Its dependence on \mathbf{w}_t complicates the process compared to the simpler supervised-learning situation in which the targets do not depend on the parameters being updated. As discussed in Chapter 11 this can lead to oscillations and/or divergence.

To address this problem Mnih et al. used a technique that brought Q-learning closer to the simpler supervised-learning case while still allowing it to bootstrap. Whenever a certain number, C , of updates had been done to the weights \mathbf{w} of the action-value network, they inserted the network’s current weights into another network and held these duplicate weights fixed for the next C updates of \mathbf{w} . The outputs of this duplicate network over the next C updates of \mathbf{w} were used as the Q-learning targets. Letting \tilde{q} denote the output of this duplicate network, then instead of (16.3) the update rule was:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t).$$

A final modification of standard Q-learning was also found to improve stability. They clipped the error term $R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)$ so that it remained in the interval $[-1, 1]$.

Mnih et al. conducted a large number of learning runs on 5 of the games to gain insight into the effect that various of DQN’s design features had on its performance. They ran DQN with the four combinations of experience replay and the duplicate target network being included or not included. Although the results varied from game to game, each of these features alone significantly improved performance, and very dramatically improved performance when used together. Mnih et al. also studied the role played by the deep convolutional ANN in DQN’s learning ability by comparing the deep convolutional version of DQN with a version having a network of just one linear layer, both receiving the same stacked preprocessed video frames. Here, the improvement of the deep convolutional version over the linear version was particularly striking across all 5 of the test games.

Creating artificial agents that excel over a diverse collection of challenging tasks has been an enduring goal of artificial intelligence. The promise of machine learning as a means for achieving this has been frustrated by the need to craft problem-specific representations. DeepMind’s DQN stands as a major step forward by demonstrating that a single agent can learn problem-specific features enabling it to acquire human-competitive skills over a range of tasks. This demonstration did not produce one agent that simultaneously excelled at all the tasks (because learning occurred separately for each task), but it showed that deep learning can reduce, and possibly eliminate, the need for problem-specific design and tuning. As Mnih et al. point out, however, DQN is not a complete solution to the problem of task-independent learning. Although the skills needed to excel on the Atari games were markedly diverse, all the games were played by observing video images, which made a deep convolutional ANN a natural choice for this collection of tasks. In addition, DQN’s performance on some of the Atari 2600 games fell considerably short of human skill levels on these games. The games most difficult for DQN—especially Montezuma’s Revenge on which DQN learned to perform about as well as the random player—require deep planning beyond what DQN was designed to do. Further, learning control skills through extensive practice, like DQN learned how to play the Atari games, is just one of the types of learning humans routinely accomplish. Despite these limitations, DQN advanced the state-of-the-art in machine learning by impressively demonstrating the promise of combining reinforcement learning with modern methods of deep learning.

16.6 Mastering the Game of Go

The ancient Chinese game of Go has challenged artificial intelligence researchers for many decades. Methods that achieve human-level skill, or even superhuman-level skill, in other games have not been successful in producing strong Go programs. Thanks to a very active community of Go programmers and international competitions, the level of Go program play has improved significantly over the years. Until recently, however, no Go program had been able to play anywhere near the level of a human Go master.

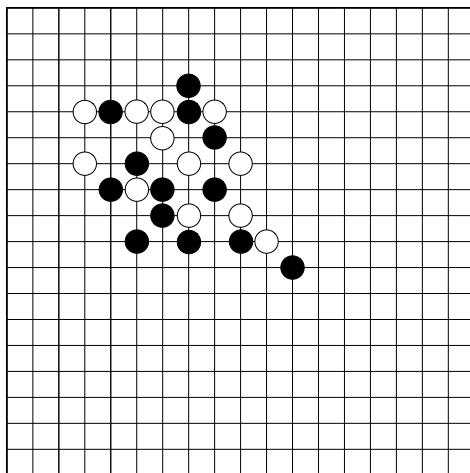
A team at DeepMind (Silver et al., 2016) developed the program *AlphaGo* that broke this barrier by combining deep ANNs (Section 9.7), supervised learning, Monte Carlo

tree search (MCTS, Section 8.11), and reinforcement learning. By the time of Silver et al.’s 2016 publication, *AlphaGo* had been shown to be decisively stronger than other Go programs, and it had defeated the European Go champion Fan Hui 5 games to 0. These were the first victories of a Go program over a professional human Go player without handicap in full Go games. Shortly thereafter, a similar version of *AlphaGo* won stunning victories over the 18-time world champion Lee Sedol, winning 4 out of a 5 games in a challenge match, making worldwide headline news. Artificial intelligence researchers thought that it would be many more years, perhaps decades, before a program reached this level of play.

Here we describe *AlphaGo* and a successor program called *AlphaGo Zero* (Silver et al. 2017a). Where in addition to reinforcement learning, *AlphaGo* relied on supervised learning from a large database of expert human moves, *AlphaGo Zero* used only reinforcement learning and no human data or guidance beyond the basic rules of the game (hence the *Zero* in its name). We first describe *AlphaGo* in some detail in order to highlight the relative simplicity of *AlphaGo Zero*, which is both higher-performing and more of a pure reinforcement learning program.

In many ways, both *AlphaGo* and *AlphaGo Zero* are descendants of Tesauro’s TD-Gammon (Section 16.1), itself a descendant of Samuel’s checkers player (Section 16.2). All these programs included reinforcement learning over simulated games of self-play. *AlphaGo* and *AlphaGo Zero* also built upon the progress made by DeepMind on playing Atari games with the program DQN (Section 16.5) that used deep convolutional ANNs to approximate optimal value functions.

Go is a game between two players who alternately place black and white ‘stones’ on unoccupied intersections, or ‘points,’ on a board with a grid of 19 horizontal and 19 vertical lines to produce positions like that shown to the right. The game’s goal is to capture an area of the board larger than that captured by the opponent. Stones are captured according to simple rules. A player’s stones are captured if they are completely surrounded by the other player’s stones, meaning that there is no horizontally or vertically adjacent point that is unoccupied. For example, the left panel of Figure 16.5 (on the next page) shows three white stones with an unoccupied adjacent point (labeled X). If black were to place a stone on X, then the three white stones would be captured and taken off the board (middle panel). However, if white were to place a stone on point X first, then the possibility of this capture would be blocked (right panel). Other rules are needed to prevent infinite capturing/recapturing loops. The game ends when neither player wishes to place another stone. These rules are simple, but they produce a very complex game that has had wide appeal for thousands of years.



A Go board configuration

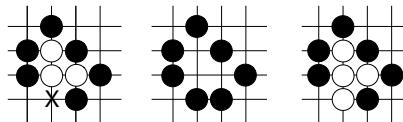


Figure 16.5: Go capturing rule. Left: the three white stones are not surrounded because point X is unoccupied. Middle: if black places a stone on X, the three white stones are captured and removed from the board. Right: if white places a stone on point X first, the capture is blocked.

Methods that produce strong play for other games, such as chess, have not worked as well for Go. The search space for Go is significantly larger than that of chess because Go has a larger number of legal moves per position than chess (≈ 250 versus ≈ 35) and Go games tend to involve more moves than chess games (≈ 150 versus ≈ 80). But the size of the search space is not the major factor that makes Go so difficult. Exhaustive search is infeasible for both chess and Go, and Go on smaller boards (e.g., 9×9) has proven to be exceedingly difficult as well. Experts agree that the major stumbling block to creating stronger-than-amateur Go programs is the difficulty of defining an adequate position evaluation function. A good evaluation function allows search to be truncated at a feasible depth by providing relatively easy-to-compute predictions of what deeper search would likely yield. According to Müller (2002): “No simple yet reasonable evaluation function will ever be found for Go.” A major step forward was the introduction of MCTS to Go programs. The strongest programs at the time of *AlphaGo*’s development all included MCTS, but master-level skill remained elusive.

Recall from Section 8.11 that MCTS is a decision-time planning procedure that does not attempt to learn and store a global evaluation function. Like a rollout algorithm (Section 8.10), it runs many Monte Carlo simulations of entire episodes (here, entire Go games) to select each action (here, each Go move: where to place a stone or to resign). Unlike a simple rollout algorithm, however, MCTS is an iterative procedure that incrementally extends a search tree whose root node represents the current environment state. As illustrated in Figure 8.10, each iteration traverses the tree by simulating actions guided by statistics associated with the tree’s edges. In its basic version, when a simulation reaches a leaf node of the search tree, MCTS expands the tree by adding some, or all, of the leaf node’s children to the tree. From the leaf node, or one of its newly added child nodes, a rollout is executed: a simulation that typically proceeds all the way to a terminal state, with actions selected by a rollout policy. When the rollout completes, the statistics associated with the search tree’s edges that were traversed in this iteration are updated by backing up the return produced by the rollout. MCTS continues this process, starting each time at the search tree’s root at the current state, for as many iterations as possible given the time constraints. Then, finally, an action from the root node (which still represents the current environment state) is selected according to statistics accumulated in the root node’s outgoing edges. This is the action the agent takes. After the environment transitions to its next state, MCTS is executed again with the root node set to represent the new current state. The search tree at the start of this next execution might be just this new root node, or it might include descendants of this node left over from MCTS’s previous execution. The remainder of the tree is discarded.

16.6.1 AlphaGo

The main innovation that made *AlphaGo* such a strong player is that it selected moves by a novel version of MCTS that was guided by both a policy and a value function learned by reinforcement learning with function approximation provided by deep convolutional ANNs. Another key feature is that instead of reinforcement learning starting from random network weights, it started from weights that were the result of previous supervised learning from a large collection of human expert moves.

The DeepMind team called *AlphaGo*'s modification of basic MCTS “asynchronous policy and value MCTS,” or APV-MCTS. It selected actions via basic MCTS as described above but with some twists in how it extended its search tree and how it evaluated action edges. In contrast to basic MCTS, which expands its current search tree by using stored action values to select an unexplored edge from a leaf node, APV-MCTS, as implemented in *AlphaGo*, expanded its tree by choosing an edge according to probabilities supplied by a 13-layer deep convolutional ANN, called the *SL-policy network*, trained previously by supervised learning to predict moves contained in a database of nearly 30 million human expert moves.

Then, also in contrast to basic MCTS, which evaluates the newly-added state node solely by the return of a rollout initiated from it, APV-MCTS evaluated the node in two ways: by this return of the rollout, but also by a value function, v_θ , learned previously by a reinforcement learning method. If s was the newly-added node, its value became

$$v(s) = (1 - \eta)v_\theta(s) + \eta G, \quad (16.4)$$

where G was the return of the rollout and η controlled the mixing of the values resulting from these two evaluation methods. In *AlphaGo*, these values were supplied by the *value network*, another 13-layer deep convolutional ANN that was trained as we describe below to output estimated values of board positions. APV-MCTS's rollouts in *AlphaGo* were simulated games with both players using a fast *rollout policy* provided by a simple linear network, also trained by supervised learning before play. Throughout its execution, APV-MCTS kept track of how many simulations passed through each edge of the search tree, and when its execution completed, the most-visited edge from the root node was selected as the action to take, here the move *AlphaGo* actually made in a game.

The value network had the same structure as the deep convolutional SL policy network except that it had a single output unit that gave estimated values of game positions instead of the SL policy network's probability distributions over legal actions. Ideally, the value network would output optimal state values, and it might have been possible to approximate the optimal value function along the lines of TD-Gammon described above: self-play with nonlinear $TD(\lambda)$ coupled to a deep convolutional ANN. But the DeepMind team took a different approach that held more promise for a game as complex as Go. They divided the process of training the value network into two stages. In the first stage, they created the best policy they could by using reinforcement learning to train an *RL policy network*. This was a deep convolutional ANN with the same structure as the SL policy network. It was initialized with the final weights of the SL policy network that were learned via supervised learning, and then policy-gradient reinforcement learning was used to improve upon the SL policy. In the second stage of training the value network,

the team used Monte Carlo policy evaluation on data obtained from a large number of simulated self-play games with moves selected by the RL policy network.

Figure 16.6 illustrates the networks used by *AlphaGo* and the steps taken to train them in what the DeepMind team called the “*AlphaGo* pipeline.” All these networks were trained before any live game play took place, and their weights remained fixed throughout live play.

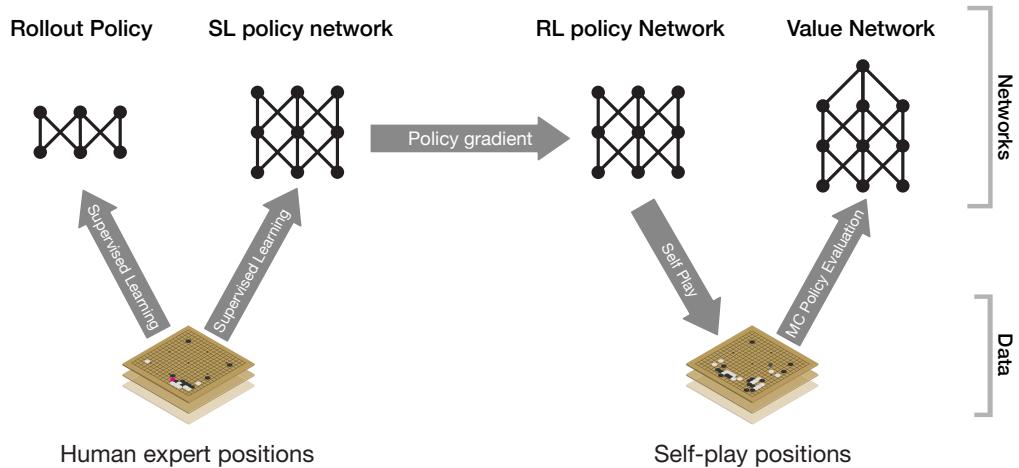


Figure 16.6: *AlphaGo* pipeline. Adapted with permission from Macmillan Publishers Ltd: *Nature*, vol. 529(7587), p. 485, ©2016.

Here is some more detail about *AlphaGo*’s ANNs and their training. The identically-structured SL and RL policy networks were similar to DQN’s deep convolutional network described in Section 16.5 for playing Atari games, except that they had 13 convolutional layers with the final layer consisting of a soft-max unit for each point on the 19×19 Go board. The networks’ input was a $19 \times 19 \times 48$ image stack in which each point on the Go board was represented by the values of 48 binary or integer-valued features. For example, for each point, one feature indicated if the point was occupied by one of *AlphaGo*’s stones, one of its opponent’s stones, or was unoccupied, thus providing the “raw” representation of the board configuration. Other features were based on the rules of Go, such as the number of adjacent points that were empty, the number of opponent stones that would be captured by placing a stone there, the number of turns since a stone was placed there, and other features that the design team considered to be important.

Training the SL policy network took approximately 3 weeks using a distributed implementation of stochastic gradient ascent on 50 processors. The network achieved 57% accuracy, where the best accuracy achieved by other groups at the time of publication was 44.4%. Training the RL policy network was done by policy gradient reinforcement learning over simulated games between the RL policy network’s current policy and opponents using policies randomly selected from policies produced by earlier iterations of the learning algorithm. Playing against a randomly selected collection of opponents

prevented overfitting to the current policy. The reward signal was +1 if the current policy won, -1 if it lost, and zero otherwise. These games directly pitted the two policies against one another without involving MCTS. By simulating many games in parallel on 50 processors, the DeepMind team trained the RL policy network on a million games in a single day. In testing the final RL policy, they found that it won more than 80% of games played against the SL policy, and it won 85% of games played against a Go program using MCTS that simulated 100,000 games per move.

The value network, whose structure was similar to that of the SL and RL policy networks except for its single output unit, received the same input as the SL and RL policy networks with the exception that there was an additional binary feature giving the current color to play. Monte Carlo policy evaluation was used to train the network from data obtained from a large number of self-play games played using the RL policy. To avoid overfitting and instability due to the strong correlations between positions encountered in self-play, the DeepMind team constructed a data set of 30 million positions each chosen randomly from a unique self-play game. Then training was done using 50 million mini-batches each of 32 positions drawn from this data set. Training took one week on 50 GPUs.

The rollout policy was learned prior to play by a simple linear network trained by supervised learning from a corpus of 8 million human moves. The rollout policy network had to output actions quickly while still being reasonably accurate. In principle, the SL or RL policy networks could have been used in the rollouts, but the forward propagation through these deep networks took too much time for either of them to be used in rollout simulations, a great many of which had to be carried out for each move decision during live play. For this reason, the rollout policy network was less complex than the other policy networks, and its input features could be computed more quickly than the features used for the policy networks. The rollout policy network allowed approximately 1,000 complete game simulations per second to be run on each of the processing threads that *AlphaGo* used.

One may wonder why the SL policy was used instead of the better RL policy to select actions in the expansion phase of APV-MCTS. These policies took the same amount of time to compute because they used the same network architecture. The team actually found that *AlphaGo* played better against human opponents when APV-MCTS used as the SL policy instead of the RL policy. They conjectured that the reason for this was that the latter was tuned to respond to optimal moves rather than to the broader set of moves characteristic of human play. Interestingly, the situation was reversed for the value function used by APV-MCTS. They found that when APV-MCTS used the value function derived from the RL policy, it performed better than if it used the value function derived from the SL policy.

Several methods worked together to produce *AlphaGo*'s impressive playing skill. The DeepMind team evaluated different versions of *AlphaGo* in order to assess the contributions made by these various components. The parameter η in (16.4) controlled the mixing of game state evaluations produced by the value network and by rollouts. With $\eta = 0$, *AlphaGo* used just the value network without rollouts, and with $\eta = 1$, evaluation relied just on rollouts. They found that *AlphaGo* using just the value network played

better than the rollout-only *AlphaGo*, and in fact played better than the strongest of all other Go programs existing at the time. The best play resulted from setting $\eta = 0.5$, indicating that combining the value network with rollouts was particularly important to *AlphaGo*'s success. These evaluation methods complemented one another: the value network evaluated the high-performance RL policy that was too slow to be used in live play, while rollouts using the weaker but much faster rollout policy were able to add precision to the value network's evaluations for specific states that occurred during games.

Overall, *AlphaGo*'s remarkable success fueled a new round of enthusiasm for the promise of artificial intelligence, specifically for systems combining reinforcement learning with deep ANNs, to address problems in other challenging domains.

16.6.2 AlphaGo Zero

Building upon the experience with *AlphaGo*, a DeepMind team developed *AlphaGo Zero* (Silver et al. 2017a). In contrast to *AlphaGo*, this program used *no human data or guidance beyond the basic rules of the game* (hence the *Zero* in its name). It learned exclusively from self-play reinforcement learning, with input giving just “raw” descriptions of the placements of stones on the Go board. *AlphaGo Zero* implemented a form of policy iteration (Section 4.3), interleaving policy evaluation with policy improvement. Figure 16.7 is an overview of *AlphaGo Zero*'s algorithm. A significant difference between *AlphaGo Zero* and *AlphaGo* is that *AlphaGo Zero* used MCTS to select moves throughout self-play reinforcement learning, whereas *AlphaGo* used MCTS for live play after—but not during—learning. Other differences besides not using any human data or human-crafted features are that *AlphaGo Zero* used only one deep convolutional ANN and used a simpler version of MCTS.

AlphaGo Zero's MCTS was simpler than the version used by *AlphaGo* in that it did not include rollouts of complete games, and therefore did not need a rollout policy. Each iteration of *AlphaGo Zero*'s MCTS ran a simulation that ended at a leaf node of the current search tree instead of at the terminal position of a complete game simulation. But as in *AlphaGo*, each iteration of MCTS in *AlphaGo Zero* was guided by the output of a deep convolutional network, labeled f_θ in Figure 16.7, where θ is the network's weight vector. The input to the network, whose architecture we describe below, consisted of raw representations of board positions, and its output had two parts: a scalar value, v , an estimate of the probability that the current player will win from from the current board position, and a vector, \mathbf{p} , of move probabilities, one for each possible stone placement on the current board, plus the pass, or resign, move.

Instead of selecting self-play actions according to the probabilities \mathbf{p} , however, *AlphaGo Zero* used these probabilities, together with the network's value output, to direct each execution of MCTS, which returned new move probabilities, shown in Figure 16.7 as the policies π_i . These policies benefitted from the many simulations that MCTS conducted each time it executed. The result was that the policy actually followed by *AlphaGo Zero* was an improvement over the policy given by the network's outputs \mathbf{p} . Silver et al. (2017a) wrote that “MCTS may therefore be viewed as a powerful *policy improvement operator*.”

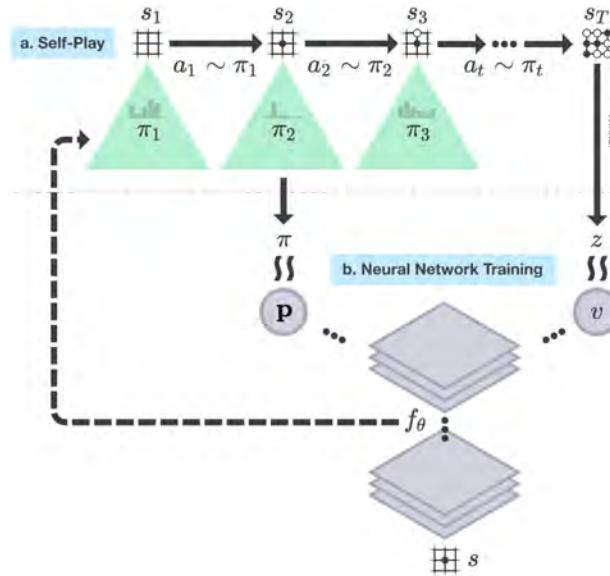


Figure 16.7: *AlphaGo Zero* self-play reinforcement learning. a) The program played many games against itself, one shown here as a sequence of board positions $s_i, i = 1, 2, \dots, T$, with moves $a_i, i = 1, 2, \dots, T$, and winner z . Each move a_i was determined by action probabilities π_i returned by MCTS executed from root node s_i and guided by a deep convolutional network, here labeled f_θ , with latest weights θ . Shown here for just one position s but repeated for all s_i , the network's inputs were raw representations of board positions s_i (together with several past positions, though not shown here), and its outputs were vectors p of move probabilities that guided MCTS's forward searches, and scalar values v that estimated the probability of the current player winning from each position s_i . b) Deep convolutional network training. Training examples were randomly sampled steps from recent self-play games. Weights θ were updated to move the policy vector p toward the probabilities π returned by MCTS, and to include the winners z in the estimated win probability v . Reprinted from draft of Silver et al. (2017a) with permission of the authors and DeepMind.

Here is more detail about *AlphaGo Zero*'s ANN and how it was trained. The network took as input a $19 \times 19 \times 17$ image stack consisting of 17 binary feature planes. The first 8 feature planes were raw representations of the positions of the current player's stones in the current and seven past board configurations: a feature value was 1 if a player's stone was on the corresponding point, and was 0 otherwise. The next 8 feature planes similarly coded the positions of the opponent's stones. A final input feature plane had a constant value indicating the color of the current play: 1 for black; 0 for white. Because repetition is not allowed in Go and one player is given some number of "compensation points" for not getting the first move, the current board position is not a Markov state of Go. This is why features describing past board positions and the color feature were needed.

The network was "two-headed," meaning that after a number of initial layers, the network split into two separate "heads" of additional layers that separately fed into two sets of output units. In this case, one head fed 362 output units producing $19^2 + 1$ move

probabilities \mathbf{p} , one for each possible stone placement plus pass; the other head fed just one output unit producing the scalar v , an estimate of the probability that the current player will win from the current board position. The network before the split consisted of 41 convolutional layers, each followed by batch normalization, and with skip connections added to implement residual learning by pairs of layers (see Section 9.7). Overall, move probabilities and values were computed by 43 and 44 layers respectively.

Starting with random weights, the network was trained by stochastic gradient descent (with momentum, regularization, and step-size parameter decreasing as training continues) using batches of examples sampled uniformly at random from all the steps of the most recent 500,000 games of self-play with the current best policy. Extra noise was added to the network’s output \mathbf{p} to encourage exploration of all possible moves. At periodic checkpoints during training, which Silver et al. (2017a) chose to be at every 1,000 training steps, the policy output by the ANN with the latest weights was evaluated by simulating 400 games (using MCTS with 1,600 iterations to select each move) against the current best policy. If the new policy won (by a margin set to reduce noise in the outcome), then it became the best policy to be used in subsequent self-play. The network’s weights were updated to make the network’s policy output \mathbf{p} more closely match the policy returned by MCTS, and to make its value output, v , more closely match the probability that the current best policy wins from the board position represented by the network’s input.

The DeepMind team trained *AlphaGo Zero* over 4.9 million games of self-play, which took about 3 days. Each move of each game was selected by running MCTS for 1,600 iterations, taking approximately 0.4 second per move. Network weights were updated over 700,000 batches each consisting of 2,048 board configurations. They then ran tournaments with the trained *AlphaGo Zero* playing against the version of *AlphaGo* that defeated Fan Hui by 5 games to 0, and against the version that defeated Lee Sedol by 4 games to 1. They used the Elo rating system to evaluate the relative performances of the programs. The difference between two Elo ratings is meant to predict the outcome of games between the players. The Elo ratings of *AlphaGo Zero*, the version of *AlphaGo* that played against Fan Hui, and the version that played against Lee Sedol were respectively 4,308, 3,144, and 3,739. The gaps in these Elo ratings translate into predictions that *AlphaGo Zero* would defeat these other programs with probabilities very close to one. In a match of 100 games between *AlphaGo Zero*, trained as described, and the exact version of *AlphaGo* that defeated Lee Sedol held under the same conditions that were used in that match, *AlphaGo Zero* defeated *AlphaGo* in all 100 games.

The DeepMind team also compared *AlphaGo Zero* with a program using an ANN with the same architecture but trained by supervised learning to predict human moves in a data set containing nearly 30 million positions from 160,000 games. They found that the supervised-learning player initially played better than *AlphaGo Zero*, and was better at predicting human expert moves, but played less well after *AlphaGo Zero* was trained for a day. This suggested that *AlphaGo Zero* had discovered a strategy for playing that was different from how humans play. In fact, *AlphaGo Zero* discovered, and came to prefer, some novel variations of classical move sequences.

Final tests of *AlphaGo Zero*’s algorithm were conducted with a version having a larger ANN and trained over 29 million self-play games, which took about 40 days, again starting

with random weights. This version achieved an Elo rating of 5,185. The team pitted this version of *AlphaGo Zero* against a program called *AlphaGo Master*, the strongest program at the time, that was identical to *AlphaGo Zero* but, like *AlphaGo*, used human data and features. *AlphaGo Master*'s Elo rating was 4,858, and it had defeated the strongest human professional players 60 to 0 in online games. In a 100 game match, *AlphaGo Zero* with the larger network and more extensive learning defeated *AlphaGo Master* 89 games to 11, thus providing a convincing demonstration of the problem-solving power of *AlphaGo Zero*'s algorithm.

AlphaGo Zero soundly demonstrated that superhuman performance can be achieved by pure reinforcement learning, augmented by a simple version of MCTS, and deep ANNs with very minimal knowledge of the domain and no reliance on human data or guidance. We will surely see systems inspired by the DeepMind accomplishments of both *AlphaGo* and *AlphaGo Zero* applied to challenging problems in other domains.

Recently, yet a better program, *AlphaZero*, was described by Silver et al. (2017b) that does not even incorporate knowledge of Go. *AlphaZero* is a general reinforcement learning algorithm that improves over the world's hitherto best programs in the diverse games of Go, chess, and shogi.

16.7 Personalized Web Services

Personalizing web services such as the delivery of news articles or advertisements is one approach to increasing users' satisfaction with a website or to increase the yield of a marketing campaign. A policy can recommend content considered to be the best for each particular user based on a profile of that user's interests and preferences inferred from their history of online activity. This is a natural domain for machine learning, and in particular, for reinforcement learning. A reinforcement learning system can improve a recommendation policy by making adjustments in response to user feedback. One way to obtain user feedback is by means of website satisfaction surveys, but for acquiring feedback in real time it is common to monitor user clicks as indicators of interest in a link.

A method long used in marketing called *A/B testing* is a simple type of reinforcement learning used to decide which of two versions, A or B, of a website users prefer. Because it is non-associative, like a two-armed bandit problem, this approach does not personalize content delivery. Adding context consisting of features describing individual users and the content to be delivered allows personalizing service. This has been formalized as a contextual bandit problem (or an associative reinforcement learning problem, Section 2.9) with the objective of maximizing the total number of user clicks. Li, Chu, Langford, and Schapire (2010) applied a contextual bandit algorithm to the problem of personalizing the Yahoo! Front Page Today webpage (one of the most visited pages on the internet at the time of their research) by selecting the news story to feature. Their objective was to maximize the *click-through rate* (CTR), which is the ratio of the total number of clicks all users make on a webpage to the total number of visits to the page. Their contextual bandit algorithm improved over a standard non-associative bandit algorithm by 12.5%.

Theocharous, Thomas, and Ghavamzadeh (2015) argued that better results are possible

by formulating personalized recommendation as a Markov decision problem (MDP) with the objective of maximizing the total number of clicks users make over repeated visits to a website. Policies derived from the contextual bandit formulation are greedy in the sense that they do not take long-term effects of actions into account. These policies effectively treat each visit to a website as if it were made by a new visitor uniformly sampled from the population of the website's visitors. By not using the fact that many users repeatedly visit the same websites, greedy policies do not take advantage of possibilities provided by long-term interactions with individual users.

As an example of how a marketing strategy might take advantage of long-term user interaction, Theocharous et al. contrasted a greedy policy with a longer-term policy for displaying ads for buying a product, say a car. The ad displayed by the greedy policy might offer a discount if the user buys the car immediately. A user either takes the offer or leaves the website, and if they ever return to the site, they would likely see the same offer. A longer-term policy, on the other hand, can transition the user "down a sales funnel" before presenting the final deal. It might start by describing the availability of favorable financing terms, then praise an excellent service department, and then, on the next visit, offer the final discount. This type of policy can result in more clicks by a user over repeated visits to the site, and if the policy is suitably designed, more eventual sales.

Working at Adobe Systems Incorporated, Theocharous et al. conducted experiments to see if policies designed to maximize clicks over the long term could in fact improve over short-term greedy policies. The Adobe Marketing Cloud, a set of tools that many companies use to run digital marketing campaigns, provides infrastructure for automating user-targeted advertising and fund-raising campaigns. Actually deploying novel policies using these tools entails significant risk because a new policy may end up performing poorly. For this reason, the research team needed to assess what a policy's performance would be if it were to be actually deployed, but to do so on the basis of data collected under the execution of other policies. A critical aspect of this research, then, was off-policy evaluation. Further, the team wanted to do this with high confidence to reduce the risk of deploying a new policy. Although high confidence off-policy evaluation was a central component of this research (see also Thomas, 2015; Thomas, Theocharous, and Ghavamzadeh, 2015), here we focus only on the algorithms and their results.

Theocharous et al. compared the results of two algorithms for learning ad recommendation policies. The first algorithm, which they called *greedy optimization*, had the goal of maximizing only the probability of immediate clicks. As in the standard contextual bandit formulation, this algorithm did not take the long-term effects of recommendations into account. The other algorithm, a reinforcement learning algorithm based on an MDP formulation, aimed at improving the number of clicks users made over multiple visits to a website. They called this latter algorithm *life-time value* (LTV) optimization. Both algorithms faced challenging problems because the reward signal in this domain is very sparse because users usually do not click on ads, and user clicking is very random so that returns have high variance.

Data sets from the banking industry were used for training and testing these algorithms. The data sets consisted of many complete trajectories of customer interaction with a bank's website that showed each customer one out of a collection of possible offers. If

a customer clicked, the reward was 1, and otherwise it was 0. One data set contained approximately 200,000 interactions from a month of a bank’s campaign that randomly offered one of 7 offers. The other data set from another bank’s campaign contained 4,000,000 interactions involving 12 possible offers. All interactions included customer features such as the time since the customer’s last visit to the website, the number of their visits so far, the last time the customer clicked, geographic location, one of a collection of interests, and features giving demographic information.

Greedy optimization was based on a mapping estimating the probability of a click as a function of user features. The mapping was learned via supervised learning from one of the data sets by means of a random forest (RF) algorithm (Breiman, 2001). RF algorithms have been widely used for large-scale applications in industry because they are effective predictive tools that tend not to overfit and are relatively insensitive to outliers and noise. Theocharous et al. then used the mapping to define an ε -greedy policy that selected with probability $1-\varepsilon$ the offer predicted by the RF algorithm to have the highest probability of producing a click, and otherwise selected from the other offers uniformly at random.

LTV optimization used a batch-mode reinforcement learning algorithm called *fitted Q iteration* (FQI). It is a variant of *fitted value iteration* (Gordon, 1999) adapted to Q-learning. Batch mode means that the entire data set for learning is available from the start, as opposed to the online mode of the algorithms we focus on in this book in which data are acquired sequentially while the learning algorithm executes. Batch-mode reinforcement learning algorithms are sometimes necessary when online learning is not practical, and they can use any batch-mode supervised learning regression algorithm, including algorithms known to scale well to high-dimensional spaces. The convergence of FQI depends on properties of the function approximation algorithm (Gordon, 1999). For their application to LTV optimization, Theocharous et al. used the same RF algorithm they used for the greedy optimization approach. Because in this case FQI convergence is not monotonic, Theocharous et al. kept track of the best FQI policy by off-policy evaluation using a validation training set. The final policy for testing the LTV approach was the ε -greedy policy based on the best policy produced by FQI with the initial action-value function set to the mapping produced by the RF for the greedy optimization approach.

To measure the performance of the policies produced by the greedy and LTV approaches, Theocharous et al. used the CTR metric and a metric they called the LTV metric. These metrics are similar, except that the LTV metric critically distinguishes between individual website visitors:

$$\text{CTR} = \frac{\text{Total \# of Clicks}}{\text{Total \# of Visits}},$$

$$\text{LTV} = \frac{\text{Total \# of Clicks}}{\text{Total \# of Visitors}}.$$

Figure 16.8 illustrates how these metrics differ. Each circle represents a user visit to the site; black circles are visits at which the user clicks. Each row represents visits by a particular user. By not distinguishing between visitors, the CTR for these sequences is 0.35, whereas the LTV is 1.5. Because LTV is larger than CTR to the extent that

individual users revisit the site, it is an indicator of how successful a policy is in encouraging users to engage in extended interactions with the site.

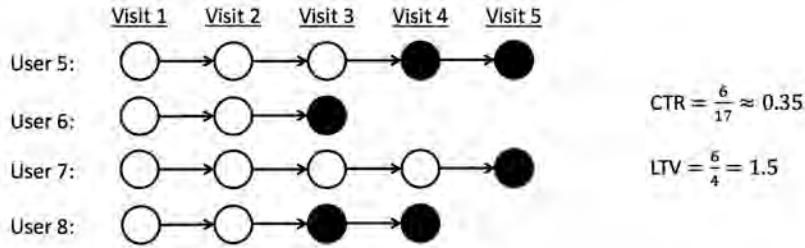


Figure 16.8: Click through rate (CTR) versus life-time value (LTV). Each circle represents a user visit; black circles are visits at which the user clicks. Adapted from Theocharous et al. (2015).

Testing the policies produced by the greedy and LTV approaches was done using a high confidence off-policy evaluation method on a test data set consisting of real-world interactions with a bank website served by a random policy. As expected, results showed that greedy optimization performed best as measured by the CTR metric, while LTV optimization performed best as measured by the LTV metric. Furthermore—although we have omitted its details—the high confidence off-policy evaluation method provided probabilistic guarantees that the LTV optimization method would, with high probability, produce policies that improve upon policies currently deployed. Assured by these probabilistic guarantees, Adobe announced in 2016 that the new LTV algorithm would be a standard component of the Adobe Marketing Cloud so that a retailer could issue a sequence of offers following a policy likely to yield higher return than a policy that is insensitive to long-term results.

16.8 Thermal Soaring

Birds and gliders take advantage of upward air currents—thermals—to gain altitude in order to maintain flight while expending little, or no, energy. Thermal soaring, as this behavior is called, is a complex skill requiring responding to subtle environmental cues to increase altitude by exploiting a rising column of air for as long as possible. Reddy, Celani, Sejnowski, and Vergassola (2016) used reinforcement learning to investigate thermal soaring policies that are effective in the strong atmospheric turbulence usually accompanying rising air currents. Their primary goal was to provide insight into the cues birds sense and how they use them to achieve their impressive thermal soaring performance, but the results also contribute to technology relevant to autonomous gliders. Reinforcement learning had previously been applied to the problem of navigating efficiently to the vicinity of a thermal updraft (Woodbury, Dunn, and Valasek, 2014) but not to the more challenging problem of soaring within the turbulence of the updraft itself.

Reddy et al. modeled the soaring problem as a continuing MDP with discounting. The agent interacted with a detailed model of a glider flying in turbulent air. They

devoted significant effort toward making the model generate realistic thermal soaring conditions, including investigating several different approaches to atmospheric modeling. For the learning experiments, air flow in a three-dimensional box with one kilometer sides, one of which was at ground level, was modeled by a sophisticated physics-based set of partial differential equations involving air velocity, temperature, and pressure. Introducing small random perturbations into the numerical simulation caused the model to produce analogs of thermal updrafts and accompanying turbulence (Figure 16.9 Left) Glider flight was modeled by aerodynamic equations involving velocity, lift, drag, and other factors governing powerless flight of a fixed-wing aircraft. Maneuvering the glider involved changing its angle of attack (the angle between the glider's wing and the direction of air flow) and its bank angle (Figure 16.9 Right).

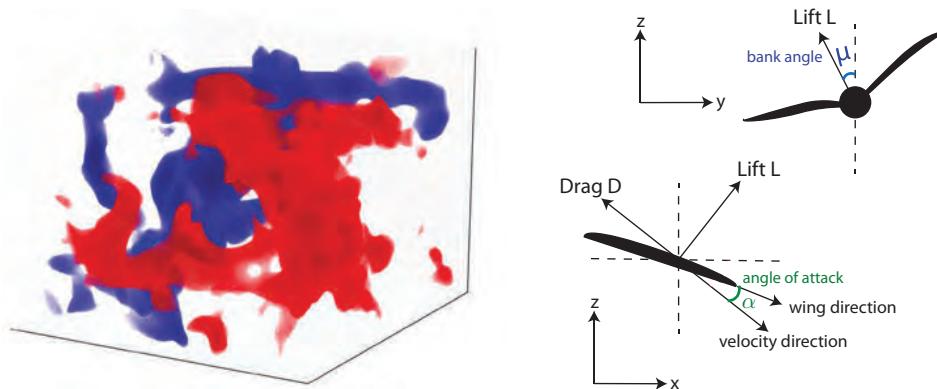


Figure 16.9: Thermal soaring model: Left: snapshot of the vertical velocity field of the simulated cube of air: in red (blue) is a region of large upward (downward) flow. Right: diagram of powerless flight showing bank angle μ and angle of attack α . Adapted with permission From PNAS vol. 113(22), p. E4879, 2016, Reddy, Celani, Sejnowski, and Vergassola, Learning to Soar in Turbulent Environments.

The interface between the agent and the environment required defining the agent's actions, the state information the agent receives from the environment, and the reward signal. By experimenting with various possibilities, Reddy et al. decided that three actions each for the angle of attack and the bank angle were enough for their purposes: increment or decrement the current bank angle and angle of attack by 5° and 2.5° , respectively, or leave them unchanged. This resulted in 3^2 possible actions. The bank angle was bounded to remain between -15° and $+15^\circ$.

Because a goal of their study was to try to determine what minimal set of sensory cues are necessary for effective soaring, both to shed light on the cues birds might use for soaring and to minimize the sensing complexity required for automated glider soaring, the authors tried various sets of signals as input to the reinforcement learning agent. They started by using state aggregation (Section 9.3) of a four-dimensional state space with dimensions giving local vertical wind speed, local vertical wind acceleration, torque depending on the difference between the vertical wind velocities at the left and right wing tips, and the local temperature. Each dimension was discretized into three bins: positive

high, negative high, and small. Results, described below, showed that only two of these dimensions were critical for effective soaring behavior.

The overall objective of thermal soaring is to gain as much altitude as possible from each rising column of air. Reddy et al. tried a straightforward reward signal that rewarded the agent at the end of each episode based on the altitude gained over the episode, a large negative reward signal if the glider touched the ground, and zero otherwise. They found that learning was not successful with this reward signal for episodes of realistic duration and that eligibility traces did not help. By experimenting with various reward signals, they found that learning was best with a reward signal that at each time step linearly combined the vertical wind velocity and vertical wind acceleration observed on the previous time step.

Learning was by one-step Sarsa, with actions selected according to a soft-max distribution based on normalized action values. Specifically, the action probabilities were computed according to (13.2) with action preferences:

$$h(s, a, \theta) = \frac{\hat{q}(s, a, \theta) - \min_b \hat{q}(s, b, \theta)}{\tau(\max_b \hat{q}(s, b, \theta) - \min_b \hat{q}(s, b, \theta))},$$

where θ is a parameter vector with one component for each action and aggregated group of states, and $\hat{q}(s, a, \theta)$ merely returned the component corresponding to s, a in the usual way for state aggregation methods. The above equation forms the action preferences by normalizing the approximate action values to the interval $[0, 1]$ then dividing by τ , a positive “temperature parameter.”³ As τ increases, the probability of selecting an action becomes less dependent on its preference; as τ decreases toward zero, the probability of selecting the most highly-preferred action approaches one, making the policy approach the greedy policy. The temperature parameter τ was initialized to 2.0 and incrementally decreased to 0.2 during learning. Action preferences were computed from the current estimates of the action values: the action with the maximum estimated action value was given preference $1/\tau$, the action with the minimum estimated action value was given preference 0, and the preferences of the other actions were scaled between these extremes. The step-size and discount-rate parameters were fixed at 0.1 and 0.98 respectively.

Each learning episode took place with the agent controlling simulated flight in an independently generated period of simulated turbulent air currents. Each episode lasted 2.5 minutes simulated with a 1 second time step. Learning effectively converged after a few hundred episodes. The left panel of Figure 16.10 shows a sample trajectory before learning where the agent selects actions randomly. Starting at the top of the volume shown, the glider’s trajectory is in the direction indicated by the arrow and quickly loses altitude. Figure 16.10’s right panel is a trajectory after learning. The glider starts at the same place (here appearing at the bottom of the volume) and gains altitude by spiraling within the rising column of air. Although Reddy et al. found that performance varied widely over different simulated periods of air flow, the number of times the glider touched the ground consistently decreased to nearly zero as learning progressed.

After experimenting with different sets of features available to the learning agent, it turned out that the combination of just vertical wind acceleration and torques worked

³Reddy et al. described this slightly differently, but our version is equivalent to theirs.

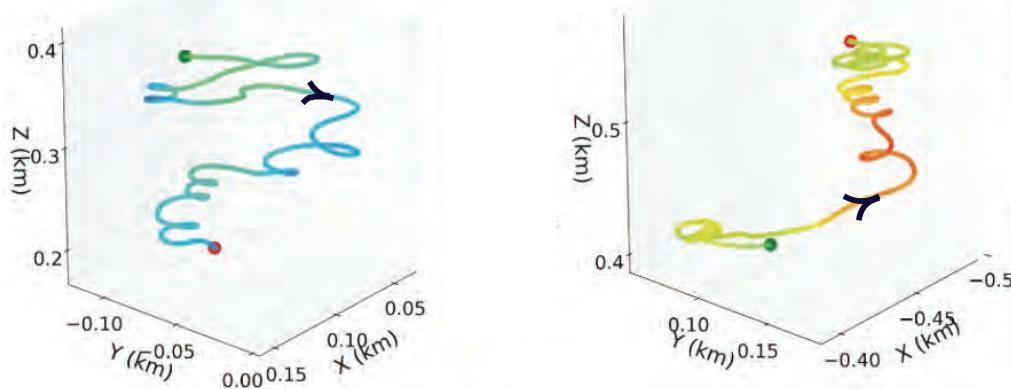


Figure 16.10: Sample thermal soaring trajectories, with arrows showing the direction of flight from the same starting point (note that the altitude scales are shifted). Left: before learning: the agent selects actions randomly and the glider descends. Right: after learning: the glider gains altitude by following a spiral trajectory. Adapted with permission from PNAS vol. 113(22), p. E4879, 2016, Reddy, Celani, Sejnowski, and Vergassola, Learning to Soar in Turbulent Environments.

best. The authors conjectured that because these features give information about the gradient of vertical wind velocity in two different directions, they allow the controller to select between turning by changing the bank angle or continuing along the same course by leaving the bank angle alone. This allows the glider to stay within a rising column of air. Vertical wind velocity is indicative of the strength of the thermal but does not help in staying within the flow. They found that sensitivity to temperature was of little help. They also found that controlling the angle of attack is not helpful in staying within a particular thermal, being useful instead for traveling between thermals when covering large distances, as in cross-country gliding and bird migration.

Due to the fact that soaring in different levels of turbulence requires different policies, training was done in conditions ranging from weak to strong turbulence. In strong turbulence the rapidly changing wind and glider velocities allowed less time for the controller to react. This reduced the amount of control possible compared to what was possible for maneuvering when fluctuations were weak. Reddy et al. examined the policies Sarsa learned under these different conditions. Common to policies learned in all regimes were these features: when sensing negative wind acceleration, bank sharply in the direction of the wing with the higher lift; when sensing large positive wind acceleration and no torque, do nothing. However, different levels of turbulence led to policy differences. Policies learned in strong turbulence were more conservative in that they preferred small bank angles, whereas in weak turbulence, the best action was to turn as much as possible by banking sharply. Systematic study of the bank angles preferred by the policies learned under the different conditions led the authors to suggest that by detecting when vertical

wind acceleration crosses a certain threshold the controller can adjust its policy to cope with different turbulence regimes.

Reddy et al. also conducted experiments to investigate the effect of the discount-rate parameter γ on the performance of the learned policies. They found that the altitude gained in an episode increased as γ increased, reaching a maximum for $\gamma = .99$, suggesting that effective thermal soaring requires taking into account long-term effects of control decisions.

This computational study of thermal soaring illustrates how reinforcement learning can further progress toward different kinds of objectives. Learning policies having access to different sets of environmental cues and control actions contributes to both the engineering objective of designing autonomous gliders and the scientific objective of improving understanding of the soaring skills of birds. In both cases, hypotheses resulting from the learning experiments can be tested in the field by instrumenting real gliders⁴ and by comparing predictions with observed bird soaring behavior.

⁴This work has recently been applied to real gliders. See Reddy, Wong-Ng, Celani, Sejnowski, and Vergassola, “Glider soaring via reinforcement learning in the field.” *Nature* 562:236–239, 2018.

Chapter 17

Frontiers

In this final chapter we touch on some topics that are beyond the scope of this book but that we see as particularly important for the future of reinforcement learning. Many of these topics bring us beyond what is reliably known, and some bring us beyond the MDP framework.

17.1 General Value Functions and Auxiliary Tasks

Over the course of this book, our notion of value function has become quite general. With off-policy learning we allowed a value function to be conditional on an arbitrary target policy. Then in Section 12.8 we generalized discounting to a *termination function* $\gamma : \mathcal{S} \mapsto [0, 1]$, so that a different discount rate could be applied at each time step in determining the return (12.17). This allowed us to express predictions about how much reward we will get over an arbitrary, state-dependent horizon. The next, and perhaps final, step is to generalize beyond rewards to permit predictions about arbitrary signals. Rather than predicting the sum of future rewards, we might predict the sum of the future values of a sound or color sensation, or of an internal, highly processed signal such as another prediction. Whatever signal is added up in this way in a value-function-like prediction, we call it the *cumulant* of that prediction. We formalize it in a *cumulant signal* $C_t \in \mathbb{R}$. Using this, a *general value function*, or GVF, is written

$$v_{\pi, \gamma, C}(s) \doteq \mathbb{E} \left[\sum_{k=t}^{\infty} \left(\prod_{i=t+1}^k \gamma(S_i) \right) C_{k+1} \mid S_t = s, A_{t:\infty} \sim \pi \right]. \quad (17.1)$$

As with conventional value functions (such as v_π or q_*) this is an ideal function that we seek to approximate with a parameterized form, which we might continue to denote $\hat{v}(s, \mathbf{w})$, although of course there would have to be a different \mathbf{w} for each prediction, that is, for each choice of π , γ , and C . Because a GVF has no necessary connection to reward, it is perhaps a misnomer to call it a *value* function. You might simply call it a prediction or, to make it more distinctive, a *forecast* (Ring, in preparation). Whatever it is called, it

is in the form of a value function and thus can be learned in the usual ways using the methods developed in this book for learning approximate value functions. Along with the learned predictions, we might also learn policies to maximize the predictions in the usual ways by Generalized Policy Iteration (Section 4.6) or by actor–critic methods. In this way an agent could learn to predict and control great numbers of signals, not just long-term reward.

Why might it be useful to predict and control signals other than long-term reward? These are *auxiliary* tasks in that they are extra (in addition to) the main task of maximizing reward. One answer is that the ability to predict and control a diverse multitude of signals can constitute a powerful kind of environmental model. As we saw in Chapter 8, a good model can enable the agent to get reward more efficiently. It takes a couple of further concepts to develop this answer clearly, so we postpone it to the next section. First let’s consider two simpler ways in which a multitude of diverse predictions can be helpful to a reinforcement learning agent.

One simple way in which auxiliary tasks can help on the main task is that they may require some of the same representations as are needed on the main task. Some of the auxiliary tasks may be easier, with less delay and a clearer connection between actions and outcomes. If good features can be found early on easy auxiliary tasks, then those features may significantly speed learning on the main task. There is no necessary reason why this has to be true, but in many cases it seems plausible. For example, if you learn to predict and control your sensors over short time scales, say seconds, then you might plausibly come up with part of the idea of physical objects, which would then greatly help with the prediction and control of long-term reward.

We might imagine an artificial neural network (ANN) in which the last layer is split into multiple parts, or *heads*, each working on a different task. One head might produce the approximate value function for the main task (with reward as its cumulant) whereas the others would produce solutions to various auxiliary tasks. All heads could propagate errors by stochastic gradient descent into the same body—the shared preceding part of the network—which would then try to form representations, in its next-to-last layer, to support all the heads. Researchers have experimented with auxiliary tasks such as predicting change in pixels, predicting the next time step’s reward, and predicting the distribution of the return. In many cases this approach has been shown to greatly accelerate learning on the main task (Jaderberg et al., 2017). Multiple predictions have similarly been repeatedly proposed as a way of directing the construction of state estimates (see Section 17.3).

Another simple way in which the learning of auxiliary tasks can improve performance is best explained by analogy to the psychological phenomena of classical conditioning (Section 14.2). One way of understanding classical conditioning is that evolution has built in a reflexive (non-learned) association to a particular action from the prediction of a particular signal. For example, humans and many other animals appear to have a built-in reflex to blink whenever their prediction of being poked in the eye exceeds some threshold. The prediction is learned, but the association from prediction to eye closure is built in, and thus the animal is saved many unprotected pokes in its eye. Similarly, the association from fear to increased heart rate, or to freezing, may be built in. Agent

designers can do something similar, connecting by design (without learning) predictions of specific events to predetermined actions. For example, a self-driving car that learns to predict whether going forward will produce a collision could be given a built-in reflex to stop, or to turn away, whenever the prediction is above some threshold. Or consider a vacuum-cleaning robot that learned to predict whether it might run out of battery power before returning to the charger and that reflexively headed back to the charger whenever the prediction became non-zero. The correct prediction would depend on the size of the house, the room the robot was in, and the age of the battery, all of which would be hard for the robot designer to know. It would be difficult for the designer to build in a reliable algorithm for deciding whether to head back to the charger in sensory terms, but it might be easy to do this in terms of the learned prediction. We foresee many possible ways like this in which learned predictions might combine usefully with built-in algorithms for controlling behavior.

Finally, perhaps the most important role for auxiliary tasks is in moving beyond the assumption we have made throughout this book that the state representation is fixed and given to the agent. To explain this role, we first have to take a few steps back to appreciate the magnitude of this assumption and the implications of removing it. We do that in Section 17.3.

17.2 Temporal Abstraction via Options

An appealing aspect of the MDP formalism is that it can be applied usefully to tasks at many different time scales. It can be used to formalize the task of deciding which muscles to twitch to grasp an object, which airplane flight to take to arrive conveniently at a distant city, and which job to take to lead a satisfying life. These tasks differ greatly in their time scales, yet each can be usefully formulated as an MDP that can be solved by planning or learning processes as described in this book. All involve interaction with the world, sequential decision making, and a goal usefully conceived of as accumulating rewards over time, and so all can be formulated as MDPs.

Although all these tasks can be formulated as MDPs, you might think that they cannot be formulated as a *single* MDP. They involve such different time scales, such different notions of choice and action! It would be no good, for example, to plan a flight across a continent at the level of muscle twitches. Yet for other tasks—such as grasping objects, throwing darts, or hitting a baseball—low-level muscle twitches may be just the right level. People do all these things seamlessly without appearing to switch between levels. Can the MDP framework be stretched to cover all the levels simultaneously?

Perhaps it can. One popular idea is to formalize an MDP at a detailed level, with a small time step, yet enable planning at higher levels using extended courses of action that correspond to many base-level time steps. To do this we need a notion of course of action that extends over many time steps and includes a notion of termination. A general way to formulate these two ideas is as a policy, π , and a state-dependent termination function, γ , as in GVF_s. We define a pair of these as a generalized notion of action termed an *option*. To execute an option $\omega = \langle \pi_\omega, \gamma_\omega \rangle$ at time t is to obtain the action to take, A_t , from $\pi_\omega(\cdot|S_t)$, then terminate at time $t + 1$ with probability $1 - \gamma_\omega(S_{t+1})$. If the option does

not terminate at $t+1$, then A_{t+1} is selected from $\pi_\omega(\cdot|S_{t+1})$, and the option terminates at $t+2$ with probability $1 - \gamma_\omega(S_{t+2})$, and so on until eventual termination. It is convenient to consider low-level actions to be special cases of options—each action a corresponds to an option $\langle \pi_\omega, \gamma_\omega \rangle$ whose policy picks the action ($\pi_\omega(s) = a$ for all $s \in \mathcal{S}$) and whose termination function is zero ($\gamma_\omega(s) = 0$ for all $s \in \mathcal{S}^+$). Options effectively extend the action space. The agent can either select a low-level action/option, terminating after one time step, or select an extended option that might execute for many time steps before terminating.

Options are designed so that they are interchangeable with low-level actions. For example, the notion of an action-value function q_π naturally generalizes to an *option*-value function that takes a state and option as input and returns the expected return starting from that state, executing that option to termination, and thereafter following the policy, π . We can also generalize the notion of policy to a *hierarchical policy* that selects from options rather than actions, where options, when selected, execute until termination. With these ideas, many of the algorithms in this book can be generalized to learn approximate option-value functions and hierarchical policies. In the simplest case, the learning process ‘jumps’ from option initiation to option termination, with an update only occurring when an option terminates. More subtly, updates can be made on each time step, using “intra-option” learning algorithms, which in general require off-policy learning.

Perhaps the most important generalization made possible by option ideas is that of the environmental model as developed in Chapters 3, 4, and 8. The conventional model of an action is the state-transition probabilities and the expected immediate reward for taking the action in each state. How do conventional action models generalize to *option models*? For options, the appropriate model is again of two parts, one corresponding to the state transition resulting from executing the option and one corresponding to the expected cumulative reward along the way. The reward part of an option model, analogous to the expected reward for state-action pairs (3.5), is

$$r(s, \omega) \doteq \mathbb{E}[R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots + \gamma^{\tau-1} R_\tau \mid S_0 = s, A_{0:\tau-1} \sim \pi_\omega, \tau \sim \gamma_\omega], \quad (17.2)$$

for all options ω and all states $s \in \mathcal{S}$, where τ is the random time step at which the option terminates according to γ_ω . Note the role of the overall discounting parameter γ in this equation—discounting is according to γ , but termination of the option is according to γ_ω . The state-transition part of an option model is a little more subtle. This part of the model characterizes the probability of each possible resulting state (as in (3.4)), but now this state may result after various numbers of time steps, each of which must be discounted differently. The model for option ω specifies, for each state s that ω might start executing in, and for each state s' that ω might terminate in,

$$p(s' | s, \omega) \doteq \sum_{k=1}^{\infty} \gamma^k \Pr\{S_k = s', \tau = k \mid S_0 = s, A_{0:k-1} \sim \pi_\omega, \tau \sim \gamma_\omega\}. \quad (17.3)$$

Note that, because of the factor of γ^k , this $p(s'|s, \omega)$ is no longer a transition probability and no longer sums to one over all values of s' . (Nevertheless, we continue to use the ‘|’ notation in p .)

The above definition of the transition part of an option model allows us to formulate Bellman equations and dynamic programming algorithms that apply to all options, including low-level actions as a special case. For example, the general Bellman equation for the state values of a hierarchical policy π is

$$v_\pi(s) = \sum_{\omega \in \Omega(s)} \pi(\omega|s) \left[r(s, \omega) + \sum_{s'} p(s'|s, \omega) v_\pi(s') \right], \quad (17.4)$$

where $\Omega(s)$ denotes the set of options available in state s . If $\Omega(s)$ includes only the low-level actions, then this equation reduces to a version of the usual Bellman equation (3.14), except of course γ is included in the new p (17.3) and thus does not appear. Similarly, the corresponding planning algorithms also have no γ . For example, the value iteration algorithm with options, analogous to (4.10), is

$$v_{k+1}(s) \doteq \max_{\omega \in \Omega(s)} \left[r(s, \omega) + \sum_{s'} p(s'|s, \omega) v_k(s') \right], \text{ for all } s \in \mathcal{S}.$$

If $\Omega(s)$ includes all the low-level actions available in each state s , then this algorithm converges to the conventional v_* , from which the optimal policy can be computed. However, it is particularly useful to plan with options when only a subset of the possible options are considered (in $\Omega(s)$) in each state. Value iteration will then converge to the best hierarchical policy limited to the restricted set of options. Although this policy may be sub-optimal, convergence can be much faster because fewer options are considered and because each option can jump over many time steps.

To plan with options, the agent must either be given the option models, or learn them. One natural way to learn an option model is to formulate it as a collection of GVF (as defined in the preceding section) and then learn the GVF using the methods presented in this book. It is not difficult to see how this could be done for the reward part of the option model. You merely choose one GVF's cumulant to be the reward ($C_t = R_t$), its policy to be the option's policy ($\pi = \pi_\omega$), and its termination function to be the discount rate times the option's termination function ($\gamma(s) = \gamma \cdot \gamma_\omega(s)$). The true GVF then equals the reward part of the option model, $v_{\pi, \gamma, C}(s) = r(s, \omega)$, and the learning methods described in this book can be used to approximate it. The state-transition part of the option model is a little more complicated. You need to allocate one GVF for each state that the option might terminate in. We don't want these GVF to accumulate anything except when the option terminates, and then only when termination is in the appropriate state. This can be achieved by choosing the cumulant of the GVF that predicts transition to state s' to be $C_t = (1 - \gamma_\omega(S_t)) \mathbb{1}_{S_t=s'}$. The GVF's policy and termination functions are chosen the same as for the reward part of the option model. The true GVF then equals the s' portion of the option's state-transition model, $v_{\pi, \gamma, C}(s) = p(s'|s, \omega)$, and again this book's methods could be employed to learn it. Although each of these steps is seemingly natural, putting them all together (including function approximation and other essential components) is quite challenging and beyond the current state of the art.

Exercise 17.1 This section has presented options for the discounted case, but discounting is arguably inappropriate for control when using function approximation (Section 10.4). What is the natural Bellman equation for a hierarchical policy, analogous to (17.4), but for the average reward setting (Section 10.3)? What are the two parts of the option model, analogous to (17.2) and (17.3), for the average reward setting? \square

17.3 Observations and State

Throughout this book we have written the learned approximate value functions (and the policies in Chapter 13) as functions of the environment’s state. This is a significant limitation of the methods presented in Part I, in which the learned value function was implemented as a table such that any value function could be exactly approximated; that case is tantamount to assuming that the state of the environment is completely observed by the agent. But in many cases of interest, and certainly in the lives of all natural intelligences, the sensory input gives only partial information about the state of the world. Some objects may be occluded by others, or behind the agent, or miles away. In these cases, potentially important aspects of the environment’s state are not directly observable, and it is a strong, unrealistic, and limiting assumption to assume that the learned value function is implemented as a table over the environment’s state space.

The framework of parametric function approximation that we developed in Part II is far less restrictive and, arguably, no limitation at all. In Part II we retained the assumption that the learned value functions (and policies) are functions of the environment’s state, but allowed these functions to be arbitrarily restricted by the parameterization. It is somewhat surprising and not widely recognized that function approximation includes important aspects of partial observability. For example, if there is a state variable that is not observable, then the parameterization can be chosen such that the approximate value does not depend on that state variable. The effect is just as if the state variable were not observable. Because of this, all the results obtained for the parameterized case apply to partial observability without change. In this sense, the case of parameterized function approximation includes the case of partial observability.

Nevertheless, there are many issues that cannot be investigated without a more explicit treatment of partial observability. Although we cannot give them a full treatment here, we can outline the changes that would be needed to do so. There are four steps.

First, we would change the problem. The environment would emit not its states, but only *observations*—signals that depend on its state but, like a robot’s sensors, provide only partial information about it. For convenience, without loss of generality, we assume that the reward is a direct, known function of the observation (perhaps the observation is a vector, and the reward is one of its components). The environmental interaction would then have no explicit states or rewards, but could simply be an alternating sequence of actions $A_t \in \mathcal{A}$ and observations $O_t \in \mathcal{O}$:

$$A_0, O_1, A_1, O_2, A_2, O_3, A_3, O_4, \dots,$$

going on forever (cf. Equation 3.1) or forming episodes each ending with a special terminal observation.

Second, we can recover the idea of state as used in this book from the sequence of observations and actions. Let us use the word *history*, and the notation H_t , for an initial portion of the trajectory up to an observation: $H_t \doteq A_0, O_1, \dots, A_{t-1}, O_t$. The history represents the most that we can know about the past without looking outside of the data stream (because the history is the whole past data stream). Of course, the history grows with t and can become large and unwieldy. The idea of state is that of a compact summary of the history that is useful for predicting future sequences. To be a summary of the history, a state must be a function of history, $S_t = f(H_t)$. The summary would be informationally perfect if it retained all information about the history (and thus could be used to predict futures as accurately as could be done from the full history). In this case, the state S_t and the function f are said to have the *Markov property*, and S_t is a state as we have used the term in this book. Let us henceforth call it a *Markov state* to distinguish it from states that are summaries of the history but are not sufficient to predict all futures. In practice, the states of real agents will not be Markov but may approach it as an ideal.

To be more explicit about the Markov property it is useful to formalize possible futures. Let a *test* be any specific sequence of alternating actions and observations that might occur in the future. For example, a three-step test might be denoted $\tau = a_1 o_1 a_2 o_2 a_3 o_3$. The probability of this test given a specific history h is defined as

$$p(\tau|h) \doteq \Pr\{O_{t+1}=o_1, O_{t+2}=o_2, O_{t+3}=o_3 \mid H_t=h, A_t=a_1, A_{t+1}=a_2, A_{t+2}=a_3\}. \quad (17.5)$$

Formally, f is Markov if and only if, for any test τ , and for any histories h and h' that map to the same state under f , the test's probabilities given the two histories are equal:

$$f(h) = f(h') \Rightarrow p(\tau|h) = p(\tau|h'), \quad \text{for all } h, h', \tau \in \{\mathcal{A} \times \mathcal{O}\}^*. \quad (17.6)$$

A Markov state summarizes all the information in the history necessary for determining any test's probability. In fact, it summarizes all that is necessary for making *any prediction*, including any GVF. It also summarizes all that is necessary for optimal behavior: if f is Markov, then there is always a deterministic function π such that choosing $A_t \doteq \pi(f(H_t))$ is an optimal policy.

The third step in extending reinforcement learning to partial observability is to deal with certain computational considerations. As mentioned earlier, we want the state to be *compact*—relatively small compared to the history. (The identity function, for example, is not a good f even though it is Markov, because the corresponding state $S_t = H_t$ would grow unboundedly with time.) In addition, we don't really want a function f that takes whole histories. Instead, we want an f that can be compactly implemented with an incremental, recursive update that computes S_{t+1} from S_t , incorporating only the next increment of data, A_t and O_{t+1} :

$$S_{t+1} \doteq u(S_t, A_t, O_{t+1}), \quad \text{for all } t \geq 0, \quad (17.7)$$

with the first state S_0 given. The function u is called the *state-update* function. For example, if f were the identity ($S_t = H_t$), then u would merely extend S_t by appending A_t and O_{t+1} to it. Given f , it is always possible to construct a corresponding u , but it may

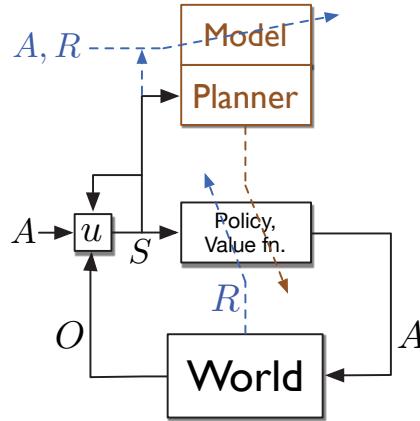


Figure 17.1: A conceptual agent architecture including a model, a planner, and a state-update function. The world in this case receives actions A and emits observations O . The observations and a copy of the action are used by the state-update function u to produce the new state. The new state is input to the policy and value function, producing the next action, and is also input to the planner (and to u). The information flows most responsible for learning are shown by dashed lines that pass diagonally across the boxes that they change. The reward R directly changes the policy and value function. The action, reward, and state change the model, which works closely with the planner to also change the policy and value function. Note that the operation of the planner can be decoupled from the agent-environment interaction, whereas the other processes should operate in lock step with this interaction to keep up with the arrival of new data. Also note that the model and planner do not deal with observations directly, but only with the states produced by u , which can act as targets for model learning.

not be computationally convenient and, as in the identity example, it may not produce a compact state. The state-update function is a central part of any agent architecture that handles partial observability. It must be efficiently computable, as no actions or predictions can be made until the state is available. An overall diagram of such an agent architecture is given in Figure 17.1.

A common strategy for finding a Markov state is to look for something compact that is recursively updatable and enables accurate short-term predictions. In fact, it is only necessary to make accurate *one-step* predictions. An important fact is that, if an f is incrementally updatable, then it is Markov if and only if all one-step tests can be accurately predicted, that is, if and only if

$$f(h) = f(h') \Rightarrow \Pr\{O_{t+1} = o | H_t = h, A_t = a\} = \Pr\{O_{t+1} = o | H_t = h', A_t = a\}, \quad (17.8)$$

for all $h, h' \in \{\mathcal{A} \times \mathcal{O}\}^*$, $o \in \mathcal{O}$ and $a \in \mathcal{A}$. Accurate one-step predictions are informationally sufficient, together with the state-update function, to accurately predict the probability of any test of any length. This can be done by iteratively and alternately making one-step predictions and applying the state-update function. From the whole tree of possibilities the exact probability of any test or the expectation of any GVF can be determined. These observations have led many researchers to focus on one-step predictions rather than directly on multi-step predictions such as GVFs. However, note

that determining long-term predictions from single-step predictions is exponentially complex in the length of the predictions. Moreover, one-step predictions can be iterated to give accurate long-term predictions only if they are exact. If there is any error or approximation in the one-step predictions, then it can compound to make the long-term predictions wildly inaccurate. In practice this is often what happens.

An example of obtaining Markov states through a state-update function is provided by the popular Bayesian approach known as *Partially Observable MDPs*, or *POMDPs*. In this approach the environment is assumed to have a well defined *latent state* X_t that underlies and produces the environment's observations, but is never available to the agent (and is not to be confused with the state S_t used by the agent to make predictions and decisions). The natural Markov state, S_t , for a POMDP is the *distribution* over the latent states given the history, called the *belief state*. For concreteness, assume the usual case in which there are a finite number of hidden states, $X_t \in \{1, 2, \dots, d\}$. Then the belief state is the vector $S_t \doteq \mathbf{s}_t \in [0, 1]^d$ with components

$$\mathbf{s}_t[i] \doteq \Pr\{X_t = i \mid H_t\}, \text{ for all possible latent states } i \in \{1, 2, \dots, d\}. \quad (17.9)$$

The belief state remains the same size (same number of components) even as t grows. It can also be incrementally updated by Bayes' rule, assuming complete knowledge of the internal workings of the environment. Specifically, the i th component of the belief-state update function is

$$u(\mathbf{s}, a, o)[i] \doteq \frac{\sum_{x=1}^d \mathbf{s}[x] p(i, o|x, a)}{\sum_{x=1}^d \sum_{x'=1}^d \mathbf{s}[x] p(x', o|x, a)}, \quad \text{for all } a \in \mathcal{A}, o \in \mathcal{O}, \quad (17.10)$$

and for all belief states \mathbf{s} with components $\mathbf{s}[x]$, where the four-argument p function here is not the usual one for MDPs (as in Chapter 3), but the analogous one for POMDPs, in terms of the *latent state*: $p(x', o|x, a) \doteq \Pr\{X_t = x', O_t = o \mid X_{t-1} = x, A_{t-1} = a\}$. This approach is popular in theoretical work and has many significant applications, but its assumptions and computational complexity scale poorly, and we do not recommend it as an approach to artificial intelligence.

Another example of Markov states is provided by *Predictive State Representations*, or *PSRs*. PSRs address the weakness of the POMDP approach that the semantics of its agent state S_t are grounded in the environment state, X_t , which is never observed and thus is difficult to learn about. In PSRs and related approaches, the semantics of the agent state is instead grounded in predictions about future observations and actions, which are readily observable. In PSRs, a Markov state is defined as a d -vector of the probabilities of d specially chosen "core" tests as defined above (17.5). The vector is then updated by a state-update function u that is analogous to Bayes rule, but with a semantics grounded in observable data, which arguably makes it easier to learn. This approach has been extended in many ways, including end-tests, compositional tests, powerful "spectral" methods, and closed-loop and temporally abstract tests learned by TD methods. Some of the best theoretical developments are for systems known as *Observable Operator Models* (OOMs) and Sequential Systems (Thon, 2017).

The fourth and final step in our brief outline of how to handle partial observability in reinforcement learning is to re-introduce approximation. As discussed in the introduction

to Part II, to approach artificial intelligence ambitiously we must embrace approximation. This is just as true for states as it is for value functions. We must accept and work with an approximate notion of state. The approximate state will play the same role in our algorithms as before, so we continue to use the notation S_t for the state used by the agent, even though it may not be Markov.

Perhaps the simplest example of an approximate state is just the latest observation, $S_t \doteq O_t$. Of course this approach cannot handle any hidden state information. It would be better to use the last k observations and actions, $S_t \doteq O_t, A_{t-1}, O_{t-1}, \dots, A_{t-k}$, for some $k \geq 1$, which can be achieved by a state-update function that just shifts the new data in and the oldest data out. This *kth-order history* approach is still very simple, but can greatly increase the agent's capabilities compared to trying to use the single immediate observation directly as the state.

What happens when the Markov property (17.8) is only approximately satisfied? Unfortunately, long-term prediction performance can degrade dramatically when one-step predictions become even slightly inaccurate. Longer-term tests, GVF_s, and state-update functions may or may not approximate better. The short-term and long-term approximation objectives are just different, and there are no useful theoretical guarantees at present.

Nevertheless, there are still reasons to think that the general idea outlined in this section applies to the approximate case. The general idea is that a state that is good for some predictions is also good for others—in particular, that a Markov state, sufficient for one-step predictions, is also sufficient for all others. If we step back from that specific result for the Markov case, the general idea is similar to what we discussed in Section 17.1 with multi-headed learning and auxiliary tasks. We discussed how representations that were good for the auxiliary tasks were often also good for the main task. Taken together, these suggest an approach to both partial observability and representation learning in which multiple predictions are pursued and used to direct the construction of state features. The guarantee provided by the perfect-but-impractical Markov property is replaced by the heuristic that what's good for some predictions may be good for others. This approach scales well with computational resources. With a powerful computer we could experiment with large numbers of predictions, perhaps favoring those that are most similar to the ones of ultimate interest, that are easiest to learn reliably, or that satisfy other criteria. It is important here to move beyond selecting the predictions manually. The agent should do it. This would require a general language for predictions, so that the agent can systematically explore a large space of possible predictions, sifting through them for the ones that are most useful.

In particular, both POMDP and PSR approaches can be applied with approximate states. The semantics of the state is often useful in forming the state-update function, as it is in these two approaches and in the *kth-order history* approach. However, there is not a strong need for the state to be accurate with respect to its semantics in order to retain useful information. Some approaches to state augmentation, such as Echo state networks (Jaeger, 2002), keep almost arbitrary information about the history and can nevertheless perform well. There are many possibilities, and we expect more work and ideas in this area. Learning the state-update function for an approximate state is a major part of the representation learning problem as it arises in reinforcement learning.

17.4 Designing Reward Signals

A major advantage of reinforcement learning over supervised learning is that reinforcement learning does not rely on detailed instructional information: generating a reward signal does not depend on knowledge of what the agent’s correct actions should be. But the success of a reinforcement learning application strongly depends on how well the reward signal frames the goal of the application’s designer and how well the signal assesses progress in reaching that goal. For these reasons, designing a reward signal is a critical part of any application of reinforcement learning.

By designing a reward signal we mean designing the part of an agent’s environment that is responsible for computing each scalar reward R_t and sending it to the agent at each time t . In our discussion of terminology at the end of Chapter 14, we said that R_t is more like a signal generated inside an animal’s brain than it is like an object or event in the animal’s external environment. The parts of our brains that generate these signals for us evolved over millions of years to be well suited to the challenges our ancestors had to face in their struggles to propagate their genes to future generations. We should therefore not think that designing a good reward signal is always an easy thing to do!

One challenge is to design a reward signal so that as an agent learns, its behavior approaches, and ideally eventually achieves, what the application’s designer actually desires. This can be easy if the designer’s goal is simple and easy to identify, such as finding the solution to a well-defined problem or earning a high score in a well-defined game. In cases like these, it is usual to reward the agent according to its success in solving the problem or its success in improving its score. But some problems involve goals that are difficult to translate into reward signals. This is especially true when the problem requires the agent to skillfully perform a complex task or set of tasks, such as would be required of a useful household robotic assistant. Further, reinforcement learning agents can discover unexpected ways to make their environments deliver reward, some of which might be undesirable, or even dangerous. This is a longstanding and critical challenge for any method, like reinforcement learning, that is based on optimization. We discuss this issue more in Section 17.6, the final section of this book.

Even when there is a simple and easily identifiable goal, the problem of *sparse reward* often arises. Delivering non-zero reward frequently enough to allow the agent to achieve the goal once, let alone to learn to achieve it efficiently from multiple initial conditions, can be a daunting challenge. State-action pairs that clearly deserve to trigger reward may be few and far between, and rewards that mark progress toward a goal can be infrequent because progress is difficult or even impossible to detect. The agent may wander aimlessly for long periods of time (what Minsky, 1961, called the “plateau problem”).

In practice, designing a reward signal is often left to an informal trial-and-error search for a signal that produces acceptable results. If the agent fails to learn, learns too slowly, or learns the wrong thing, then the designer tweaks the reward signal and tries again. To do this, the designer judges the agent’s performance by criteria that he or she is attempting to translate into a reward signal so that the agent’s goal matches his or her own. And if learning is too slow, the designer may try to design a non-sparse reward signal that effectively guides learning throughout the agent’s interaction with its environment.

It is tempting to address the sparse reward problem by rewarding the agent for achieving subgoals that the designer thinks are important way stations to the overall goal. But augmenting the reward signal with well-intentioned supplemental rewards may lead the agent to behave differently from what is intended; the agent may end up not achieving the overall goal. A better way to provide such guidance is to leave the reward signal alone and instead augment the value-function approximation with an initial guess of what it should ultimately be, or augment it with initial guesses as to what certain parts of it should be. For example, suppose we want to offer $v_0 : \mathcal{S} \rightarrow \mathbb{R}$ as an initial guess at the true optimal value function v_* , and that we are using linear function approximation with features $\mathbf{x} : \mathcal{S} \rightarrow \mathbb{R}^d$. Then we would define the initial value function approximation as

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) + v_0(s), \quad (17.11)$$

and update the weights \mathbf{w} as usual. If the initial weight vector is $\mathbf{0}$, then the initial value function will be v_0 , but the asymptotic solution quality will be determined by the feature vectors as usual. This initialization can also be done for arbitrary nonlinear approximators and arbitrary forms of v_0 , though it is not guaranteed to always accelerate learning.

A particularly effective approach to the sparse reward problem is the *shaping* technique introduced by the psychologist B. F. Skinner and described in Section 14.3. The effectiveness of this technique relies on the fact that sparse reward problems are not just problems with the reward signal; they are also problems with an agent’s policy in that it prevents the agent from frequently encountering rewarding states. Shaping involves changing the reward signal as learning proceeds, starting from a reward signal that is not sparse given the agent’s initial behavior, and gradually modifying it toward a reward signal suited to the problem of original interest. Shaping might also involve modifying the dynamics of the task as learning proceeds. Each modification is made so that the agent is frequently rewarded given its current behavior. The agent faces a sequence of increasingly-difficult reinforcement learning problems, where what is learned at each stage makes the next-harder problem relatively easy because the agent now encounters reward more frequently than it would if it did not have prior experience with easier problems. This kind of shaping is an essential technique in training animals, and it is effective in computational reinforcement learning as well.

What if you have no idea what the rewards should be, but there is another agent, perhaps a person, who is already expert at the task and whose behavior can be observed? In this case you could use methods known variously as “imitation learning,” “learning from demonstration,” and “apprenticeship learning.” The idea here is to benefit from the expert agent but leave open the possibility of eventually performing better. Learning from an expert’s behavior can be done either by learning directly by supervised learning or by extracting a reward signal using what is known as “inverse reinforcement learning” and then using a reinforcement learning algorithm with that reward signal to learn a policy. The task of inverse reinforcement learning as explored by Ng and Russell (2000) is to try to recover the expert’s reward signal from the expert’s behavior alone. This cannot be done exactly because a policy can be optimal with respect to many different reward signals (for example, all policies are optimal with respect to a constant reward signal), but it is possible to find plausible reward signal candidates. Unfortunately, strong

assumptions are required, including knowledge of the environment’s dynamics and of the feature vectors in which the reward signal is linear. The method also requires completely solving the problem (e.g., by dynamic programming methods) multiple times. These difficulties notwithstanding, Abbeel and Ng (2004) argue that the inverse reinforcement learning approach can sometimes be more effective than supervised learning for benefiting from the behavior of an expert.

Another approach to finding a good reward signal is to automate the trial-and-error search for a good signal that we mentioned above. From an application perspective, the reward signal is a parameter of the learning algorithm. As is true for other algorithm parameters, the search for a good reward signal can be automated by defining a space of feasible candidates and applying an optimization algorithm. The optimization algorithm evaluates each candidate reward signal by running the reinforcement learning system with that signal for some number of steps, and then scoring the overall result by a “high-level” objective function intended to encode the designer’s true goal, ignoring the limitations of the agent. Reward signals can even be improved via online gradient ascent, where the gradient is that of the high-level objective function (Sorg, Lewis, and Singh, 2010). Relating this approach to the natural world, the algorithm for optimizing the high-level objective function is analogous to evolution, where the high-level objective function is an animal’s evolutionary fitness determined by the number of its offspring that survive to reproductive age.

Computational experiments with this bilevel optimization approach—one level analogous to evolution, and the other due to reinforcement learning by individual agents—have confirmed that intuition alone is not always adequate to devise a good reward signal (Singh, Lewis, and Barto, 2009). The performance of a reinforcement learning agent as evaluated by the high-level objective function can be very sensitive to details of the agent’s reward signal in subtle ways determined by the agent’s limitations and the environment in which it acts and learns. These experiments have also demonstrated that an agent’s goal should not always be the same as the goal of the agent’s designer.

At first this seems counterintuitive, but it may be impossible for the agent to achieve the designer’s goal no matter what its reward signal is. The agent has to learn under various kinds of constraints, such as limited computational power, limited access to information about its environment, or limited time to learn. When there are constraints like these, learning to achieve a goal that is different from the designer’s goal can sometimes end up getting closer to the designer’s goal than if that goal were pursued directly (Sorg, Singh, and Lewis, 2010; Sorg, 2011). Examples of this in the natural world are easy to find. Because we cannot directly assess the nutritional value of most foods, evolution—the designer of our reward signal—gave us a reward signal that makes us seek certain tastes. Though certainly not infallible (indeed, possibly detrimental in environments that differ in certain ways from ancestral environments), this compensates for many of our limitations: our limited sensory abilities, the limited time over which we can learn, and the risks involved in finding a healthy diet through personal experimentation. Similarly, because an animal cannot always observe its own evolutionary fitness, that objective function does not work as a reward signal for learning. Evolution instead provides reward signals that are sensitive to observable predictors of evolutionary fitness.

Finally, remember that a reinforcement learning agent is not necessarily like a complete

organism or robot; it can be a component of a larger behaving system. This means that reward signals may be influenced by things inside the larger behaving agent, such as motivational states, memories, ideas, or even hallucinations. Reward signals may also depend on properties of the learning process itself, such as measures of how much progress learning is making. Making reward signals sensitive to information about internal factors such as these makes it possible for an agent to learn how to control the “cognitive architecture” of which it is a part, as well as to acquire knowledge and skills that would be difficult to learn from a reward signal that depended only on external events. Possibilities like these led to the idea of “intrinsically-motivated reinforcement learning” that we briefly discuss further at the end of the following section.

17.5 Remaining Issues

In this book we have presented the foundations of a reinforcement learning approach to artificial intelligence. Roughly speaking, that approach is based on model-free and model-based methods working together, as in the Dyna architecture of Chapter 8, combined with function approximation as developed in Part II. The focus has been on online and incremental algorithms, which we see as fundamental even to model-based methods, and on how these can be applied in off-policy training situations. The full rationale for the latter has been presented only in this last chapter. That is, we have all along presented off-policy learning as an appealing way to deal with the explore/exploit dilemma, but only in this chapter have we discussed learning about many diverse auxiliary tasks simultaneously with GVF_s and learning about the world hierarchically in terms of temporally-abstract option models, both of which involve off-policy learning. Much remains to be worked out, as we have indicated throughout the book and as evidenced by the directions for additional research discussed in this chapter. But suppose we are generous and grant the broad outlines of everything that we have done in the book *and* everything that has been outlined so far in this chapter. What would remain after that? Of course we can’t know for sure what will be required, but we can make some guesses. In this section we highlight six further issues which it seems to us will still need to be addressed by future research.

First, we still need powerful parametric function approximation methods that work well in fully incremental and online settings. Methods based on deep learning and ANNs are a major step in this direction but, still, only work well with batch training on large data sets, with training from extensive off-line self play, or with learning from the interleaved experience of multiple agents on the same task. These and other settings are ways of working around a basic limitation of today’s deep learning methods, which struggle to learn rapidly in the incremental, online settings that are most natural for the reinforcement learning algorithms emphasized in this book. The problem is sometimes described as one of “catastrophic interference” or “correlated data.” When something new is learned it tends to replace what has previously been learned rather than adding to it, with the result that the benefit of the older learning is lost. Techniques such as “replay buffers” are often used to retain and replay old data so that its benefits are not permanently lost. An honest assessment has to be that current deep learning methods are not well suited to online learning. We see no reason that this limitation is insurmountable, but algorithms

that address it, while at the same time retaining the advantages of deep learning, have not yet been devised. Most current deep learning research is directed toward working around this limitation rather than removing it.

Second (and perhaps closely related), we still need methods for learning features such that subsequent learning generalizes well. This issue is an instance of a general problem variously called “representation learning,” “constructive induction,” and “meta-learning”—how can we use experience not just to learn a given desired function, but to learn inductive biases such that future learning generalizes better and is thus faster? This is an old problem, dating back to the origins of artificial intelligence and pattern recognition in the 1950s and 1960s.¹ Such age should give one pause. Perhaps there is no solution. But it is equally likely that the time for finding a solution and demonstrating its effectiveness has not yet arrived. Today machine learning is conducted at a far larger scale than it has been in the past, and the potential benefits of a good representation learning method have become much more apparent. We note that a new annual conference—the International Conference on Learning Representations—has been exploring this and related topics every year since 2013. It is also less common to explore representation learning within a reinforcement learning context. Reinforcement learning brings some new possibilities to this old issue, such as the auxiliary tasks discussed in Section 17.1. In reinforcement learning, the problem of representation learning can be identified with the problem of learning the state-update function discussed in Section 17.3.

Third, we still need scalable methods for planning with learned environment models. Planning methods have proven extremely effective in applications such as AlphaGo Zero and computer chess in which the model of the environment is known from the rules of the game or can otherwise be supplied by human designers. But cases of full model-based reinforcement learning, in which the environment model is learned from data and then used for planning, are rare. The Dyna system described in Chapter 8 is one example, but as described there and in most subsequent work it uses a tabular model without function approximation, which greatly limits its applicability. Only a few studies have included learned linear models, and even fewer have also explored the inclusion of temporally-abstract models using options as discussed in Section 17.2.

More work is needed before planning with learned models can be effective. For example, the learning of the model needs to be selective because the scope of a model strongly affects planning efficiency. If a model focuses on the key consequences of the most important options, then planning can be efficient and rapid, but if a model includes details of unimportant consequences of options that are unlikely to be selected, then planning may be almost useless. Environment models should be constructed judiciously with regard to both their states and dynamics with the goal of optimizing the planning process. The various parts of the model should be continually monitored as to the degree to which they contribute to, or detract from, planning efficiency. The field has not yet addressed this complex of issues or designed model-learning methods that take into account their implications.

¹Some would claim that deep learning solves this problem, for example, that DQN as described in Section 16.5 illustrates a solution, but we are unconvinced. There is as yet little evidence that deep learning alone solves the representation learning problem in a general and efficient way.

A fourth issue that needs to be addressed in future research is that of automating the choice of tasks on which an agent works and uses to structure its developing competence. It is usual in machine learning for human designers to set the tasks that the learning agent is expected to master. Because these tasks are known in advance and remain fixed, they can be built into the learning algorithm code. However, looking ahead, we will want the agent to make its own choices about what tasks it should try to master. These might be subtasks of a specific overall task that is already known, or they might be intended to create building blocks that permit more efficient learning of many different tasks that the agent is likely to face in the future but which are currently unknown.

These tasks may be like the auxiliary tasks or the GVF_s discussed in Section 17.1, or tasks solved by options as discussed in Section 17.2. In forming a GVF, for example, what should the cumulant, the policy, and the termination function be? The current state of the art is to select these manually, but far greater power and generality would come from making these task choices automatically, particularly when they derive from what the agent has previously constructed as a result of representation learning or experience with previous subproblems. If GVF design is automated, then the design choices themselves will have to be explicitly represented. Rather than the task choices being in the mind of the designer and built into the code, they will have to be in the machine itself in such a way that they can be set and changed, monitored, filtered, and searched among automatically. Tasks could then be built hierarchically upon others much like features are in an ANN. The tasks are the questions, and the contents of the ANN are the answers to those questions. We expect there will need to be a full hierarchy of questions to match the hierarchy of answers provided by modern deep learning methods.

The fifth issue that we would like to highlight for future research is that of the interaction between behavior and learning via some computational analog of *curiosity*. In this chapter we have been imagining a setting in which many tasks are being learned simultaneously, using off-policy methods, from the same stream of experience. The actions taken will of course influence this stream of experience, which in turn will determine how much learning occurs and which tasks are learned. When reward is not available, or not strongly influenced by behavior, the agent is free to choose actions that maximize in some sense the learning on the tasks, that is, to use some measure of learning progress as an internal or “intrinsic” reward, implementing a computational form of curiosity. In addition to measuring learning progress, intrinsic reward can, among other possibilities, signal the receipt of unexpected, novel, or otherwise interesting input, or can assess the agent’s ability to cause changes in its environment. Intrinsic reward signals generated in these ways can be used by an agent to pose tasks for itself by defining auxiliary tasks, GVFs, or options, as discussed above, so that skills learned in this way can contribute to the agent’s ability to master future tasks. The result is a computational analog of something like *play*. Many preliminary studies of such uses of intrinsic reward signals have been conducted, and exciting topics for future research remain in this general area.

A final issue that demands attention in future research is that of developing methods to make it acceptably safe to embed reinforcement learning agents into physical environments. This is one of the most pressing areas for future research, and we discuss it further in the following section.

17.6 Reinforcement Learning and the Future of Artificial Intelligence

When we were writing the first edition of this book in the mid-1990s, artificial intelligence was making significant progress and was having an impact on society, though it was mostly still the *promise* of artificial intelligence that was inspiring developments. Machine learning was part of that outlook, but it had not yet become indispensable to artificial intelligence. By today that promise has transitioned to applications that are changing the lives of millions of people, and machine learning has come into its own as a key technology. As we write this second edition, some of the most remarkable developments in artificial intelligence have involved reinforcement learning, most notably “deep reinforcement learning”—reinforcement learning with function approximation by deep artificial neural networks. We are at the beginning of a wave of real-world applications of artificial intelligence, many of which will include reinforcement learning, deep and otherwise, that will impact our lives in ways that are hard to predict.

But an abundance of successful real-world applications does not mean that true artificial intelligence has arrived. Despite great progress in many areas, the gulf between artificial intelligence and the intelligence of humans, and other animals, remains great. Superhuman performance can be achieved in some domains, even formidable domains like Go, but it remains a significant challenge to develop systems that are like us in being complete, interactive agents having general adaptability and problem-solving skills, emotional sophistication, creativity, and the ability to learn quickly from experience. With its focus on learning by interacting with dynamic environments, reinforcement learning, as it develops over the future, will be a critical component of agents with these abilities.

Reinforcement learning’s connections to psychology and neuroscience (Chapters 14 and 15) underscore its relevance to another longstanding goal of artificial intelligence: shedding light on fundamental questions about the mind and how it emerges from the brain. Reinforcement learning theory is already contributing to our understanding of the brain’s reward, motivation, and decision-making processes, and there is good reason to believe that through its links to computational psychiatry, reinforcement learning theory will contribute to methods for treating mental disorders, including drug abuse and addiction.

Another contribution that reinforcement learning can make over the future is as an aid to human decision making. Policies derived by reinforcement learning in simulated environments can advise human decision makers in such areas as education, healthcare, transportation, energy, and public-sector resource allocation. Particularly relevant is the key feature of reinforcement learning that it takes long-term consequences of decisions into account. This is very clear in games like backgammon and Go, where some of the most impressive results of reinforcement learning have been demonstrated, but it is also a property of many high-stakes decisions that affect our lives and our planet. Reinforcement learning follows related methods for advising human decision making that have been developed in the past by decision analysts in many disciplines. With advanced function approximation methods and massive computational power, reinforcement learning

methods have the potential to overcome some of the difficulties of scaling up traditional decision-support methods to larger and more complex problems.

The rapid pace of advances in artificial intelligence has led to warnings that artificial intelligence poses serious threats to our societies, even to humanity itself. The renowned scientist and artificial intelligence pioneer Herbert Simon anticipated the warnings we are hearing today in a presentation at the Earthware Symposium at CMU in 2000 (Simon, 2000). He spoke of the eternal conflict between the promise and perils of any new knowledge, reminding us of the Greek myths of Prometheus, the idealized hero of modern science, who stole fire from the gods for the benefit of mankind, and of Pandora, whose mythical box could be opened by a small and innocent action to release untold perils on the world. While accepting that this conflict is inevitable, Simon urged us to recognize that as designers of our future and not mere spectators, the decisions *we* make can tilt the scale in Prometheus' favor. This is certainly true for reinforcement learning, which can benefit society but can also produce undesirable outcomes if it is carelessly deployed. Thus, the *safety* of artificial intelligence applications involving reinforcement learning is a topic that deserves careful attention.

A reinforcement learning agent can learn by interacting with either the real world or with a simulation of some piece of the real world, or by a mixture of these two sources of experience. Simulators provide safe environments in which an agent can explore and learn without risking real damage to itself or to its environment. In most current applications, policies are learned from simulated experience instead of direct interaction with the real world. In addition to avoiding undesirable real-world consequences, learning from simulated experience can make virtually unlimited data available for learning, generally at less cost than needed to obtain real experience, and because simulations typically run much faster than real time, learning can often occur more quickly than if it relied on real experience.

Nevertheless, the full potential of reinforcement learning requires reinforcement learning agents to be embedded into the flow of real-world experience, where they act, explore, and learn in *our* world, and not just in *their* worlds. After all, reinforcement learning algorithms—at least those upon which we focus in this book—are designed to learn online, and they emulate many aspects of how animals are able to survive in nonstationary and hostile environments. Embedding reinforcement learning agents in the real world can be transformative in realizing the promises of artificial intelligence to amplify and extend human abilities.

A major reason for wanting a reinforcement learning agent to act and learn in the real world is that it is often difficult, sometimes impossible, to simulate real-world experience with enough fidelity to make the resulting policies, whether derived by reinforcement learning or by other methods, work well—and safely—when directing real actions. This is especially true for environments whose dynamics depend on the behavior of humans, such as in education, healthcare, transportation, and public policy—domains that can surely benefit from improved decision making. However, it is for real-world embedded agents that warnings about potential dangers of artificial intelligence need to be heeded.

Some of these warnings are particularly relevant to reinforcement learning. Because reinforcement learning is based on optimization, it inherits the plusses and minuses of all optimization methods. On the minus side is the problem of devising objective functions,

or reward signals in the case of reinforcement learning, so that optimization produces the desired results while avoiding undesirable results. We said in Section 17.4 that reinforcement learning agents can discover unexpected ways to make their environments deliver reward, some of which might be undesirable, or even dangerous. When we specify what we want a system to learn only indirectly, as we do in designing a reinforcement learning system’s reward signal, we will not know how closely the agent will fulfill our desire until its learning is complete. This is hardly a new problem with reinforcement learning; recognition of it has a long history in both literature and engineering. For example, in Goethe’s poem “The Sorcerer’s Apprentice” (Goethe, 1878), the apprentice uses magic to enchant a broom to do his job of fetching water, but the result is an unintended flood due to the apprentice’s inadequate knowledge of magic. In the engineering context, Norbert Wiener, the founder of cybernetics, warned of this problem more than half a century ago by relating the supernatural story of “The Monkey’s Paw” (Wiener, 1964): “... it grants what you ask for, not what you should have asked for or what you intend” (p. 59). The problem has also been discussed at length in a modern context by Nick Bostrom (2014). Anyone having experience with reinforcement learning has likely seen their systems discover unexpected ways to obtain a lot of reward. Sometimes the unexpected behavior is good: it solves a problem in a nice new way. In other instances, what the agent learns violates considerations that the system designer may never have thought about. Careful design of reward signals is essential if an agent is to act in the real world with no opportunity for human vetting of its actions or means to easily interrupt its behavior.

Despite the possibility of unintended negative consequences, optimization has been used for hundreds of years by engineers, architects, and others whose designs have positively impacted the world. We owe much that is good in our environment to the application of optimization methods. Many approaches have been developed to mitigate the risk of optimization, such as adding hard and soft constraints, restricting optimization to robust and risk-sensitive policies, and optimizing with multiple objective functions. Some of these approaches have been adapted to reinforcement learning, and more research is needed to address these concerns. The problem of ensuring that a reinforcement learning agent’s goal is attuned to our own remains a challenge.

Another challenge if reinforcement learning agents are to act and learn in the real world is not just about what they might learn *eventually*, but about how they will behave while they are learning. How do you make sure that an agent gets enough experience to learn a high-performing policy, all the while not harming its environment, other agents, or itself (or more realistically, while keeping the probability of harm acceptably low)? This problem is also not novel or unique to reinforcement learning. Risk management and mitigation for embedded reinforcement learning is similar to what control engineers have had to confront from the beginning of using automatic control in situations where a controller’s behavior can have unacceptable, possibly catastrophic, consequences, as in the control of an aircraft or a delicate chemical process. Control applications rely on careful system modeling, model validation, and extensive testing, and there is a highly-developed body of theory aimed at ensuring convergence and stability of adaptive controllers designed for use when the dynamics of the system to be controlled are not fully known. Theoretical guarantees are never iron-clad because they depend on the validity of the assumptions underlying the mathematics, but without this theory, combined with risk-management

and mitigation practices, automatic control—adaptive and otherwise—would not be as beneficial as it is today in improving the quality, efficiency, and cost-effectiveness of processes on which we have come to rely. One of the most pressing areas for future reinforcement learning research is to adapt and extend methods developed in control engineering with the goal of making it acceptably safe to fully embed reinforcement learning into physical environments.

In closing, we return to Simon’s call for us to recognize that we are designers of our future and not simply spectators. By decisions we make as individuals, and by the influence we can exert on how our societies are governed, we can work toward ensuring that the benefits made possible by a new technology outweigh the harm it can cause. There is ample opportunity to do this in the case of reinforcement learning, which can help improve the quality, fairness, and sustainability of life on our planet, but which can also release new perils. A threat already here is the displacement of jobs caused by applications of artificial intelligence. Still there are good reasons to believe that the benefits of artificial intelligence can outweigh the disruption it causes. As to safety, hazards possible with reinforcement learning are not completely different from those that have been managed successfully for related applications of optimization and control methods. As reinforcement learning moves out into the real world in future applications, developers have an obligation to follow best practices that have evolved for similar technologies, while at the same time extending them to make sure that Prometheus keeps the upper hand.

Bibliographical and Historical Remarks

17.1 General value functions were first explicitly identified by Sutton and colleagues (Sutton, 1995a; Sutton et al., 2011; Modayil, White, and Sutton, 2013). Ring (in preparation) developed an extensive thought experiment with GVF (“forecasts”) that has been influential despite not yet having been published.

The first demonstrations of multi-headed learning in reinforcement learning were by Jaderberg et al. (2017). Bellemare, Dabney, and Munos (2017) showed that predicting more things about the distribution of reward could significantly accelerate learning to optimize its expectation, an instance of auxiliary tasks. Many others have since taken up this line of research.

The general theory of classical conditioning as learned predictions together with built-in, reflexive reactions to the predictions has not to our knowledge been clearly articulated in the psychological literature. Modayil and Sutton (2014) describe it as an approach to the engineering of robots and other agents, calling it “Pavlovian control” to allude to its roots in classical conditioning.

17.2 The formalization of temporally abstract courses of action as options was introduced by Sutton, Precup, and Singh (1999), building on prior work by Parr (1998) and Sutton (1995a), and on classical work on Semi-MDPs (e.g., see Puterman, 1994). Precup’s (2000) PhD thesis developed option ideas fully. An important limitation of these early works is that they did not treat the off-policy case

with function approximation. Intra-option learning in general requires off-policy learning, which could not be done reliably with function approximation at that time. Although now we have a variety of stable off-policy learning methods using function approximation, their combination with option ideas had not been significantly explored at the time of publication of this book. Barto and Mahadevan (2003) and Hengst (2012) review the options formalism and other approaches to temporal abstraction.

Using GVF_s to implement option models has not previously been described. Our presentation uses the trick introduced by Modyal, White, and Sutton (2014) for predicting signals at the termination of policies.

Among the few works that have learned option models with function approximation are those by Sorg and Singh (2010), and by Bacon, Harb, and Precup (2017).

The extension of options and option models to the average-reward setting has not yet been developed in the literature.

- 17.3** A good presentation of the POMDP approach is given by Monahan (1982). PSRs and tests were introduced by Littman, Sutton, and Singh (2002). OOMs were introduced by Jaeger (1997, 1998, 2000). Sequential Systems, which unify PSRs, OOMs, and many other works, were introduced in the PhD thesis of Michael Thon (2017; Thon and Jaeger, 2015). Extensions to networks of temporal relationships were developed by Tanner (2006; Sutton and Tanner, 2005) and then extended to options (Sutton, Rafols, and Koop, 2006).

The theory of reinforcement learning with a non-Markov state representation was developed explicitly by Singh, Jaakkola, and Jordan (1994; Jaakkola, Singh, and Jordan, 1995). Early reinforcement learning approaches to partial observability were developed by Chrisman (1992), McCallum (1993, 1995), Parr and Russell (1995), Littman, Cassandra, and Kaelbling (1995), and by Lin and Mitchell (1992).

- 17.4** Early efforts to include advice and teaching in reinforcement learning include those by Lin (1992), Maclin and Shavlik (1994), Clouse (1996), and Clouse and Utgoff (1992).

Skinner's shaping should not be confused with the "potential-based shaping" technique introduced by Ng, Harada, and Russell (1999). Their technique has been shown by Wiewiora (2003) to be equivalent to the simpler idea of providing an initial approximation to the value function, as in (17.11).

- 17.5** We recommend the book by Goodfellow, Bengio, and Courville (2016) for discussion of today's deep learning techniques. The problem of catastrophic interference in ANNs was developed by McCloskey and Cohen (1989), Ratcliff (1990), and French (1999). The idea of a replay buffer was introduced by Lin (1992) and used prominently in deep learning in the Atari game playing system (Section 16.5, Mnih et al., 2013, 2015).

Minsky (1961) was one of the first to identify the problem of representation learning.

Among the few works to consider planning with learned, approximate models are those by Kuvayev and Sutton (1996), Sutton, Szepesvari, Geramifard, and Bowling (2008), Nouri and Littman (2009), and Hester and Stone (2012).

The need to be selective in model construction to avoid slowing planning is well known in artificial intelligence. Some of the classic work is by Minton (1990) and Tambe, Newell, and Rosenbloom (1990). Hauskrecht, Meuleau, Kaelbling, Dean, and Boutilier (1998) showed this effect in MDPs with deterministic options.

Schmidhuber (1991a, b) proposed how something like curiosity would result if reward signals were a function of how quickly an agent's environment model is improving. The empowerment function proposed by Klyubin, Polani, and Nehaniv (2005) is an information-theoretic measure of an agent's ability to control its environment that can function as an intrinsic reward signal. Baldassarre and Mirolli (2013) is a collection of contributions by researchers studying intrinsic reward and motivation from both biological and computational perspectives, including a perspective on "intrinsically-motivated reinforcement learning," to use the term introduced by Singh, Barto, and Chentenez (2004). See also Oudeyer and Kaplan (2007), Oudeyer, Kaplan, and Hafner (2007), and Barto (2013).

References

- Abbeel, P., Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*. ACM, New York.
- Abramson, B. (1990). Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):182–193.
- Adams, C. D. (1982). Variations in the sensitivity of instrumental responding to reinforcer devaluation. *The Quarterly Journal of Experimental Psychology*, 34(2):77–98.
- Adams, C. D., Dickinson, A. (1981). Instrumental responding following reinforcer devaluation. *The Quarterly Journal of Experimental Psychology*, 33(2):109–121.
- Adams, R. A., Huys, Q. J. M., Roiser, J. P. (2015). Computational Psychiatry: towards a mathematically informed understanding of mental illness. *Journal of Neurology, Neurosurgery & Psychiatry*. doi:10.1136/jnnp-2015-310737
- Agrawal, R. (1995). Sample mean based index policies with $O(\log n)$ regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078.
- Agre, P. E. (1988). *The Dynamic Structure of Everyday Life*. PhD thesis, Massachusetts Institute of Technology, Cambridge MA. AI-TR 1085, MIT Artificial Intelligence Laboratory.
- Agre, P. E., Chapman, D. (1990). What are plans for? *Robotics and Autonomous Systems*, 6(1-2):17–34.
- Aizerman, M. A., Braverman, E. I., Rozonoer, L. I. (1964). Probability problem of pattern recognition learning and potential functions method. *Avtomat. i Telemekh*, 25(9):1307–1323.
- Albus, J. S. (1971). A theory of cerebellar function. *Mathematical Biosciences*, 10(1-2):25–61.
- Albus, J. S. (1981). *Brain, Behavior, and Robotics*. Byte Books, Peterborough, NH.
- Aleksandrov, V. M., Sysoev, V. I., Shemeneva, V. V. (1968). Stochastic optimization of systems. *Izv. Akad. Nauk SSSR, Tekh. Kibernetika*:14–19.
- Amari, S. I. (1998). Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276.
- An, P. C. E. (1991). *An Improved Multi-dimensional CMAC Neural network: Receptive Field Function and Placement*. PhD thesis, University of New Hampshire, Durham.
- An, P. C. E., Miller, W. T., Parks, P. C. (1991). Design improvements in associative memories for cerebellar model articulation controllers (CMAC). *Artificial Neural Networks*, pp. 1207–1210, Elsevier North-Holland. <http://www.incompleteideas.net/papers/AnMillerParks1991.pdf>
- Anderson, C. W. (1986). *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, University of Massachusetts, Amherst.
- Anderson, C. W. (1987). Strategy learning with multilayer connectionist representations. In *Proceedings of the 4th International Workshop on Machine Learning*, pp. 103–114. Morgan Kaufmann.

- Anderson, C. W. (1989). Learning to control an inverted pendulum using neural networks. *IEEE Control Systems Magazine*, 9(3):31–37.
- Anderson, J. A., Silverstein, J. W., Ritz, S. A., Jones, R. S. (1977). Distinctive features, categorical perception, and probability learning: Some applications of a neural model. *Psychological Review*, 84(5):413–451.
- Andreae, J. H. (1963). STELLA, A scheme for a learning machine. In *Proceedings of the 2nd IFAC Congress, Basle*, pp. 497–502. Butterworths, London.
- Andreae, J. H. (1969). Learning machines—a unified view. In A. R. Meetham and R. A. Hudson (Eds.), *Encyclopedia of Information, Linguistics, and Control*, pp. 261–270. Pergamon, Oxford.
- Andreae, J. H. (1977). *Thinking with the Teachable Machine*. Academic Press, London.
- Andreae, J. H. (2017a). A model of how the brain learns: A short introduction to multiple context associative learning (MCAL) and the PP system. Unpublished report.
- Andreae, J. H. (2017b). Working memory for the associative learning of language. Unpublished report.
- Andreae, J. H., Cashin, P. M. (1969). A learning machine with monologue. *International Journal of Man-Machine Studies*, 1(1):1–20.
- Arthur, W. B. (1991). Designing economic agents that act like human agents: A behavioral approach to bounded rationality. *The American Economic Review*, 81(2):353–359.
- Asadi, K., Allen, C., Roderick, M., Mohamed, A. R., Konidaris, G., Littman, M. (2017). Mean actor critic. ArXiv:1709.00503.
- Atkeson, C. G. (1992). Memory-based approaches to approximating continuous functions. In *Sante Fe Institute Studies in the Sciences of Complexity*, Proceedings Vol. 12, pp. 521–521. Addison-Wesley.
- Atkeson, C. G., Moore, A. W., Schaal, S. (1997). Locally weighted learning. *Artificial Intelligence Review*, 11:11–73.
- Auer, P., Cesa-Bianchi, N., Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.
- Bacon, P. L., Harb, J., Precup, D. (2017). The option-critic architecture. In *Proceedings of the Association for the Advancement of Artificial Intelligence*, pp. 1726–1734.
- Baird, L. C. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Machine Learning*, pp. 30–37. Morgan Kaufmann.
- Baird, L. C. (1999). *Reinforcement Learning through Gradient Descent*. PhD thesis, Carnegie Mellon University, Pittsburgh PA.
- Baird, L. C., Klopf, A. H. (1993). Reinforcement learning with high-dimensional, continuous actions. Wright Laboratory, Wright-Patterson Air Force Base, Tech. Rep. WL-TR-93-1147.
- Baird, L., Moore, A. W. (1999). Gradient descent for general reinforcement learning. In *Advances in Neural Information Processing Systems 11*, pp. 968–974. MIT Press, Cambridge MA.
- Baldassarre, G., Mirolli, M. (Eds.) (2013). *Intrinsically Motivated Learning in Natural and Artificial Systems*. Springer-Verlag, Berlin Heidelberg.
- Balke, A., Pearl, J. (1994). Counterfactual probabilities: Computational methods, bounds and applications. In *Proceedings of the Tenth International Conference on Uncertainty in Artificial Intelligence*, pp. 46–54. Morgan Kaufmann.
- Baras, D., Meir, R. (2007). Reinforcement learning, spike-time-dependent plasticity, and the BCM rule. *Neural Computation*, 19(8):2245–2279.

- Barnard, E. (1993). Temporal-difference methods and Markov models. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(2):357–365.
- Barreto, A. S., Precup, D., Pineau, J. (2011). Reinforcement learning using kernel-based stochastic factorization. In *Advances in Neural Information Processing Systems 24*, pp. 720–728. Curran Associates, Inc.
- Bartlett, P. L., Baxter, J. (1999). Hebbian synaptic modifications in spiking neurons that learn. Technical report, Research School of Information Sciences and Engineering, Australian National University.
- Bartlett, P. L., Baxter, J. (2000). A biologically plausible and locally optimal learning algorithm for spiking neurons. Rapport technique, Australian National University.
- Barto, A. G. (1985). Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology*, 4(4):229–256.
- Barto, A. G. (1986). Game-theoretic cooperativity in networks of self-interested units. In J. S. Denker (Ed.), *Neural Networks for Computing*, pp. 41–46. American Institute of Physics, New York.
- Barto, A. G. (1989). From chemotaxis to cooperativity: Abstract exercises in neuronal learning strategies. In R. Durbin, R. Maill and G. Mitchison (Eds.), *The Computing Neuron*, pp. 73–98. Addison-Wesley, Reading, MA.
- Barto, A. G. (1990). Connectionist learning for control: An overview. In T. Miller, R. S. Sutton, and P. J. Werbos (Eds.), *Neural Networks for Control*, pp. 5–58. MIT Press, Cambridge, MA.
- Barto, A. G. (1991). Some learning tasks from a control perspective. In L. Nadel and D. L. Stein (Eds.), *1990 Lectures in Complex Systems*, pp. 195–223. Addison-Wesley, Redwood City, CA.
- Barto, A. G. (1992). Reinforcement learning and adaptive critic methods. In D. A. White and D. A. Sofge (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pp. 469–491. Van Nostrand Reinhold, New York.
- Barto, A. G. (1995a). Adaptive critics and the basal ganglia. In J. C. Houk, J. L. Davis, and D. G. Beiser (Eds.), *Models of Information Processing in the Basal Ganglia*, pp. 215–232. MIT Press, Cambridge, MA.
- Barto, A. G. (1995b). Reinforcement learning. In M. A. Arbib (Ed.), *Handbook of Brain Theory and Neural Networks*, pp. 804–809. MIT Press, Cambridge, MA.
- Barto, A. G. (2011). Adaptive real-time dynamic programming. In C. Sammut and G. I Webb (Eds.), *Encyclopedia of Machine Learning*, pp. 19–22. Springer Science and Business Media.
- Barto, A. G. (2013). Intrinsic motivation and reinforcement learning. In G. Baldassarre and M. Mirolli (Eds.), *Intrinsically Motivated Learning in Natural and Artificial Systems*, pp. 17–47. Springer-Verlag, Berlin Heidelberg.
- Barto, A. G., Anandan, P. (1985). Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15(3):360–375.
- Barto, A. G., Anderson, C. W. (1985). Structural learning in connectionist systems. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pp. 43–54.
- Barto, A. G., Anderson, C. W., Sutton, R. S. (1982). Synthesis of nonlinear control surfaces by a layered associative search network. *Biological Cybernetics*, 43(3):175–185.
- Barto, A. G., Bradtke, S. J., Singh, S. P. (1991). Real-time learning and control using asynchronous dynamic programming. Technical Report 91-57. Department of Computer and Information Science, University of Massachusetts, Amherst.
- Barto, A. G., Bradtke, S. J., Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2):81–138.

- Barto, A. G., Duff, M. (1994). Monte Carlo matrix inversion and reinforcement learning. In *Advances in Neural Information Processing Systems 6*, pp. 687–694. Morgan Kaufmann, San Francisco.
- Barto, A. G., Jordan, M. I. (1987). Gradient following without back-propagation in layered networks. In M. Caudill and C. Butler (Eds.), *Proceedings of the IEEE First Annual Conference on Neural Networks*, pp. II629–II636. SOS Printing, San Diego.
- Barto, A. G., Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379.
- Barto, A. G., Singh, S. P. (1990). On the computational economics of reinforcement learning. In *Connectionist Models: Proceedings of the 1990 Summer School*. Morgan Kaufmann.
- Barto, A. G., Sutton, R. S. (1981a). Goal seeking components for adaptive intelligence: An initial assessment. Technical Report AFWAL-TR-81-1070. Air Force Wright Aeronautical Laboratories/Avionics Laboratory, Wright-Patterson AFB, OH.
- Barto, A. G., Sutton, R. S. (1981b). Landmark learning: An illustration of associative search. *Biological Cybernetics*, 42(1):1–8.
- Barto, A. G., Sutton, R. S. (1982). Simulation of anticipatory responses in classical conditioning by a neuron-like adaptive element. *Behavioural Brain Research*, 4(3):221–235.
- Barto, A. G., Sutton, R. S., Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):835–846. Reprinted in J. A. Anderson and E. Rosenfeld (Eds.), *Neurocomputing: Foundations of Research*, pp. 535–549. MIT Press, Cambridge, MA, 1988.
- Barto, A. G., Sutton, R. S., Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 40(3):201–211.
- Barto, A. G., Sutton, R. S., Watkins, C. J. C. H. (1990). Learning and sequential decision making. In M. Gabriel and J. Moore (Eds.), *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pp. 539–602. MIT Press, Cambridge, MA.
- Baxter, J., Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350.
- Baxter, J., Bartlett, P. L., Weaver, L. (2001). Experiments with infinite-horizon, policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:351–381.
- Bellemare, M. G., Dabney, W., Munos, R. (2017). A distributional perspective on reinforcement learning. ArXiv:1707.06887.
- Bellemare, M. G., Naddaf, Y., Veness, J., Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.
- Bellemare, M. G., Veness, J., Bowling, M. (2012). Investigating contingency awareness using Atari 2600 games. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pp. 864–871. AAAI Press, Menlo Park, CA.
- Bellman, R. E. (1956). A problem in the sequential design of experiments. *Sankhya*, 16:221–229.
- Bellman, R. E. (1957a). *Dynamic Programming*. Princeton University Press, Princeton.
- Bellman, R. E. (1957b). A Markov decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684.
- Bellman, R. E., Dreyfus, S. E. (1959). Functional approximations and dynamic programming. *Mathematical Tables and Other Aids to Computation*, 13:247–251.
- Bellman, R. E., Kalaba, R., Kotkin, B. (1963). Polynomial approximation—A new computational technique in dynamic programming: Allocation processes. *Mathematical Computation*, 17:155–161.

- Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–27.
- Bengio, Y., Courville, A. C., Vincent, P. (2012). Unsupervised feature learning and deep learning: A review and new perspectives. *CoRR* 1, ArXiv:1206.5538.
- Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.
- Berg, H. C. (1975). Chemotaxis in bacteria. *Annual review of biophysics and bioengineering*, 4(1):119–136.
- Berns, G. S., McClure, S. M., Pagnoni, G., Montague, P. R. (2001). Predictability modulates human brain response to reward. *The journal of neuroscience*, 21(8):2793–2798.
- Berridge, K. C., Kringelbach, M. L. (2008). Affective neuroscience of pleasure: reward in humans and animals. *Psychopharmacology*, 199(3):457–480.
- Berridge, K. C., Robinson, T. E. (1998). What is the role of dopamine in reward: hedonic impact, reward learning, or incentive salience? *Brain Research Reviews*, 28(3):309–369.
- Berry, D. A., Fristedt, B. (1985). *Bandit Problems*. Chapman and Hall, London.
- Bertsekas, D. P. (1982). Distributed dynamic programming. *IEEE Transactions on Automatic Control*, 27(3):610–616.
- Bertsekas, D. P. (1983). Distributed asynchronous computation of fixed points. *Mathematical Programming*, 27(1):107–120.
- Bertsekas, D. P. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Englewood Cliffs, NJ.
- Bertsekas, D. P. (2005). *Dynamic Programming and Optimal Control, Volume 1*, third edition. Athena Scientific, Belmont, MA.
- Bertsekas, D. P. (2012). *Dynamic Programming and Optimal Control, Volume 2: Approximate Dynamic Programming*, fourth edition. Athena Scientific, Belmont, MA.
- Bertsekas, D. P. (2013). Rollout algorithms for discrete optimization: A survey. In *Handbook of Combinatorial Optimization*, pp. 2989–3013. Springer, New York.
- Bertsekas, D. P., Tsitsiklis, J. N. (1989). *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ.
- Bertsekas, D. P., Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.
- Bertsekas, D. P., Tsitsiklis, J. N., Wu, C. (1997). Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, 3(3):245–262.
- Bertsekas, D. P., Yu, H. (2009). Projected equation methods for approximate solution of large linear systems. *Journal of Computational and Applied Mathematics*, 227(1):27–50.
- Bhat, N., Farias, V., Moallemi, C. C. (2012). Non-parametric approximate dynamic programming via the kernel method. In *Advances in Neural Information Processing Systems 25*, pp. 386–394. Curran Associates, Inc.
- Bhatnagar, S., Sutton, R., Ghavamzadeh, M., Lee, M. (2009). Natural actor–critic algorithms. *Automatica*, 45(11).
- Biermann, A. W., Fairfield, J. R. C., Beres, T. R. (1982). Signature table systems and learning. *IEEE Transactions on Systems, Man, and Cybernetics*, 12(5):635–648.
- Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Clarendon, Oxford.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer Science + Business Media New York LLC.
- Blodgett, H. C. (1929). The effect of the introduction of reward upon the maze performance of rats. *University of California Publications in Psychology*, 4:113–134.

- Boakes, R. A., Costa, D. S. J. (2014). Temporal contiguity in associative learning: Interference and decay from an historical perspective. *Journal of Experimental Psychology: Animal Learning and Cognition*, 40(4):381–400.
- Booker, L. B. (1982). *Intelligent Behavior as an Adaptation to the Task Environment*. PhD thesis, University of Michigan, Ann Arbor.
- Bostrom, N. (2014). *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press.
- Bottou, L., Vapnik, V. (1992). Local learning algorithms. *Neural Computation*, 4(6):888–900.
- Boyan, J. A. (1999). Least-squares temporal difference learning. In *Proceedings of the 16th International Conference on Machine Learning*, pp. 49–56.
- Boyan, J. A. (2002). Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2):233–246.
- Boyan, J. A., Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*, pp. 369–376. MIT Press, Cambridge, MA.
- Bradtko, S. J. (1993). Reinforcement learning applied to linear quadratic regulation. In *Advances in Neural Information Processing Systems 5*, pp. 295–302. Morgan Kaufmann.
- Bradtko, S. J. (1994). *Incremental Dynamic Programming for On-Line Adaptive Optimal Control*. PhD thesis, University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 94-62.
- Bradtko, S. J., Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, 22:33–57.
- Bradtko, S. J., Ydstie, B. E., Barto, A. G. (1994). Adaptive linear quadratic control using policy iteration. In *Proceedings of the American Control Conference*, pp. 3475–3479. American Automatic Control Council, Evanston, IL.
- Brafman, R. I., Tennenholtz, M. (2003). R-max – a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231.
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.
- Breiter, H. C., Aharon, I., Kahneman, D., Dale, A., Shizgal, P. (2001). Functional imaging of neural responses to expectancy and experience of monetary gains and losses. *Neuron*, 30(2):619–639.
- Breland, K., Breland, M. (1961). The misbehavior of organisms. *American Psychologist*, 16(11):681–684.
- Bridle, J. S. (1990). Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimates of parameters. In *Advances in Neural Information Processing Systems 2*, pp. 211–217. Morgan Kaufmann, San Mateo, CA.
- Broomhead, D. S., Lowe, D. (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355.
- Bromberg-Martin, E. S., Matsumoto, M., Hong, S., Hikosaka, O. (2010). A pallidus-habenula-dopamine pathway signals inferred stimulus values. *Journal of Neurophysiology*, 104(2):1068–1076.
- Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.
- Brown, J., Bullock, D., Grossberg, S. (1999). How the basal ganglia use parallel excitatory and inhibitory learning pathways to selectively respond to unexpected rewarding cues. *The Journal of Neuroscience*, 19(23):10502–10511.

- Bryson, A. E., Jr. (1996). Optimal control—1950 to 1985. *IEEE Control Systems*, 13(3):26–33.
- Buchanan, B. G., Mitchell, T., Smith, R. G., Johnson, C. R., Jr. (1978). Models of learning systems. *Encyclopedia of Computer Science and technology*, 11.
- Buhusi, C. V., Schmajuk, N. A. (1999). Timing in simple conditioning and occasion setting: A neural network approach. *Behavioural Processes*, 45(1):33–57.
- Bušoniu, L., Lazaric, A., Ghavamzadeh, M., Munos, R., Babuška, R., De Schutter, B. (2012). Least-squares methods for policy iteration. In M. Wiering and M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 75–109. Springer-Verlag Berlin Heidelberg.
- Bush, R. R., Mosteller, F. (1955). *Stochastic Models for Learning*. Wiley, New York.
- Byrne, J. H., Gingrich, K. J., Baxter, D. A. (1990). Computational capabilities of single neurons: Relationship to simple forms of associative and nonassociative learning in *aplysia*. In R. D. Hawkins and G. H. Bower (Eds.), *Computational Models of Learning*, pp. 31–63. Academic Press, New York.
- Calabresi, P., Picconi, B., Tozzi, A., Filippo, M. D. (2007). Dopamine-mediated regulation of corticostriatal synaptic plasticity. *Trends in Neuroscience*, 30(5):211–219.
- Camerer, C. (2011). *Behavioral Game Theory: Experiments in Strategic Interaction*. Princeton University Press.
- Campbell, D. T. (1960). Blind variation and selective survival as a general strategy in knowledge-processes. In M. C. Yovits and S. Cameron (Eds.), *Self-Organizing Systems*, pp. 205–231. Pergamon, New York.
- Cao, X. R. (2009). Stochastic learning and optimization—A sensitivity-based approach. *Annual Reviews in Control*, 33(1):11–24.
- Cao, X. R., Chen, H. F. (1997). Perturbation realization, potentials, and sensitivity analysis of Markov processes. *IEEE Transactions on Automatic Control*, 42(10):1382–1393.
- Carlström, J., Nordström, E. (1997). Control of self-similar ATM call traffic by reinforcement learning. In *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications 3*, pp. 54–62. Erlbaum, Hillsdale, NJ.
- Chapman, D., Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the Twelfth International Conference on Artificial Intelligence*, pp. 726–731. Morgan Kaufmann, San Mateo, CA.
- Chaslot, G., Bakkes, S., Szita, I., Spronck, P. (2008). Monte-Carlo tree search: A new framework for game AI. In *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIDE-08)*, pp. 216–217. AAAI Press, Menlo Park, CA.
- Chow, C.-S., Tsitsiklis, J. N. (1991). An optimal one-way multigrid algorithm for discrete-time stochastic control. *IEEE Transactions on Automatic Control*, 36(8):898–914.
- Chrisman, L. (1992). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 183–188. AAAI/MIT Press, Menlo Park, CA.
- Christensen, J., Korf, R. E. (1986). A unified theory of heuristic evaluation functions and its application to learning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pp. 148–152. Morgan Kaufmann.
- Cichosz, P. (1995). Truncating temporal differences: On the efficient implementation of TD(λ) for reinforcement learning. *Journal of Artificial Intelligence Research*, 2:287–318.
- Ciosek, K., Whiteson, S. (2017). Expected policy gradients. ArXiv:1706.05374v1. A revised version appeared in *Proceedings of the Annual Conference of the Association for the Advancement of Artificial Intelligence*, pp. 2868–2875.
- Ciosek, K., Whiteson, S. (2018). Expected policy gradients for reinforcement learning. ArXiv: 1801.03326.

- Claridge-Chang, A., Roorda, R. D., Vrontou, E., Sjulson, L., Li, H., Hirsh, J., Miesenböck, G. (2009). Writing memories with light-addressable reinforcement circuitry. *Cell*, 139(2):405–415.
- Clark, R. E., Squire, L. R. (1998). Classical conditioning and brain systems: the role of awareness. *Science*, 280(5360):77–81.
- Clark, W. A., Farley, B. G. (1955). Generalization of pattern recognition in a self-organizing system. In *Proceedings of the 1955 Western Joint Computer Conference*, pp. 86–91.
- Clouse, J. (1996). *On Integrating Apprentice Learning and Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst. Appeared as CMPSCI Technical Report 96-026.
- Clouse, J., Utgoff, P. (1992). A teaching method for reinforcement learning systems. In *Proceedings of the 9th International Workshop on Machine Learning*, pp. 92–101. Morgan Kaufmann.
- Cobo, L. C., Zang, P., Isbell, C. L., Thomaz, A. L. (2011). Automatic state abstraction from demonstration. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pp. 1243–1248. AAAI Press.
- Connell, J. (1989). A colony architecture for an artificial creature. Technical Report AI-TR-1151. MIT Artificial Intelligence Laboratory, Cambridge, MA.
- Connell, M. E., Utgoff, P. E. (1987). Learning to control a dynamic physical system. *Computational intelligence*, 3(1):330–337.
- Contreras-Vidal, J. L., Schultz, W. (1999). A predictive reinforcement model of dopamine neurons for learning approach behavior. *Journal of Computational Neuroscience*, 6(3):191–214.
- Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games (CG'06)*, pp. 72–83. Springer-Verlag Berlin, Heidelberg.
- Courville, A. C., Daw, N. D., Touretzky, D. S. (2006). Bayesian theories of conditioning in a changing world. *Trends in Cognitive Science*, 10(7):294–300.
- Craik, K. J. W. (1943). *The Nature of Explanation*. Cambridge University Press, Cambridge.
- Cross, J. G. (1973). A stochastic learning model of economic behavior. *The Quarterly Journal of Economics*, 87(2):239–266.
- Crow, T. J. (1968). Cortical synapses and reinforcement: a hypothesis. *Nature*, 219(5155):736–737.
- Curtiss, J. H. (1954). A theoretical comparison of the efficiencies of two classical methods and a Monte Carlo method for computing one component of the solution of a set of linear algebraic equations. In H. A. Meyer (Ed.), *Symposium on Monte Carlo Methods*, pp. 191–233. Wiley, New York.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.
- Cziko, G. (1995). *Without Miracles: Universal Selection Theory and the Second Darwinian Revolution*. MIT Press, Cambridge, MA.
- Dabney, W. (2014). *Adaptive step-sizes for reinforcement learning*. PhD thesis, University of Massachusetts, Amherst.
- Dabney, W., Barto, A. G. (2012). Adaptive step-size for online temporal difference learning. In *Proceedings of the Annual Conference of the Association for the Advancement of Artificial Intelligence*.
- Daniel, J. W. (1976). Splines and efficiency in dynamic programming. *Journal of Mathematical Analysis and Applications*, 54:402–407.
- Dann, C., Neumann, G., Peters, J. (2014). Policy evaluation with temporal differences: A survey and comparison. *Journal of Machine Learning Research*, 15:809–883.

- Daw, N. D., Courville, A. C., Touretzky, D. S. (2003). Timing and partial observability in the dopamine system. In *Advances in Neural Information Processing Systems 15*, pp. 99–106. MIT Press, Cambridge, MA.
- Daw, N. D., Courville, A. C., Touretzky, D. S. (2006). Representation and timing in theories of the dopamine system. *Neural Computation*, 18(7):1637–1677.
- Daw, N. D., Niv, Y., Dayan, P. (2005). Uncertainty based competition between prefrontal and dorsolateral striatal systems for behavioral control. *Nature Neuroscience*, 8(12):1704–1711.
- Daw, N. D., Shohamy, D. (2008). The cognitive neuroscience of motivation and learning. *Social Cognition*, 26(5):593–620.
- Dayan, P. (1991). Reinforcement comparison. In D. S. Touretzky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton (Eds.), *Connectionist Models: Proceedings of the 1990 Summer School*, pp. 45–51. Morgan Kaufmann.
- Dayan, P. (1992). The convergence of $TD(\lambda)$ for general λ . *Machine Learning*, 8(3):341–362.
- Dayan, P. (2002). Matters temporal. *Trends in Cognitive Sciences*, 6(3):105–106.
- Dayan, P., Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, Cambridge, MA.
- Dayan, P., Berridge, K. C. (2014). Model-based and model-free Pavlovian reward learning: Revaluation, revision, and revaluation. *Cognitive, Affective, & Behavioral Neuroscience*, 14(2):473–492.
- Dayan, P., Niv, Y. (2008). Reinforcement learning: the good, the bad and the ugly. *Current Opinion in Neurobiology*, 18(2):185–196.
- Dayan, P., Niv, Y., Seymour, B., Daw, N. D. (2006). The misbehavior of value and the discipline of the will. *Neural Networks*, 19(8):1153–1160.
- Dayan, P., Sejnowski, T. (1994). $TD(\lambda)$ converges with probability 1. *Machine Learning*, 14(3):295–301.
- De Asis, K., Hernandez-Garcia, J. F., Holland, G. Z., Sutton, R. S. (2017). Multi-step Reinforcement Learning: A Unifying Algorithm. ArXiv:1703.01327.
- de Farias, D. P. (2002). The Linear Programming Approach to Approximate Dynamic Programming: Theory and Application. Stanford University PhD thesis.
- de Farias, D. P., Van Roy, B. (2003). The linear programming approach to approximate dynamic programming. *Operations Research* 51(6):850–865.
- Dean, T., Lin, S.-H. (1995). Decomposition techniques for planning in stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1121–1127. Morgan Kaufmann. See also Technical Report CS-95-10, Brown University, Department of Computer Science, 1995.
- Degrassi, T., Pilarski, P. M., Sutton, R. S. (2012). Model-free reinforcement learning with continuous action in practice. In *2012 American Control Conference*, pp. 2177–2182. IEEE.
- Degrassi, T., White, M., Sutton, R. S. (2012). Off-policy actor–critic. In *Proceedings of the 29th International Conference on Machine Learning*. ArXiv:1205.4839, 2012.
- Denardo, E. V. (1967). Contraction mappings in the theory underlying dynamic programming. *SIAM Review*, 9(2):165–177.
- Dennett, D. C. (1978). Why the Law of Effect Will Not Go Away. *Brainstorms*, pp. 71–89. Bradford/MIT Press, Cambridge, MA.
- Derthick, M. (1984). Variations on the Boltzmann machine learning algorithm. Carnegie-Mellon University Department of Computer Science Technical Report No. CMU-CS-84-120.
- Deutsch, J. A. (1953). A new type of behaviour theory. *British Journal of Psychology. General Section*, 44(4):304–317.

- Deutsch, J. A. (1954). A machine with insight. *Quarterly Journal of Experimental Psychology*, 6(1):6–11.
- Dick, T. (2015). *Policy Gradient Reinforcement Learning Without Regret*. M.Sc. thesis, University of Alberta.
- Dickinson, A. (1980). *Contemporary Animal Learning Theory*. Cambridge University Press.
- Dickinson, A. (1985). Actions and habits: the development of behavioral autonomy. *Phil. Trans. R. Soc. Lond. B*, 308(1135):67–78.
- Dickinson, A., Balleine, B. W. (2002). The role of learning in motivation. In C. R. Gallistel (Ed.), *Stevens' Handbook of Experimental Psychology*, volume 3, pp. 497–533. Wiley, NY.
- Dietterich, T. G., Buchanan, B. G. (1984). The role of the critic in learning systems. In O. G. Selfridge, E. L. Risssland, and M. A. Arbib (Eds.), *Adaptive Control of Ill-Defined Systems*, pp. 127–147. Plenum Press, NY.
- Dietterich, T. G., Flann, N. S. (1995). Explanation-based learning and reinforcement learning: A unified view. In A. Prideditis and S. Russell (Eds.), *Proceedings of the 12th International Conference on Machine Learning*, pp. 176–184. Morgan Kaufmann.
- Dietterich, T. G., Wang, X. (2002). Batch value function approximation via support vectors. In *Advances in Neural Information Processing Systems 14*, pp. 1491–1498. MIT Press, Cambridge, MA.
- Diuk, C., Cohen, A., Littman, M. L. (2008). An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, pp. 240–247. ACM, New York.
- Dolan, R. J., Dayan, P. (2013). Goals and habits in the brain. *Neuron*, 80(2):312–325.
- Doll, B. B., Simon, D. A., Daw, N. D. (2012). The ubiquity of model-based reinforcement learning. *Current Opinion in Neurobiology*, 22(6):1–7.
- Donahoe, J. W., Burgos, J. E. (2000). Behavior analysis and revaluation. *Journal of the Experimental Analysis of Behavior*, 74(3):331–346.
- Dorigo, M., Colombetti, M. (1994). Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370.
- Doya, K. (1996). Temporal difference learning in continuous time and space. In *Advances in Neural Information Processing Systems 8*, pp. 1073–1079. MIT Press, Cambridge, MA.
- Doya, K., Sejnowski, T. J. (1995). A novel reinforcement model of birdsong vocalization learning. In *Advances in Neural Information Processing Systems 7*, pp. 101–108. MIT Press, Cambridge, MA.
- Doya, K., Sejnowski, T. J. (1998). A computational model of birdsong learning by auditory experience and auditory feedback. In P. W. F. Poon and J. F. Brugge (Eds.), *Central Auditory Processing and Neural Modeling*, pp. 77–88. Springer, Boston, MA.
- Doyle, P. G., Snell, J. L. (1984). *Random Walks and Electric Networks*. The Mathematical Association of America. Carus Mathematical Monograph 22.
- Dreyfus, S. E., Law, A. M. (1977). *The Art and Theory of Dynamic Programming*. Academic Press, New York.
- Du, S. S., Chen, J., Li, L., Xiao, L., Zhou, D. (2017). Stochastic variance reduction methods for policy evaluation. *Proceedings of the 34th International Conference on Machine Learning*, pp. 1049–1058. ArXiv:1702.07944.
- Duda, R. O., Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. Wiley, New York.

- Duff, M. O. (1995). Q-learning for bandit problems. In *Proceedings of the 12th International Conference on Machine Learning*, pp. 209–217. Morgan Kaufmann.
- Egger, D. M., Miller, N. E. (1962). Secondary reinforcement in rats as a function of information value and reliability of the stimulus. *Journal of Experimental Psychology*, 64:97–104.
- Eshel, N., Tian, J., Bukwich, M., Uchida, N. (2016). Dopamine neurons share common response function for reward prediction error. *Nature Neuroscience*, 19(3):479–486.
- Estes, W. K. (1943). Discriminative conditioning. I. A discriminative property of conditioned anticipation. *Journal of Experimental Psychology*, 32(2):150–155.
- Estes, W. K. (1948). Discriminative conditioning. II. Effects of a Pavlovian conditioned stimulus upon a subsequently established operant response. *Journal of Experimental Psychology*, 38(2):173–177.
- Estes, W. K. (1950). Toward a statistical theory of learning. *Psychological Review*, 57(2):94–107.
- Farley, B. G., Clark, W. A. (1954). Simulation of self-organizing systems by digital computer. *IRE Transactions on Information Theory*, 4(4):76–84.
- Farries, M. A., Fairhall, A. L. (2007). Reinforcement learning with modulated spike timing-dependent synaptic plasticity. *Journal of Neurophysiology*, 98(6):3648–3665.
- Feldbaum, A. A. (1965). *Optimal Control Systems*. Academic Press, New York.
- Finch, G., Culler, E. (1934). Higher order conditioning with constant motivation. *The American Journal of Psychology*:596–602.
- Finnsson, H., Björnsson, Y. (2008). Simulation-based approach to general game playing. In *Proceedings of the Association for the Advancement of Artificial Intelligence*, pp. 259–264.
- Fiorillo, C. D., Yun, S. R., Song, M. R. (2013). Diversity and homogeneity in responses of midbrain dopamine neurons. *The Journal of Neuroscience*, 33(11):4693–4709.
- Florian, R. V. (2007). Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6):1468–1502.
- Fogel, L. J., Owens, A. J., Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley and Sons.
- French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135.
- Frey, U., Morris, R. G. M. (1997). Synaptic tagging and long-term potentiation. *Nature*, 385(6616):533–536.
- Frémaux, N., Sprekeler, H., Gerstner, W. (2010). Functional requirements for reward-modulated spike-timing-dependent plasticity. *The Journal of Neuroscience*, 30(40): 13326–13337
- Friedman, J. H., Bentley, J. L., Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226.
- Friston, K. J., Tononi, G., Reeye, G. N., Sporns, O., Edelman, G. M. (1994). Value-dependent selection in the brain: Simulation in a synthetic neural model. *Neuroscience*, 59(2):229–243.
- Fu, K. S. (1970). Learning control systems—Review and outlook. *IEEE Transactions on Automatic Control*, 15(2):210–221.
- Galanter, E., Gerstenhaber, M. (1956). On thought: The extrinsic theory. *Psychological Review*, 63(4):218–227.
- Gallistel, C. R. (2005). Deconstructing the law of effect. *Games and Economic Behavior*, 52(2):410–423.
- Gardner, M. (1973). Mathematical games. *Scientific American*, 228(1):108–115.
- Geist, M., Scherrer, B. (2014). Off-policy learning with eligibility traces: A survey. *Journal of Machine Learning Research*, 15(1):289–333.

- Gelly, S., Silver, D. (2007). Combining online and offline knowledge in UCT. *Proceedings of the 24th International Conference on Machine Learning*, pp. 273–280.
- Gelperin, A., Hopfield, J. J., Tank, D. W. (1985). The logic of *limax* learning. In A. Selverston (Ed.), *Model Neural Networks and Behavior*, pp. 247–261. Plenum Press, New York.
- Genesereth, M., Thielescher, M. (2014). General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2):1–229.
- Gershman, S. J., Moustafa, A. A., Ludvig, E. A. (2014). Time representation in reinforcement learning models of the basal ganglia. *Frontiers in Computational Neuroscience*, 7:194.
- Gershman, S. J., Pesaran, B., Daw, N. D. (2009). Human reinforcement learning subdivides structured action spaces by learning effector-specific values. *The Journal of Neuroscience*, 29(43):13524–13531.
- Ghiassian, S., Rafiee, B., Sutton, R. S. (2016). A first empirical study of emphatic temporal difference learning. Workshop on Continual Learning and Deep Learning at the Conference on Neural Information Processing Systems. ArXiv:1705.04185.
- Ghiassian, S., Patterson, A., White, M., Sutton, R. S., White, A. (2018). Online off-policy prediction. ArXiv:1811.02597.
- Gibbs, C. M., Cool, V., Land, T., Kehoe, E. J., Gormezano, I. (1991). Second-order conditioning of the rabbit's nictitating membrane response. *Integrative Physiological and Behavioral Science*, 26(4):282–295.
- Gittins, J. C., Jones, D. M. (1974). A dynamic allocation index for the sequential design of experiments. In J. Gani, K. Sarkadi, and I. Vincze (Eds.), *Progress in Statistics*, pp. 241–266. North-Holland, Amsterdam–London.
- Glimcher, P. W. (2011). Understanding dopamine and reinforcement learning: The dopamine reward prediction error hypothesis. *Proceedings of the National Academy of Sciences*, 108(Supplement 3):15647–15654.
- Glimcher, P. W. (2003). *Decisions, Uncertainty, and the Brain: The science of Neuroeconomics*. MIT Press, Cambridge, MA.
- Glimcher, P. W., Fehr, E. (Eds.) (2013). *Neuroeconomics: Decision Making and the Brain, Second Edition*. Academic Press.
- Goethe, J. W. V. (1878). The Sorcerer's Apprentice. In *The Permanent Goethe*, p. 349. The Dial Press, Inc., New York.
- Goldstein, H. (1957). *Classical Mechanics*. Addison-Wesley, Reading, MA.
- Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA.
- Goodwin, G. C., Sin, K. S. (1984). *Adaptive Filtering Prediction and Control*. Prentice-Hall, Englewood Cliffs, NJ.
- Gopnik, A., Glymour, C., Sobel, D., Schulz, L. E., Kushnir, T., Danks, D. (2004). A theory of causal learning in children: Causal maps and Bayes nets. *Psychological Review*, 111(1):3–32.
- Gordon, G. J. (1995). Stable function approximation in dynamic programming. In A. Prieditis and S. Russell (Eds.), *Proceedings of the 12th International Conference on Machine Learning*, pp. 261–268. Morgan Kaufmann. An expanded version was published as Technical Report CMU-CS-95-103. Carnegie Mellon University, Pittsburgh, PA, 1995.
- Gordon, G. J. (1996a). Chattering in SARSA(λ). CMU learning lab internal report.
- Gordon, G. J. (1996b). Stable fitted reinforcement learning. In *Advances in Neural Information Processing Systems 8*, pp. 1052–1058. MIT Press, Cambridge, MA.
- Gordon, G. J. (1999). *Approximate Solutions to Markov Decision Processes*. PhD thesis, Carnegie Mellon University, Pittsburgh PA. Pittsburgh, PA.
- Gordon, G. J. (2001). Reinforcement learning with function approximation converges to a

- region. In *Advances in Neural Information Processing Systems 13*, pp. 1040–1046. MIT Press, Cambridge, MA.
- Graybiel, A. M. (2000). The basal ganglia. *Current Biology*, 10(14):R509–R511.
- Greensmith, E., Bartlett, P. L., Baxter, J. (2002). Variance reduction techniques for gradient estimates in reinforcement learning. In *Advances in Neural Information Processing Systems 14*, pp. 1507–1514. MIT Press, Cambridge, MA.
- Greensmith, E., Bartlett, P. L., Baxter, J. (2004). Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530.
- Griffith, A. K. (1966). A new machine learning technique applied to the game of checkers. Technical Report Project MAC, Artificial Intelligence Memo 94. Massachusetts Institute of Technology, Cambridge, MA.
- Griffith, A. K. (1974). A comparison and evaluation of three machine learning procedures as applied to the game of checkers. *Artificial Intelligence*, 5(2):137–148.
- Grondman, I., Busoniu, L., Lopes, G. A., Babuska, R. (2012). A survey of actor–critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307.
- Grossberg, S. (1975). A neural model of attention, reinforcement, and discrimination learning. *International Review of Neurobiology*, 18:263–327.
- Grossberg, S., Schmajuk, N. A. (1989). Neural dynamics of adaptive timing and temporal discrimination during associative learning. *Neural Networks*, 2(2):79–102.
- Gullapalli, V. (1990). A stochastic reinforcement algorithm for learning real-valued functions. *Neural Networks*, 3(6): 671–692.
- Gullapalli, V., Barto, A. G. (1992). Shaping as a method for accelerating reinforcement learning. In *Proceedings of the 1992 IEEE International Symposium on Intelligent Control*, pp. 554–559. IEEE.
- Gurvits, L., Lin, L.-J., Hanson, S. J. (1994). Incremental learning of evaluation functions for absorbing Markov chains: New methods and theorems. Siemens Corporate Research, Princeton, NJ.
- Hackman, L. (2012). *Faster Gradient-TD Algorithms*. M.Sc. thesis, University of Alberta, Edmonton.
- Hallak, A., Tamar, A., Mannor, S. (2015). Emphatic TD Bellman operator is a contraction. ArXiv:1508.03411.
- Hallak, A., Tamar, A., Munos, R., Mannor, S. (2016). Generalized emphatic temporal difference learning: Bias-variance analysis. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 1631–1637. AAAI Press, Menlo Park, CA.
- Hammer, M. (1997). The neural basis of associative reward learning in honeybees. *Trends in Neuroscience*, 20(6):245–252.
- Hammer, M., Menzel, R. (1995). Learning and memory in the honeybee. *The Journal of Neuroscience*, 15(3):1617–1630.
- Hampson, S. E. (1983). *A Neural Model of Adaptive Behavior*. PhD thesis, University of California, Irvine.
- Hampson, S. E. (1989). *Connectionist Problem Solving: Computational Aspects of Biological Learning*. Birkhauser, Boston.
- Hare, T. A., O'Doherty, J., Camerer, C. F., Schultz, W., Rangel, A. (2008). Dissociating the role of the orbitofrontal cortex and the striatum in the computation of goal values and prediction errors. *The Journal of Neuroscience*, 28(22):5623–5630.

- Harth, E., Tzanakou, E. (1974). Alopex: A stochastic method for determining visual receptive fields. *Vision Research*, 14(12):1475–1482.
- Hassabis, D., Maguire, E. A. (2007). Deconstructing episodic memory with construction. *Trends in Cognitive Sciences*, 11(7):299–306.
- Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T., Boutilier, C. (1998). Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pp. 220–229. Morgan Kaufmann.
- Hawkins, R. D., Kandel, E. R. (1984). Is there a cell-biological alphabet for simple forms of learning? *Psychological Review*, 91(3):375–391.
- Haykin, S. (1994). *Neural networks: A Comprehensive Foundation*, Macmillan, New York.
- He, K., Huertas, M., Hong, S. Z., Tie, X., Hell, J. W., Shouval, H., Kirkwood, A. (2015). Distinct eligibility traces for LTP and LTD in cortical synapses. *Neuron*, 88(3):528–538.
- He, K., Zhang, X., Ren, S., Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the 1992 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778.
- Hebb, D. O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. John Wiley and Sons Inc., New York. Reissued by Lawrence Erlbaum Associates Inc., Mahwah NJ, 2002.
- Hengst, B. (2012). Hierarchical approaches. In M. Wiering and M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 293–323. Springer-Verlag Berlin Heidelberg.
- Herrnstein, R. J. (1970). On the Law of Effect. *Journal of the Experimental Analysis of Behavior*, 13(2):243–266.
- Hersh, R., Griego, R. J. (1969). Brownian motion and potential theory. *Scientific American*, 220(3):66–74.
- Hester, T., Stone, P. (2012). Learning and using models. In M. Wiering and M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 111–141. Springer-Verlag Berlin Heidelberg.
- Hesterberg, T. C. (1988). *Advances in Importance Sampling*, PhD thesis, Statistics Department, Stanford University.
- Hilgard, E. R. (1956). *Theories of Learning, Second Edition*. Appleton-Century-Crofts, Inc., New York.
- Hilgard, E. R., Bower, G. H. (1975). *Theories of Learning*. Prentice-Hall, Englewood Cliffs, NJ.
- Hinton, G. E. (1984). Distributed representations. Technical Report CMU-CS-84-157. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.
- Hinton, G. E., Osindero, S., Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554.
- Hochreiter, S., Schmidhuber, J. (1997). LSTM can solve hard time lag problems. In *Advances in Neural Information Processing Systems 9*, pp. 473–479. MIT Press, Cambridge, MA.
- Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.
- Holland, J. H. (1976). Adaptation. In R. Rosen and F. M. Snell (Eds.), *Progress in Theoretical Biology*, vol. 4, pp. 263–293. Academic Press, New York.
- Holland, J. H. (1986). Escaping brittleness: The possibility of general-purpose learning algorithms applied to rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, vol. 2, pp. 593–623. Morgan Kaufmann.
- Hollerman, J. R., Schultz, W. (1998). Dopamine neurons report an error in the temporal prediction of reward during learning. *Nature Neuroscience*, 1(4):304–309.

- Houk, J. C., Adams, J. L., Barto, A. G. (1995). A model of how the basal ganglia generates and uses neural signals that predict reinforcement. In J. C. Houk, J. L. Davis, and D. G. Beiser (Eds.), *Models of Information Processing in the Basal Ganglia*, pp. 249–270. MIT Press, Cambridge, MA.
- Howard, R. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.
- Hull, C. L. (1932). The goal-gradient hypothesis and maze learning. *Psychological Review*, 39(1):25–43.
- Hull, C. L. (1943). *Principles of Behavior*. Appleton-Century, New York.
- Hull, C. L. (1952). *A Behavior System*. Wiley, New York.
- Ioffe, S., Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. ArXiv:1502.03167.
- İpek, E., Mutlu, O., Martínez, J. F., Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA '08:Proceedings of the 35th Annual International Symposium on Computer Architecture*, pp. 39–50. IEEE Computer Society Washington, DC.
- Izhikevich, E. M. (2007). Solving the distal reward problem through linkage of STDP and dopamine signaling. *Cerebral Cortex*, 17(10):2443–2452.
- Jaakkola, T., Jordan, M. I., Singh, S. P. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201.
- Jaakkola, T., Singh, S. P., Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In *Advances in Neural Information Processing Systems* 7, pp. 345–352. MIT Press, Cambridge, MA.
- Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1(4):295–307.
- Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., Kavukcuoglu, K. (2016). Reinforcement learning with unsupervised auxiliary tasks. ArXiv:1611.05397.
- Jaeger, H. (1997). Observable operator models and conditioned continuation representations. Arbeitspapiere der GMD 1043, GMD Forschungszentrum Informationstechnik, Sankt Augustin, Germany.
- Jaeger, H. (1998). *Discrete Time, Discrete Valued Observable Operator Models: A Tutorial*. GMD-Forschungszentrum Informationstechnik.
- Jaeger, H. (2000). Observable operator models for discrete stochastic time series. *Neural Computation*, 12(6):1371–1398.
- Jaeger, H. (2002). Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the ‘echo state network’ approach. German National Research Center for Information Technology, Technical Report GMD report 159, 2002.
- Joel, D., Niv, Y., Ruppin, E. (2002). Actor–critic models of the basal ganglia: New anatomical and computational perspectives. *Neural Networks*, 15(4):535–547.
- Johnson, A., Redish, A. D. (2007). Neural ensembles in CA3 transiently encode paths forward of the animal at a decision point. *The Journal of Neuroscience*, 27(45):12176–12189.
- Kaelbling, L. P. (1993a). Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the 10th International Conference on Machine Learning*, pp. 167–173. Morgan Kaufmann.
- Kaelbling, L. P. (1993b). *Learning in Embedded Systems*. MIT Press, Cambridge, MA.
- Kaelbling, L. P. (Ed.) (1996). Special triple issue on reinforcement learning, *Machine Learning*, 22(1/2/3).
- Kaelbling, L. P., Littman, M. L., Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

- Kakade, S. M. (2002). A natural policy gradient. In *Advances in Neural Information Processing Systems 14*, pp. 1531–1538. MIT Press, Cambridge, MA.
- Kakade, S. M. (2003). *On the Sample Complexity of Reinforcement Learning*. PhD thesis, University of London.
- Kakutani, S. (1945). Markov processes and the Dirichlet problem. *Proceedings of the Japan Academy*, 21(3-10):227–233.
- Kalos, M. H., Whitlock, P. A. (1986). *Monte Carlo Methods*. Wiley, New York.
- Kamin, L. J. (1968). “Attention-like” processes in classical conditioning. In M. R. Jones (Ed.), *Miami Symposium on the Prediction of Behavior, 1967: Aversive Stimulation*, pp. 9–31. University of Miami Press, Coral Gables, Florida.
- Kamin, L. J. (1969). Predictability, surprise, attention, and conditioning. In B. A. Campbell and R. M. Church (Eds.), *Punishment and Aversive Behavior*, pp. 279–296. Appleton-Century-Crofts, New York.
- Kandel, E. R., Schwartz, J. H., Jessell, T. M., Siegelbaum, S. A., Hudspeth, A. J. (Eds.) (2013). *Principles of Neural Science, Fifth Edition*. McGraw-Hill Companies, Inc.
- Karampatziakis, N., Langford, J. (2010). Online importance weight aware updates. ArXiv:1011.1576.
- Kashyap, R. L., Blaydon, C. C., Fu, K. S. (1970). Stochastic approximation. In J. M. Mendel and K. S. Fu (Eds.), *Adaptive, Learning, and Pattern Recognition Systems: Theory and Applications*, pp. 329–355. Academic Press, New York.
- Kearney, A., Veeriah, V., Travnik, J., Sutton, R. S., Pilarski, P. M. (in preparation). TIDBD: Adapting Temporal-difference Step-sizes Through Stochastic Meta-descent.
- Kearns, M., Singh, S. (2002). Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2-3):209–232.
- Keerthi, S. S., Ravindran, B. (1997). Reinforcement learning. In E. Fieslerm and R. Beale (Eds.), *Handbook of Neural Computation*, C3. Oxford University Press, New York.
- Kehoe, E. J. (1982). Conditioning with serial compound stimuli: Theoretical and empirical issues. *Experimental Animal Behavior*, 1:30–65.
- Kehoe, E. J., Schreurs, B. G., Graham, P. (1987). Temporal primacy overrides prior training in serial compound conditioning of the rabbit’s nictitating membrane response. *Animal Learning & Behavior*, 15(4):455–464.
- Keiflin, R., Janak, P. H. (2015). Dopamine prediction errors in reward learning and addiction: From theory to neural circuitry. *Neuron*, 88(2):247–263.
- Kimble, G. A. (1961). *Hilgard and Marquis’ Conditioning and Learning*. Appleton-Century-Crofts, New York.
- Kimble, G. A. (1967). *Foundations of Conditioning and Learning*. Appleton-Century-Crofts, New York.
- Kingma, D., Ba, J. (2014). Adam: A method for stochastic optimization. ArXiv:1412.6980.
- Klopff, A. H. (1972). Brain function and adaptive systems—A heterostatic theory. Technical Report AFCRL-72-0164, Air Force Cambridge Research Laboratories, Bedford, MA. A summary appears in *Proceedings of the International Conference on Systems, Man, and Cybernetics (1974)*. IEEE Systems, Man, and Cybernetics Society, Dallas, TX.
- Klopff, A. H. (1975). A comparison of natural and artificial intelligence. *SIGART Newsletter*, 53:11–13.
- Klopff, A. H. (1982). *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemisphere, Washington, DC.
- Klopff, A. H. (1988). A neuronal model of classical conditioning. *Psychobiology*, 16(2):85–125.

- Klyubin, A. S., Polani, D., Nehaniv, C. L. (2005). Empowerment: A universal agent-centric measure of control. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation* (Vol. 1, pp. 128–135). IEEE.
- Kober, J., Peters, J. (2012). Reinforcement learning in robotics: A survey. In M. Wiering, M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 579–610. Springer-Verlag.
- Kocsis, L., Szepesvári, Cs. (2006). Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning*, pp. 282–293. Springer-Verlag Berlin Heidelberg.
- Kohonen, T. (1977). *Associative Memory: A System Theoretic Approach*. Springer-Verlag, Berlin.
- Koller, D., Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.
- Kolodziejek, C., Porr, B., Wörgötter, F. (2009). On the asymptotic equivalence between differential Hebbian and temporal difference learning. *Neural Computation*, 21(4):1173–1202.
- Kolter, J. Z. (2011). The fixed points of off-policy TD. In *Advances in Neural Information Processing Systems 24*, pp. 2169–2177. Curran Associates, Inc.
- Konda, V. R., Tsitsiklis, J. N. (2000). Actor-critic algorithms. In *Advances in Neural Information Processing Systems 12*, pp. 1008–1014. MIT Press, Cambridge, MA.
- Konda, V. R., Tsitsiklis, J. N. (2003). On actor-critic algorithms. *SIAM Journal on Control and Optimization*, 42(4):1143–1166.
- Konidaris, G. D., Osentoski, S., Thomas, P. S. (2011). Value function approximation in reinforcement learning using the Fourier basis. In *Proceedings of the Twenty-Fifth Conference of the Association for the Advancement of Artificial Intelligence*, pp. 380–385.
- Korf, R. E. (1988). Optimal path finding algorithms. In L. N. Kanal and V. Kumar (Eds.), *Search in Artificial Intelligence*, pp. 223–267. Springer-Verlag, Berlin.
- Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, 42(2–3), 189–211.
- Koshland, D. E. (1980). *Bacterial Chemotaxis as a Model Behavioral System*. Raven Press, New York.
- Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Vol. 1). MIT Press., Cambridge, MA.
- Kraft, L. G., Campagna, D. P. (1990). A summary comparison of CMAC neural network and traditional adaptive control systems. In T. Miller, R. S. Sutton, and P. J. Werbos (Eds.), *Neural Networks for Control*, pp. 143–169. MIT Press, Cambridge, MA.
- Kraft, L. G., Miller, W. T., Dietz, D. (1992). Development and application of CMAC neural network-based control. In D. A. White and D. A. Sofge (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pp. 215–232. Van Nostrand Reinhold, New York.
- Kumar, P. R., Varaiya, P. (1986). *Stochastic Systems: Estimation, Identification, and Adaptive Control*. Prentice-Hall, Englewood Cliffs, NJ.
- Kumar, P. R. (1985). A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, 23(3):329–380.
- Kumar, V., Kanal, L. N. (1988). The CDP, A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In L. N. Kanal and V. Kumar (Eds.), *Search in Artificial Intelligence*, pp. 1–37. Springer-Verlag, Berlin.
- Kushner, H. J., Dupuis, P. (1992). *Numerical Methods for Stochastic Control Problems in Continuous Time*. Springer-Verlag, New York.

- Kuvayev, L., Sutton, R.S. (1996). Model-based reinforcement learning with an approximate, learned model. *Proceedings of the Ninth Yale Workshop on Adaptive and Learning Systems*, pp. 101–105, Yale University, New Haven, CT.
- Lagoudakis, M., Parr, R. (2003). Least squares policy iteration. *Journal of Machine Learning Research*, 4 (Dec):1107–1149.
- Lai, T. L., Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22.
- Lakshmivarahan, S., Narendra, K. S. (1982). Learning algorithms for two-person zero-sum stochastic games with incomplete information: A unified approach. *SIAM Journal of Control and Optimization*, 20(4):541–552.
- Lammel, S., Lim, B. K., Malenka, R. C. (2014). Reward and aversion in a heterogeneous midbrain dopamine system. *Neuropharmacology*, 76:353–359.
- Lane, S. H., Handelman, D. A., Gelfand, J. J. (1992). Theory and development of higher-order CMAC neural networks. *IEEE Control Systems*, 12(2):23–30.
- LeCun, Y. (1985). Une procédure d'apprentissage pour réseau à seuil asymétrique (a learning scheme for asymmetric threshold networks). In *Proceedings of Cognitiva 85*, Paris, France.
- LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.
- Legenstein, R. W., Maass, D. P. (2008). A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback. *PLoS Computational Biology*, 4(10).
- Levy, W. B., Steward, D. (1983). Temporal contiguity requirements for long-term associative potentiation/depression in the hippocampus. *Neuroscience*, 8(4):791–797.
- Lewis, F. L., Liu, D. (Eds.) (2012). *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control*. John Wiley and Sons.
- Lewis, R. L., Howes, A., Singh, S. (2014). Computational rationality: Linking mechanism and behavior through utility maximization. *Topics in Cognitive Science*, 6(2):279–311.
- Li, L. (2012). Sample complexity bounds of exploration. In M. Wiering and M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 175–204. Springer-Verlag Berlin Heidelberg.
- Li, L., Chu, W., Langford, J., Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World Wide Web*, pp. 661–670. ACM, New York.
- Lin, C.-S., Kim, H. (1991). CMAC-based adaptive critic self-learning control. *IEEE Transactions on Neural Networks*, 2(5):530–533.
- Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321.
- Lin, L.-J., Mitchell, T. (1992). Reinforcement learning with hidden states. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pp. 271–280. MIT Press, Cambridge, MA.
- Littman, M. L., Cassandra, A. R., Kaelbling, L. P. (1995). Learning policies for partially observable environments: Scaling up. In *Proceedings of the 12th International Conference on Machine Learning*, pp. 362–370. Morgan Kaufmann.
- Littman, M. L., Dean, T. L., Kaelbling, L. P. (1995). On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, pp. 394–402.
- Littman, M. L., Sutton, R. S., Singh, S. (2002). Predictive representations of state. In *Advances in Neural Information Processing Systems 14*, pp. 1555–1561. MIT Press, Cambridge, MA.
- Liu, J. S. (2001). *Monte Carlo Strategies in Scientific Computing*. Springer-Verlag, Berlin.

- Ljung, L. (1998). System identification. In A. Procházka, J. Uhlíř, P. W. J. Rayner, and N. G. Kingsbury (Eds.), *Signal Analysis and Prediction*, pp. 163–173. Springer Science + Business Media New York, LLC.
- Ljung, L., Söderstrom, T. (1983). *Theory and Practice of Recursive Identification*. MIT Press, Cambridge, MA.
- Ljungberg, T., Apicella, P., Schultz, W. (1992). Responses of monkey dopamine neurons during learning of behavioral reactions. *Journal of Neurophysiology*, 67(1):145–163.
- Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28(1):47–66.
- Luce, D. (1959). *Individual Choice Behavior*. Wiley, New York.
- Ludvig, E. A., Bellemare, M. G., Pearson, K. G. (2011). A primer on reinforcement learning in the brain: Psychological, computational, and neural perspectives. In E. Alonso and E. Mondragón (Eds.), *Computational Neuroscience for Advancing Artificial Intelligence: Models, Methods and Applications*, pp. 111–44. Medical Information Science Reference, Hershey PA.
- Ludvig, E. A., Sutton, R. S., Kehoe, E. J. (2008). Stimulus representation and the timing of reward-prediction errors in models of the dopamine system. *Neural Computation*, 20(12):3034–3054.
- Ludvig, E. A., Sutton, R. S., Kehoe, E. J. (2012). Evaluating the TD model of classical conditioning. *Learning & behavior*, 40(3):305–319.
- Machado, A. (1997). Learning the temporal dynamics of behavior. *Psychological Review*, 104(2):241–265.
- Mackintosh, N. J. (1975). A theory of attention: Variations in the associability of stimuli with reinforcement. *Psychological Review*, 82(4):276–298.
- Mackintosh, N. J. (1983). *Conditioning and Associative Learning*. Clarendon Press, Oxford.
- Maclin, R., Shavlik, J. W. (1994). Incorporating advice into agents that learn from reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 694–699. AAAI Press, Menlo Park, CA.
- Maei, H. R. (2011). *Gradient Temporal-Difference Learning Algorithms*. PhD thesis, University of Alberta, Edmonton.
- Maei, H. R. (2018). Convergent actor-critic algorithms under off-policy training and function approximation. ArXiv:1802.07842.
- Maei, H. R., Sutton, R. S. (2010). GQ(λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence*, pp. 91–96.
- Maei, H. R., Szepesvári, Cs., Bhatnagar, S., Precup, D., Silver, D., Sutton, R. S. (2009). Convergent temporal-difference learning with arbitrary smooth function approximation. In *Advances in Neural Information Processing Systems 22*, pp. 1204–1212. Curran Associates, Inc.
- Maei, H. R., Szepesvári, Cs., Bhatnagar, S., Sutton, R. S. (2010). Toward off-policy learning control with function approximation. In *Proceedings of the 27th International Conference on Machine Learning*, pp. 719–726.
- Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1):159–196.
- Mahadevan, S., Liu, B., Thomas, P., Dabney, W., Giguere, S., Jacek, N., Gemp, I., Liu, J. (2014). Proximal reinforcement learning: A new theory of sequential decision making in primal-dual spaces. ArXiv:1405.6757.
- Mahadevan, S., Connell, J. (1992). Automatic programming of behavior-based robots using

- reinforcement learning. *Artificial Intelligence*, 55(2-3):311–365.
- Mahmood, A. R. (2017). *Incremental Off-Policy Reinforcement Learning Algorithms*. PhD thesis, University of Alberta, Edmonton.
- Mahmood, A. R., Sutton, R. S. (2015). Off-policy learning based on weighted importance sampling with linear computational complexity. In *Proceedings of the 31st Conference on Uncertainty in Artificial Intelligence*, pp. 552–561. AUAI Press Corvallis, Oregon.
- Mahmood, A. R., Sutton, R. S., Degris, T., Pilarski, P. M. (2012). Tuning-free step-size adaptation. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing, Proceedings*, pp. 2121–2124. IEEE.
- Mahmood, A. R., Yu, H., Sutton, R. S. (2017). Multi-step off-policy learning without importance sampling ratios. ArXiv:1702.03006.
- Mahmood, A. R., van Hasselt, H., Sutton, R. S. (2014). Weighted importance sampling for off-policy learning with linear function approximation. *Advances in Neural Information Processing Systems 27*, pp. 3014–3022. Curran Associates, Inc.
- Marbach, P., Tsitsiklis, J. N. (1998). Simulation-based optimization of Markov reward processes. MIT Technical Report LIDS-P-2411.
- Marbach, P., Tsitsiklis, J. N. (2001). Simulation-based optimization of Markov reward processes. *IEEE Transactions on Automatic Control*, 46(2):191–209.
- Markram, H., Lübke, J., Frotscher, M., Sakmann, B. (1997). Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science*, 275(5297):213–215.
- Martínez, J. F., İpek, E. (2009). Dynamic multicore resource management: A machine learning approach. *Micro, IEEE*, 29(5):8–17.
- Mataric, M. J. (1994). Reward functions for accelerated learning. In *Proceedings of the 11th International Conference on Machine Learning*, pp. 181–189. Morgan Kaufmann.
- Matsuda, W., Furuta, T., Nakamura, K. C., Hioki, H., Fujiyama, F., Arai, R., Kaneko, T. (2009). Single nigrostriatal dopaminergic neurons form widely spread and highly dense axonal arborizations in the neostriatum. *The Journal of Neuroscience*, 29(2):444–453.
- Mazur, J. E. (1994). *Learning and Behavior*, 3rd ed. Prentice-Hall, Englewood Cliffs, NJ.
- McCallum, A. K. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the 10th International Conference on Machine Learning*, pp. 190–196. Morgan Kaufmann.
- McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester NY.
- McCloskey, M., Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24:109–165.
- McClure, S. M., Daw, N. D., Montague, P. R. (2003). A computational substrate for incentive salience. *Trends in Neurosciences*, 26(8):423–428.
- McCulloch, W. S., Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133.
- McMahan, H. B., Gordon, G. J. (2005). Fast Exact Planning in Markov Decision Processes. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 151–160.
- Melo, F. S., Meyn, S. P., Ribeiro, M. I. (2008). An analysis of reinforcement learning with function approximation. In *Proceedings of the 25th International Conference on Machine Learning*, pp. 664–671.
- Mendel, J. M. (1966). A survey of learning control systems. *ISA Transactions*, 5:297–303.

- Mendel, J. M., McLaren, R. W. (1970). Reinforcement learning control and pattern recognition systems. In J. M. Mendel and K. S. Fu (Eds.), *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications*, pp. 287–318. Academic Press, New York.
- Michie, D. (1961). Trial and error. In S. A. Barnett and A. McLaren (Eds.), *Science Survey, Part 2*, pp. 129–145. Penguin, Harmondsworth.
- Michie, D. (1963). Experiments on the mechanisation of game learning. 1. characterization of the model and its parameters. *The Computer Journal*, 6(3):232–263.
- Michie, D. (1974). *On Machine Intelligence*. Edinburgh University Press, Edinburgh.
- Michie, D., Chambers, R. A. (1968). BOXES, An experiment in adaptive control. In E. Dale and D. Michie (Eds.), *Machine Intelligence 2*, pp. 137–152. Oliver and Boyd, Edinburgh.
- Miller, R. (1981). *Meaning and Purpose in the Intact Brain: A Philosophical, Psychological, and Biological Account of Conscious Process*. Clarendon Press, Oxford.
- Miller, W. T., An, E., Glanz, F., Carter, M. (1990). The design of CMAC neural networks for control. *Adaptive and Learning Systems*, 1:140–145.
- Miller, W. T., Glanz, F. H. (1996). *UNH-CMAC version 2.1: The University of New Hampshire Implementation of the Cerebellar Model Arithmetic Computer - CMAC*. Robotics Laboratory Technical Report, University of New Hampshire, Durham.
- Miller, S., Williams, R. J. (1992). Learning to control a bioreactor using a neural net Dyna-Q system. In *Proceedings of the Seventh Yale Workshop on Adaptive and Learning Systems*, pp. 167–172. Center for Systems Science, Dunham Laboratory, Yale University, New Haven.
- Miller, W. T., Scalera, S. M., Kim, A. (1994). Neural network control of dynamic balance for a biped walking robot. In *Proceedings of the Eighth Yale Workshop on Adaptive and Learning Systems*, pp. 156–161. Center for Systems Science, Dunham Laboratory, Yale University, New Haven.
- Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3):363–391.
- Minsky, M. L. (1954). *Theory of Neural-Analog Reinforcement Systems and Its Application to the Brain-Model Problem*. PhD thesis, Princeton University.
- Minsky, M. L. (1961). Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers*, 49:8–30. Reprinted in E. A. Feigenbaum and J. Feldman (Eds.), *Computers and Thought*, pp. 406–450. McGraw-Hill, New York, 1963.
- Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, NJ.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2013). Playing atari with deep reinforcement learning. ArXiv:1312.5602.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.
- Modayil, J., Sutton, R. S. (2014). Prediction driven behavior: Learning predictions that drive fixed responses. In *AAAI-14 Workshop on Artificial Intelligence and Robotics*, Quebec City, Canada.
- Modayil, J., White, A., Sutton, R. S. (2014). Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior*, 22(2):146–160.
- Monahan, G. E. (1982). State of the art—a survey of partially observable Markov decision processes: theory, models, and algorithms. *Management Science*, 28(1):1–16.

- Montague, P. R., Dayan, P., Nowlan, S. J., Pouget, A., Sejnowski, T. J. (1993). Using aperiodic reinforcement for directed self-organization during development. In *Advances in Neural Information Processing Systems 5*, pp. 969–976. Morgan Kaufmann.
- Montague, P. R., Dayan, P., Person, C., Sejnowski, T. J. (1995). Bee foraging in uncertain environments using predictive hebbian learning. *Nature*, 377(6551):725–728.
- Montague, P. R., Dayan, P., Sejnowski, T. J. (1996). A framework for mesencephalic dopamine systems based on predictive Hebbian learning. *The Journal of Neuroscience*, 16(5):1936–1947.
- Montague, P. R., Dolan, R. J., Friston, K. J., Dayan, P. (2012). Computational psychiatry. *Trends in Cognitive Sciences*, 16(1):72–80.
- Montague, P. R., Sejnowski, T. J. (1994). The predictive brain: Temporal coincidence and temporal order in synaptic learningmechanisms. *Learning & Memory*, 1(1):1–33.
- Moore, A. W. (1990). *Efficient Memory-Based Learning for Robot Control*. PhD thesis, University of Cambridge.
- Moore, A. W., Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13(1):103–130.
- Moore, A. W., Schneider, J., Deng, K. (1997). Efficient locally weighted polynomial regression predictions. In *Proceedings of the 14th International Conference on Machine Learning*. Morgan Kaufmann.
- Moore, J. W., Blazis, D. E. J. (1989). Simulation of a classically conditioned response: A cerebellar implementation of the sutton-barto-desmond model. In J. H. Byrne and W. O. Berry (Eds.), *Neural Models of Plasticity*, pp. 187–207. Academic Press, San Diego, CA.
- Moore, J. W., Choi, J.-S., Brunzell, D. H. (1998). Predictive timing under temporal uncertainty: The time derivative model of the conditioned response. In D. A. Rosenbaum and C. E. Collyer (Eds.), *Timing of Behavior*, pp. 3–34. MIT Press, Cambridge, MA.
- Moore, J. W., Desmond, J. E., Berthier, N. E., Blazis, E. J., Sutton, R. S., Barto, A. G. (1986). Simulation of the classically conditioned nictitating membrane response by a neuron-like adaptive element: I. Response topography, neuronal firing, and interstimulus intervals. *Behavioural Brain Research*, 21(2):143–154.
- Moore, J. W., Marks, J. S., Castagna, V. E., Polewan, R. J. (2001). Parameter stability in the TD model of complex CR topographies. In *Society for Neuroscience Abstracts*, 27:642.
- Moore, J. W., Schmajuk, N. A. (2008). Kamin blocking. *Scholarpedia*, 3(5):3542.
- Moore, J. W., Stickney, K. J. (1980). Formation of attentional-associative networks in real time:Role of the hippocampus and implications for conditioning. *Physiological Psychology*, 8(2):207–217.
- Mukundan, J., Martínez, J. F. (2012). MORSE, Multi-objective reconfigurable self-optimizing memory scheduler. In *IEEE 18th International Symposium on High Performance Computer Architecture*, pp. 1–12.
- Müller, M. (2002). Computer Go. *Artificial Intelligence*, 134(1):145–179.
- Munos, R., Stepleton, T., Harutyunyan, A., Bellemare, M. (2016). Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems 29*, pp. 1046–1054. Curran Associates, Inc.
- Naddaf, Y. (2010). *Game-Independent AI Agents for Playing Atari 2600 Console Games*. PhD thesis, University of Alberta, Edmonton.
- Narendra, K. S., Thathachar, M. A. L. (1974). Learning automata—A survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:323–334.
- Narendra, K. S., Thathachar, M. A. L. (1989). *Learning Automata: An Introduction*. Prentice-Hall, Englewood Cliffs, NJ.

- Narendra, K. S., Wheeler, R. M. (1983). An N-player sequential stochastic game with identical payoffs. *IEEE Transactions on Systems, Man, and Cybernetics*, 6:1154–1158.
- Narendra, K. S., Wheeler, R. M. (1986). Decentralized learning in finite Markov chains. *IEEE Transactions on Automatic Control*, 31(6):519–526.
- Nedić, A., Bertsekas, D. P. (2003). Least squares policy evaluation algorithms with linear function approximation. *Discrete Event Dynamic Systems*, 13(1-2):79–110.
- Ng, A. Y. (2003). *Shaping and Policy Search in Reinforcement Learning*. PhD thesis, University of California, Berkeley.
- Ng, A. Y., Harada, D., Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In I. Bratko and S. Dzeroski (Eds.), *Proceedings of the 16th International Conference on Machine Learning*, pp. 278–287.
- Ng, A. Y., Russell, S. J. (2000). Algorithms for inverse reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pp. 663–670.
- Niv, Y. (2009). Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53(3):139–154.
- Niv, Y., Daw, N. D., Dayan, P. (2006). How fast to work: Response vigor, motivation and tonic dopamine. In *Advances in Neural Information Processing Systems 18*, pp. 1019–1026. MIT Press, Cambridge, MA.
- Niv, Y., Daw, N. D., Joel, D., Dayan, P. (2007). Tonic dopamine: opportunity costs and the control of response vigor. *Psychopharmacology*, 191(3):507–520.
- Niv, Y., Joel, D., Dayan, P. (2006). A normative perspective on motivation. *Trends in Cognitive Sciences*, 10(8):375–381.
- Nouri, A., Littman, M. L. (2009). Multi-resolution exploration in continuous spaces. In *Advances in Neural Information Processing Systems 21*, pp. 1209–1216. Curran Associates, Inc.
- Nowé, A., Vrancx, P., Hauwerc, Y.-M. D. (2012). Game theory and multi-agent reinforcement learning. In M. Wiering and M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 441–467. Springer-Verlag Berlin Heidelberg.
- Nutt, D. J., Lingford-Hughes, A., Erritzoe, D., Stokes, P. R. A. (2015). The dopamine theory of addiction: 40 years of highs and lows. *Nature Reviews Neuroscience*, 16(5):305–312.
- O’Doherty, J. P., Dayan, P., Friston, K., Critchley, H., Dolan, R. J. (2003). Temporal difference models and reward-related learning in the human brain. *Neuron*, 38(2):329–337.
- O’Doherty, J. P., Dayan, P., Schultz, J., Deichmann, R., Friston, K., Dolan, R. J. (2004). Dissociable roles of ventral and dorsal striatum in instrumental conditioning. *Science*, 304(5669):452–454.
- Ólafsdóttir, H. F., Barry, C., Saleem, A. B., Hassabis, D., Spiers, H. J. (2015). Hippocampal place cells construct reward related sequences through unexplored space. *Elife*, 4:e06063.
- Oh, J., Guo, X., Lee, H., Lewis, R. L., Singh, S. (2015). Action-conditional video prediction using deep networks in Atari games. In *Advances in Neural Information Processing Systems 28*, pp. 2845–2853. Curran Associates, Inc.
- Olds, J., Milner, P. (1954). Positive reinforcement produced by electrical stimulation of the septal area and other regions of rat brain. *Journal of Comparative and Physiological Psychology*, 47(6):419–427.
- O'Reilly, R. C., Frank, M. J. (2006). Making working memory work: A computational model of learning in the prefrontal cortex and basal ganglia. *Neural Computation*, 18(2):283–328.
- O'Reilly, R. C., Frank, M. J., Hazy, T. E., Watz, B. (2007). PVLV, the primary value and learned value Pavlovian learning algorithm. *Behavioral Neuroscience*, 121(1):31–49.

- Omohundro, S. M. (1987). Efficient algorithms with neural network behavior. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.
- Ormoneit, D., Sen, Š. (2002). Kernel-based reinforcement learning. *Machine Learning*, 49(2-3):161–178.
- Oudeyer, P.-Y., Kaplan, F. (2007). What is intrinsic motivation? A typology of computational approaches. *Frontiers in Neurorobotics*, 1:6.
- Oudeyer, P.-Y., Kaplan, F., Hafner, V. V. (2007). Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation*, 11(2):265–286.
- Padoa-Schioppa, C., Assad, J. A. (2006). Neurons in the orbitofrontal cortex encode economic value. *Nature*, 441(7090):223–226.
- Page, C. V. (1977). Heuristics for signature table analysis as a pattern recognition technique. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(2):77–86.
- Pagnoni, G., Zink, C. F., Montague, P. R., Berns, G. S. (2002). Activity in human ventral striatum locked to errors of reward prediction. *Nature Neuroscience*, 5(2):97–98.
- Pan, W.-X., Schmidt, R., Wickens, J. R., Hyland, B. I. (2005). Dopamine cells respond to predicted events during classical conditioning: Evidence for eligibility traces in the reward-learning network. *The Journal of Neuroscience*, 25(26):6235–6242.
- Park, J., Kim, J., Kang, D. (2005). An RLS-based natural actor-critic algorithm for locomotion of a two-linked robot arm. *Computational Intelligence and Security*:65–72.
- Parks, P. C., Militzer, J. (1991). Improved allocation of weights for associative memory storage in learning control systems. In *IFAC Design Methods of Control Systems*, Zurich, Switzerland, pp. 507–512.
- Parr, R. (1988). *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California, Berkeley.
- Parr, R., Li, L., Taylor, G., Painter-Wakefield, C., Littman, M. L. (2008). An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pp. 752–759.
- Parr, R., Russell, S. (1995). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1088–1094. Morgan Kaufmann.
- Pavlov, I. P. (1927). *Conditioned Reflexes*. Oxford University Press, London.
- Pawlak, V., Kerr, J. N. D. (2008). Dopamine receptor activation is required for corticostriatal spike-timing-dependent plasticity. *The Journal of Neuroscience*, 28(10):2435–2446.
- Pawlak, V., Wickens, J. R., Kirkwood, A., Kerr, J. N. D. (2010). Timing is not everything: neuromodulation opens the STDP gate. *Frontiers in Synaptic Neuroscience*, 2:146. doi:10.3389/fnsyn.2010.00146.
- Pearce, J. M., Hall, G. (1980). A model for Pavlovian learning: Variation in the effectiveness of conditioning but not unconditioned stimuli. *Psychological Review*, 87(6):532–552.
- Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA.
- Pearl, J. (1995). Causal diagrams for empirical research. *Biometrika*, 82(4):669–688.
- Pecevski, D., Maass, W., Legenstein, R. A. (2008). Theoretical analysis of learning with reward-modulated spike-timing-dependent plasticity. In *Advances in Neural Information Processing Systems 20*, pp. 881–888. Curran Associates, Inc.
- Peng, J. (1993). *Efficient Dynamic Programming-Based Learning for Control*. PhD thesis, Northeastern University, Boston MA.

- Peng, J. (1995). Efficient memory-based dynamic programming. In *Proceedings of the 12th International Conference on Machine Learning*, pp. 438–446.
- Peng, J., Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4):437–454.
- Peng, J., Williams, R. J. (1994). Incremental multi-step Q-learning. In *Proceedings of the 11th International Conference on Machine Learning*, pp. 226–232. Morgan Kaufmann, San Francisco.
- Peng, J., Williams, R. J. (1996). Incremental multi-step Q-learning. *Machine Learning*, 22(1):283–290.
- Perkins, T. J., Pendrith, M. D. (2002). On the existence of fixed points for Q-learning and Sarsa in partially observable domains. In *Proceedings of the 19th International Conference on Machine Learning*, pp. 490–497.
- Perkins, T. J., Precup, D. (2003). A convergent form of approximate policy iteration. In *Advances in Neural Information Processing Systems 15*, pp. 1627–1634. MIT Press, Cambridge, MA.
- Peters, J., Büchel, C. (2010). Neural representations of subjective reward value. *Behavioral Brain Research*, 213(2):135–141.
- Peters, J., Schaal, S. (2008). Natural actor–critic. *Neurocomputing*, 71(7):1180–1190.
- Peters, J., Vijayakumar, S., Schaal, S. (2005). Natural actor–critic. In *European Conference on Machine Learning*, pp. 280–291. Springer Berlin Heidelberg.
- Pezzulo, G., van der Meer, M. A. A., Lansink, C. S., Pennartz, C. M. A. (2014). Internally generated sequences in learning and executing goal-directed behavior. *Trends in Cognitive Science*, 18(12):647–657.
- Pfeiffer, B. E., Foster, D. J. (2013). Hippocampal place-cell sequences depict future paths to remembered goals. *Nature*, 497(7447):74–79.
- Phansalkar, V. V., Thathachar, M. A. L. (1995). Local and global optimization algorithms for generalized learning automata. *Neural Computation*, 7(5):950–973.
- Poggio, T., Girosi, F. (1989). A theory of networks for approximation and learning. A.I. Memo 1140. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.
- Poggio, T., Girosi, F. (1990). Regularization algorithms for learning that are equivalent to multilayer networks. *Science*, 247(4945):978–982.
- Polyak, B. T. (1990). New stochastic approximation type procedures. *Automat. i Telemekh*, 7(98-107):2 (in Russian).
- Polyak, B. T., Juditsky, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855.
- Powell, M. J. D. (1987). Radial basis functions for multivariate interpolation: A review. In J. C. Mason and M. G. Cox (Eds.), *Algorithms for Approximation*, pp. 143–167. Clarendon Press, Oxford.
- Powell, W. B. (2011). *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, Second edition. John Wiley and Sons.
- Powers, W. T. (1973). *Behavior: The Control of Perception*. Aldine de Gruyter, Chicago. 2nd expanded edition 2005.
- Precup, D. (2000). *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst.
- Precup, D., Sutton, R. S., Dasgupta, S. (2001). Off-policy temporal-difference learning with function approximation. In *Proceedings of the 18th International Conference on Machine Learning*, pp. 417–424.

- Precup, D., Sutton, R. S., Paduraru, C., Koop, A., Singh, S. (2006). Off-policy learning with options and recognizers. In *Advances in Neural Information Processing Systems 18*, pp. 1097–1104. MIT Press, Cambridge, MA.
- Precup, D., Sutton, R. S., Singh, S. (2000). Eligibility traces for off-policy policy evaluation. In *Proceedings of the 17th International Conference on Machine Learning*, pp. 759–766. Morgan Kaufmann.
- Puterman, M. L. (1994). *Markov Decision Problems*. Wiley, New York.
- Puterman, M. L., Shin, M. C. (1978). Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24(11):1127–1137.
- Quartz, S., Dayan, P., Montague, P. R., Sejnowski, T. J. (1992). Expectation learning in the brain using diffuse ascending connections. In *Society for Neuroscience Abstracts*, 18:1210.
- Randløv, J., Alstrøm, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the 15th International Conference on Machine Learning*, pp. 463–471.
- Rangel, A., Camerer, C., Montague, P. R. (2008). A framework for studying the neurobiology of value-based decision making. *Nature Reviews Neuroscience*, 9(7):545–556.
- Rangel, A., Hare, T. (2010). Neural computations associated with goal-directed choice. *Current Opinion in Neurobiology*, 20(2):262–270.
- Rao, R. P., Sejnowski, T. J. (2001). Spike-timing-dependent Hebbian plasticity as temporal difference learning. *Neural Computation*, 13(10):2221–2237.
- Ratcliff, R. (1990). Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, 97(2):285–308.
- Reddy, G., Celani, A., Sejnowski, T. J., Vergassola, M. (2016). Learning to soar in turbulent environments. *Proceedings of the National Academy of Sciences*, 113(33):E4877–E4884.
- Redish, D. A. (2004). Addiction as a computational process gone awry. *Science*, 306(5703):1944–1947.
- Reetz, D. (1977). Approximate solutions of a discounted Markovian decision process. *Bonner Mathematische Schriften*, 98:77–92.
- Rescorla, R. A., Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement. In A. H. Black and W. F. Prokasy (Eds.), *Classical Conditioning II*, pp. 64–99. Appleton-Century-Crofts, New York.
- Revusky, S., Garcia, J. (1970). Learned associations over long delays. In G. Bower (Ed.), *The Psychology of Learning and Motivation*, v. 4, pp. 1–84. Academic Press, Inc., New York.
- Reynolds, J. N. J., Wickens, J. R. (2002). Dopamine-dependent plasticity of corticostriatal synapses. *Neural Networks*, 15(4):507–521.
- Ring, M. B. (in preparation). Representing knowledge as forecasts (and state as knowledge).
- Ripley, B. D. (2007). *Pattern Recognition and Neural Networks*. Cambridge University Press.
- Rixner, S. (2004). Memory controller optimizations for web servers. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, p. 355–366. IEEE Computer Society.
- Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535.
- Robertie, B. (1992). Carbon versus silicon: Matching wits with TD-Gammon. *Inside Backgammon*, 2(2):14–22.
- Romo, R., Schultz, W. (1990). Dopamine neurons of the monkey midbrain: Contingencies of responses to active touch during self-initiated arm movements. *Journal of Neurophysiology*, 63(3):592–624.

- Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington, DC.
- Ross, S. (1983). *Introduction to Stochastic Dynamic Programming*. Academic Press, New York.
- Ross, T. (1933). Machines that think. *Scientific American*, 148(4):206–208.
- Rubinstein, R. Y. (1981). *Simulation and the Monte Carlo Method*. Wiley, New York.
- Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. I, *Foundations*. Bradford/MIT Press, Cambridge, MA.
- Rummery, G. A. (1995). *Problem Solving with Reinforcement Learning*. PhD thesis, University of Cambridge.
- Rummery, G. A., Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University.
- Ruppert, D. (1988). Efficient estimations from a slowly convergent Robbins-Monro process. Cornell University Operations Research and Industrial Engineering Technical Report No. 781.
- Russell, S., Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice-Hall, Englewood Cliffs, NJ.
- Russo, D. J., Van Roy, B., Kazerouni, A., Osband, I., Wen, Z. (2018). A tutorial on Thompson sampling, *Foundations and Trends in Machine Learning*. ArXiv:1707.02038.
- Rust, J. (1996). Numerical dynamic programming in economics. In H. Amman, D. Kendrick, and J. Rust (Eds.), *Handbook of Computational Economics*, pp. 614–722. Elsevier, Amsterdam.
- Saddoris, M. P., Cacciapaglia, F., Wightman, R. M., Carelli, R. M. (2015). Differential dopamine release dynamics in the nucleus accumbens core and shell reveal complementary signals for error prediction and incentive motivation. *The Journal of Neuroscience*, 35(33):11572–11582.
- Saksida, L. M., Raymond, S. M., Touretzky, D. S. (1997). Shaping robot behavior using principles from instrumental conditioning. *Robotics and Autonomous Systems*, 22(3):231–249.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3(3), 210–229.
- Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II—Recent progress. *IBM Journal on Research and Development*, 11(6):601–617.
- Schaal, S., Atkeson, C. G. (1994). Robot juggling: Implementation of memory-based learning. *IEEE Control Systems*, 14(1):57–71.
- Schmajuk, N. A. (2008). Computational models of classical conditioning. *Scholarpedia*, 3(3):1664.
- Schmidhuber, J. (1991a). Curious model-building control systems. In *Proceedings of the IEEE International Joint Conference on Neural Networks*, pp. 1458–1463. IEEE.
- Schmidhuber, J. (1991b). A possibility for implementing curiosity and boredom in model-building neural controllers. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pp. 222–227. MIT Press, Cambridge, MA.
- Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 6:85–117.
- Schmidhuber, J., Storck, J., Hochreiter, S. (1994). Reinforcement driven information acquisition in nondeterministic environments. Technical report, Fakultät für Informatik, Technische Universität München, München, Germany.
- Schraudolph, N. N. (1999). Local gain adaptation in stochastic gradient descent. In *Proceedings of the International Conference on Artificial Neural Networks*, pp. 569–574. IEEE, London.

- Schraudolph, N. N. (2002). Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738.
- Schraudolph, N. N., Yu, J., Aberdeen, D. (2006). Fast online policy gradient learning with SMD gain vector adaptation. In *Advances in Neural Information Processing Systems*, pp. 1185–1192.
- Schulman, J., Chen, X., Abbeel, P. (2017). Equivalence between policy gradients and soft Q-Learning. ArXiv:1704.06440.
- Schultz, D. G., Melsa, J. L. (1967). *State Functions and Linear Control Systems*. McGraw-Hill, New York.
- Schultz, W. (1998). Predictive reward signal of dopamine neurons. *Journal of Neurophysiology*, 80(1):1–27.
- Schultz, W., Apicella, P., Ljungberg, T. (1993). Responses of monkey dopamine neurons to reward and conditioned stimuli during successive steps of learning a delayed response task. *The Journal of Neuroscience*, 13(3):900–913.
- Schultz, W., Dayan, P., Montague, P. R. (1997). A neural substrate of prediction and reward. *Science*, 275(5306):1593–1598.
- Schultz, W., Romo, R. (1990). Dopamine neurons of the monkey midbrain: contingencies of responses to stimuli eliciting immediate behavioral reactions. *Journal of Neurophysiology*, 63(3):607–624.
- Schultz, W., Romo, R., Ljungberg, T., Mirenowicz, J., Hollerman, J. R., Dickinson, A. (1995). Reward-related signals carried by dopamine neurons. In J. C. Houk, J. L. Davis, and D. G. Beiser (Eds.), *Models of Information Processing in the Basal Ganglia*, pp. 233–248. MIT Press, Cambridge, MA.
- Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the 10th International Conference on Machine Learning*, pp. 298–305. Morgan Kaufmann.
- Schweitzer, P. J., Seidmann, A. (1985). Generalized polynomial approximations in Markovian decision processes. *Journal of Mathematical Analysis and Applications*, 110(2):568–582.
- Selfridge, O. G. (1978). Tracking and trailing: Adaptation in movement strategies. Technical report, Bolt Beranek and Newman, Inc. Unpublished report.
- Selfridge, O. G. (1984). Some themes and primitives in ill-defined systems. In O. G. Selfridge, E. L. Rissland, and M. A. Arbib (Eds.), *Adaptive Control of Ill-Defined Systems*, pp. 21–26. Plenum Press, NY. Proceedings of the NATO Advanced Research Institute on Adaptive Control of Ill-defined Systems, NATO Conference Series II, Systems Science, Vol. 16.
- Selfridge, O. J., Sutton, R. S., Barto, A. G. (1985). Training and tracking in robotics. In A. Joshi (Ed.), *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 670–672. Morgan Kaufmann.
- Seo, H., Barraclough, D., Lee, D. (2007). Dynamic signals related to choices and outcomes in the dorsolateral prefrontal cortex. *Cerebral Cortex*, 17(suppl 1):110–117.
- Seung, H. S. (2003). Learning in spiking neural networks by reinforcement of stochastic synaptic transmission. *Neuron*, 40(6):1063–1073.
- Shah, A. (2012). Psychological and neuroscientific connections with reinforcement learning. In M. Wiering and M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 507–537. Springer-Verlag Berlin Heidelberg.
- Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine and Journal of Science*, 41(314):256–275.

- Shannon, C. E. (1951). Presentation of a maze-solving machine. In H. V. Forester (Ed.), *Cybernetics. Transactions of the Eighth Conference*, pp. 173–180. Josiah Macy Jr. Foundation.
- Shannon, C. E. (1952). “Theseus” maze-solving mouse. <http://cyberneticzoo.com/mazesolvers/1952—theseus-maze-solving-mouse—claude-shannon-american/>.
- Shelton, C. R. (2001). *Importance Sampling for Reinforcement Learning with Multiple Objectives*. PhD thesis, Massachusetts Institute of Technology, Cambridge MA.
- Shepard, D. (1968). A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 23rd ACM National Conference*, pp. 517–524. ACM, New York.
- Sherman, J., Morrison, W. J. (1949). Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix (abstract). *Annals of Mathematical Statistics*, 20(4):621.
- Shewchuk, J., Dean, T. (1990). Towards learning time-varying functions with high input dimensionality. In *Proceedings of the Fifth IEEE International Symposium on Intelligent Control*, pp. 383–388. IEEE Computer Society Press, Los Alamitos, CA.
- Shimansky, Y. P. (2009). Biologically plausible learning in neural networks: a lesson from bacterial chemotaxis. *Biological Cybernetics*, 101(5-6):379–385.
- Si, J., Barto, A., Powell, W., Wunsch, D. (Eds.) (2004). *Handbook of Learning and Approximate Dynamic Programming*. John Wiley and Sons.
- Silver, D. (2009). *Reinforcement Learning and Simulation Based Search in the Game of Go*. PhD thesis, University of Alberta, Edmonton.
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.
- Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning*, pp. 387–395.
- Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, L., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., Hassabis, D. (2017a). Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359.
- Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simoyan, K., Hassabis, D. (2017b). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. ArXiv:1712.01815.
- Şimşek, Ö., Algórta, S., Kothiyal, A. (2016). Why most decisions are easy in tetris—And perhaps in other sequential decision problems, as well. In *Proceedings of the 33rd International Conference on Machine Learning*, pp. 1757–1765.
- Simon, H. (2000). Lecture at the Earthware Symposium, Carnegie Mellon University. <https://www.youtube.com/watch?v=EZhyi-8DBJc>.
- Singh, S. P. (1992a). Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 202–207. AAAI/MIT Press, Menlo Park, CA.
- Singh, S. P. (1992b). Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *Proceedings of the 9th International Workshop on Machine Learning*, pp. 406–415. Morgan Kaufmann.
- Singh, S. P. (1993). *Learning to Solve Markovian Decision Processes*. PhD thesis, University of Massachusetts, Amherst.

- Singh, S. P. (Ed.) (2002). Special double issue on reinforcement learning, *Machine Learning*, 49(2-3).
- Singh, S., Barto, A. G., Chentanez, N. (2005). Intrinsically motivated reinforcement learning. In *Advances in Neural Information Processing Systems 17*, pp. 1281–1288. MIT Press, Cambridge, MA.
- Singh, S. P., Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems 9*, pp. 974–980. MIT Press, Cambridge, MA.
- Singh, S. P., Jaakkola, T., Jordan, M. I. (1994). Learning without state-estimation in partially observable Markovian decision problems. In *Proceedings of the 11th International Conference on Machine Learning*, pp. 284–292. Morgan Kaufmann.
- Singh, S., Jaakkola, T., Littman, M. L., Szepesvári, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308.
- Singh, S. P., Jaakkola, T., Jordan, M. I. (1995). Reinforcement learning with soft state aggregation. In *Advances in Neural Information Processing Systems 7*, pp. 359–368. MIT Press, Cambridge, MA.
- Singh, S., Lewis, R. L., Barto, A. G. (2009). Where do rewards come from? In N. Taatgen and H. van Rijn (Eds.), *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, pp. 2601–2606. Cognitive Science Society.
- Singh, S., Lewis, R. L., Barto, A. G., Sorg, J. (2010). Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2):70–82. Special issue on Active Learning and Intrinsically Motivated Exploration in Robots: Advances and Challenges.
- Singh, S. P., Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3):123–158.
- Skinner, B. F. (1938). *The Behavior of Organisms: An Experimental Analysis*. Appleton-Century, New York.
- Skinner, B. F. (1958). Reinforcement today. *American Psychologist*, 13(3):94–99.
- Skinner, B. F. (1963). Operant behavior. *American Psychologist*, 18(8):503–515.
- Sofge, D. A., White, D. A. (1992). Applied learning: Optimal control for manufacturing. In D. A. White and D. A. Sofge (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pp. 259–281. Van Nostrand Reinhold, New York.
- Sorg, J. D. (2011). *The Optimal Reward Problem: Designing Effective Reward for Bounded Agents*. PhD thesis, University of Michigan, Ann Arbor.
- Sorg, J., Lewis, R. L., Singh, S. P. (2010). Reward design via online gradient ascent. In *Advances in Neural Information Processing Systems 23*, pp. 2190–2198. Curran Associates, Inc.
- Sorg, J., Singh, S. (2010). Linear options. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, pp. 31–38.
- Sorg, J., Singh, S., Lewis, R. (2010). Internal rewards mitigate agent boundedness. In *Proceedings of the 27th International Conference on Machine Learning*, pp. 1007–1014.
- Spence, K. W. (1947). The role of secondary reinforcement in delayed reward learning. *Psychological Review*, 54(1):1–8.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.
- Staddon, J. E. R. (1983). *Adaptive Behavior and Learning*. Cambridge University Press.

- Stanfill, C., Waltz, D. (1986). Toward memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228.
- Steinberg, E. E., Keiflin, R., Boivin, J. R., Witten, I. B., Deisseroth, K., Janak, P. H. (2013). A causal link between prediction errors, dopamine neurons and learning. *Nature Neuroscience*, 16(7):966–973.
- Sterling, P., Laughlin, S. (2015). *Principles of Neural Design*. MIT Press, Cambridge, MA.
- Sternberg, S. (1963). Stochastic learning theory. In: *Handbook of Mathematical Psychology*, Volume II, R. D. Luce, R. R. Bush, and E. Galanter (Eds.). John Wiley & Sons.
- Sugiyama, M., Hachiya, H., Morimura, T. (2013). *Statistical Reinforcement Learning: Modern Machine Learning Approaches*. Chapman & Hall/CRC.
- Suri, R. E., Bargas, J., Arbib, M. A. (2001). Modeling functions of striatal dopamine modulation in learning and planning. *Neuroscience*, 103(1):65–85.
- Suri, R. E., Schultz, W. (1998). Learning of sequential movements by neural network model with dopamine-like reinforcement signal. *Experimental Brain Research*, 121(3):350–354.
- Suri, R. E., Schultz, W. (1999). A neural network model with dopamine-like reinforcement signal that learns a spatial delayed response task. *Neuroscience*, 91(3):871–890.
- Sutton, R. S. (1978a). Learning theory support for a single channel theory of the brain. Unpublished report.
- Sutton, R. S. (1978b). Single channel theory: A neuronal theory of learning. *Brain Theory Newsletter*, 4:72–75. Center for Systems Neuroscience, University of Massachusetts, Amherst, MA.
- Sutton, R. S. (1978c). *A unified theory of expectation in classical and instrumental conditioning*. Bachelors thesis, Stanford University.
- Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst.
- Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9–44 (important erratum p. 377).
- Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Workshop on Machine Learning*, pp. 216–224. Morgan Kaufmann.
- Sutton, R. S. (1991a). Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin*, 2(4):160–163. ACM, New York.
- Sutton, R. S. (1991b). Planning by incremental dynamic programming. In *Proceedings of the 8th International Workshop on Machine Learning*, pp. 353–357. Morgan Kaufmann.
- Sutton, R. S. (Ed.) (1992a). *Reinforcement Learning*. Kluwer Academic Press. Reprinting of a special double issue on reinforcement learning, *Machine Learning*, 8(3-4).
- Sutton, R. S. (1992b). Adapting bias by gradient descent: An incremental version of delta-bar-delta. *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 171–176, MIT Press.
- Sutton, R. S. (1992c). Gain adaptation beats least squares? *Proceedings of the Seventh Yale Workshop on Adaptive and Learning Systems*, pp. 161–166, Yale University, New Haven, CT.
- Sutton, R. S. (1995a). TD models: Modeling the world at a mixture of time scales. In *Proceedings of the 12th International Conference on Machine Learning*, pp. 531–539. Morgan Kaufmann.
- Sutton, R. S. (1995b). On the virtues of linear learning and trajectory distributions. In *Proceedings of the Workshop on Value Function Approximation at The 12th International Conference on Machine Learning*.

- Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pp. 1038–1044. MIT Press, Cambridge, MA.
- Sutton, R. S. (2009). The grand challenge of predictive empirical abstract knowledge. *Working Notes of the IJCAI-09 Workshop on Grand Challenges for Reasoning from Experiences*.
- Sutton, R. S. (2015a) Introduction to reinforcement learning with function approximation. Tutorial at the Conference on Neural Information Processing Systems, Montreal, December 7, 2015.
- Sutton, R. S. (2015b) True online Emphatic TD(λ): Quick reference and implementation guide. ArXiv:1507.07147. Code is available in Python and C++ by downloading the source files of this arXiv paper as a zip archive.
- Sutton, R. S., Barto, A. G. (1981a). Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review*, 88(2):135–170.
- Sutton, R. S., Barto, A. G. (1981b). An adaptive network that constructs and uses an internal model of its world. *Cognition and Brain Theory*, 3:217–246.
- Sutton, R. S., Barto, A. G. (1987). A temporal-difference model of classical conditioning. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, pp. 355–378. Erlbaum, Hillsdale, NJ.
- Sutton, R. S., Barto, A. G. (1990). Time-derivative models of Pavlovian reinforcement. In M. Gabriel and J. Moore (Eds.), *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pp. 497–537. MIT Press, Cambridge, MA.
- Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, Cs., Wiewiora, E. (2009a). Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th International Conference on Machine Learning*, pp. 993–1000. ACM, New York.
- Sutton, R. S., Szepesvári, Cs., Maei, H. R. (2009b). A convergent $O(d^2)$ temporal-difference algorithm for off-policy learning with linear function approximation. In *Advances in Neural Information Processing Systems 21*, pp. 1609–1616. Curran Associates, Inc.
- Sutton, R. S., Mahmood, A. R., Precup, D., van Hasselt, H. (2014). A new Q(λ) with interim forward view and Monte Carlo equivalence. In *Proceedings of the International Conference on Machine Learning*, 31. *JMLR W&CP* 32(2).
- Sutton, R. S., Mahmood, A. R., White, M. (2016). An emphatic approach to the problem of off-policy temporal-difference learning. *Journal of Machine Learning Research*, 17(73):1–29.
- Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, pp. 1057–1063. MIT Press, Cambridge, MA.
- Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems*, pp. 761–768, Taipei, Taiwan.
- Sutton, R. S., Pinette, B. (1985). The learning of world models by connectionist networks. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pp. 54–64.
- Sutton, R. S., Precup, D., Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211.
- Sutton, R. S., Rafols, E., Koop, A. (2006). Temporal abstraction in temporal-difference networks. In *Advances in neural information processing systems*, pp. 1313–1320.
- Sutton, R. S., Singh, S. P., McAllester, D. A. (2000). Comparing policy-gradient algorithms. Unpublished manuscript.

- Sutton, R. S., Szepesvári, Cs., Geramifard, A., Bowling, M., (2008). Dyna-style planning with linear function approximation and prioritized sweeping. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pp. 528–536.
- Sutton, R. S., Tanner, B. (2005). Temporal-difference networks. In *Advances in Neural Information Processing Systems 17*, p. 1377–1384.
- Szepesvári, Cs. (2010). Algorithms for reinforcement learning. In *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103. Morgan and Claypool.
- Szita, I. (2012). Reinforcement learning in games. In M. Wiering and M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 539–577. Springer-Verlag Berlin Heidelberg.
- Tadepalli, P., Ok, D. (1994). H-learning: A reinforcement learning method to optimize undiscounted average reward. Technical Report 94-30-01. Oregon State University, Computer Science Department, Corvallis.
- Tadepalli, P., Ok, D. (1996). Scaling up average reward reinforcement learning by approximating the domain models and the value function. In *Proceedings of the 13th International Conference on Machine Learning*, pp. 471–479.
- Takahashi, Y., Schoenbaum, G., and Niv, Y. (2008). Silencing the critics: Understanding the effects of cocaine sensitization on dorsolateral and ventral striatum in the context of an actor/critic model. *Frontiers in Neuroscience*, 2(1):86–99.
- Tambe, M., Newell, A., Rosenbloom, P. S. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5(3):299–348.
- Tan, M. (1991). Learning a cost-sensitive internal representation for reinforcement learning. In L. A. Birnbaum and G. C. Collins (Eds.), *Proceedings of the 8th International Workshop on Machine Learning*, pp. 358–362. Morgan Kaufmann.
- Tanner, B. (2006). Temporal-Difference Networks. MSc thesis, University of Alberta.
- Taylor, G., Parr, R. (2009). Kernelized value function approximation for reinforcement learning. In *Proceedings of the 26th International Conference on Machine Learning*, pp. 1017–1024. ACM, New York.
- Taylor, M. E., Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10:1633–1685.
- Tesauro, G. (1986). Simple neural models of classical conditioning. *Biological Cybernetics*, 55(2-3):187–200.
- Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8(3-4):257–277.
- Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219.
- Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68.
- Tesauro, G. (2002). Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199.
- Tesauro, G., Galperin, G. R. (1997). On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing Systems 9*, pp. 1068–1074. MIT Press, Cambridge, MA.
- Tesauro, G., Gondek, D. C., Lechner, J., Fan, J., Prager, J. M. (2012). Simulation, learning, and optimization techniques in Watson’s game strategies. *IBM Journal of Research and Development*, 56(3-4):16–1–16–11.
- Tesauro, G., Gondek, D. C., Lenchner, J., Fan, J., Prager, J. M. (2013). Analysis of WATSON’s strategies for playing Jeopardy! *Journal of Artificial Intelligence Research*, 47:205–251.
- Tham, C. K. (1994). *Modular On-Line Function Approximation for Scaling up Reinforcement Learning*. PhD thesis, University of Cambridge.

- Thathachar, M. A. L., Sastry, P. S. (1985). A new approach to the design of reinforcement schemes for learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15(1):168–175.
- Thathachar, M., Sastry, P. S. (2002). Varieties of learning automata: an overview. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 36(6):711–722.
- Thathachar, M., Sastry, P. S. (2011). *Networks of Learning Automata: Techniques for Online Stochastic Optimization*. Springer Science & Business Media.
- Theocharous, G., Thomas, P. S., Ghavamzadeh, M. (2015). Personalized ad recommendation for life-time value optimization guarantees. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*. AAAI Press, Palo Alto, CA.
- Thistlthwaite, D. (1951). A critical review of latent learning and related experiments. *Psychological Bulletin*, 48(2):97–129.
- Thomas, P. S. (2014). Bias in natural actor–critic algorithms. In *Proceedings of the 31st International Conference on Machine Learning, JMLR W&CP* 32(1), pp. 441–448.
- Thomas, P. S. (2015). *Safe Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst.
- Thomas, P. S., Brunskill, E. (2017). Policy gradient methods for reinforcement learning with function approximation and action-dependent baselines. ArXiv:1706.06643.
- Thomas, P. S., Theocharous, G., Ghavamzadeh, M. (2015). High-confidence off-policy evaluation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pp. 3000–3006. AAAI Press, Menlo Park, CA.
- Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294.
- Thompson, W. R. (1934). On the theory of apportionment. *American Journal of Mathematics*, 57: 450–457.
- Thon, M. (2017). *Spectral Learning of Sequential Systems*. PhD thesis, Jacobs University Bremen.
- Thon, M., Jaeger, H. (2015). Links between multiplicity automata, observable operator models and predictive state representations: a unified learning framework. *The Journal of Machine Learning Research*, 16(1):103–147.
- Thorndike, E. L. (1898). Animal intelligence: An experimental study of the associative processes in animals. *The Psychological Review, Series of Monograph Supplements*, II(4).
- Thorndike, E. L. (1911). *Animal Intelligence*. Hafner, Darien, CT.
- Thorp, E. O. (1966). *Beat the Dealer: A Winning Strategy for the Game of Twenty-One*. Random House, New York.
- Tian, T. (in preparation) *An Empirical Study of Sliding-Step Methods in Temporal Difference Learning*. M.Sc thesis, University of Alberta, Edmonton.
- Tieleman, T., Hinton, G. (2012). Lecture 6.5–RMSProp. COURSERA: Neural networks for machine learning 4.2:26–31.
- Tolman, E. C. (1932). *Purposive Behavior in Animals and Men*. Century, New York.
- Tolman, E. C. (1948). Cognitive maps in rats and men. *Psychological Review*, 55(4):189–208.
- Tsai, H.-S., Zhang, F., Adamantidis, A., Stuber, G. D., Bonci, A., de Lecea, L., Deisseroth, K. (2009). Phasic firing in dopaminergic neurons is sufficient for behavioral conditioning. *Science*, 324(5930):1080–1084.
- Tsetlin, M. L. (1973). *Automaton Theory and Modeling of Biological Systems*. Academic Press, New York.

- Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and Q-learning. *Machine Learning*, 16(3):185–202.
- Tsitsiklis, J. N. (2002). On the convergence of optimistic policy iteration. *Journal of Machine Learning Research*, 3:59–72.
- Tsitsiklis, J. N., Van Roy, B. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1-3):59–94.
- Tsitsiklis, J. N., Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690.
- Tsitsiklis, J. N., Van Roy, B. (1999). Average cost temporal-difference learning. *Automatica*, 35(11):1799–1808.
- Turing, A. M. (1948). Intelligent machinery. In B. Jack Copeland (Ed.) (2004), *The Essential Turing*, pp. 410–432. Oxford University Press, Oxford.
- Ungar, L. H. (1990). A bioreactor benchmark for adaptive network-based process control. In W. T. Miller, R. S. Sutton, and P. J. Werbos (Eds.), *Neural Networks for Control*, pp. 387–402. MIT Press, Cambridge, MA.
- Unnikrishnan, K. P., Venugopal, K. P. (1994). Alopex: A correlation-based learning algorithm for feedforward and recurrent neural networks. *Neural Computation*, 6(3): 469–490.
- Urbanczik, R., Senn, W. (2009). Reinforcement learning in populations of spiking neurons. *Nature neuroscience*, 12(3):250–252.
- Urbanowicz, R. J., Moore, J. H. (2009). Learning classifier systems: A complete introduction, review, and roadmap. *Journal of Artificial Evolution and Applications*. 10.1155/2009/736398.
- Valentin, V. V., Dickinson, A., O'Doherty, J. P. (2007). Determining the neural substrates of goal-directed learning in the human brain. *The Journal of Neuroscience*, 27(15):4019–4026.
- van Hasselt, H. (2010). Double Q-learning. In *Advances in Neural Information Processing Systems 23*, pp. 2613–2621. Curran Associates, Inc.
- van Hasselt, H. (2011). *Insights in Reinforcement Learning: Formal Analysis and Empirical Evaluation of Temporal-difference Learning*. SIKS dissertation series number 2011-04.
- van Hasselt, H. (2012). Reinforcement learning in continuous state and action spaces. In M. Wiering and M. van Otterlo (Eds.), *Reinforcement Learning: State-of-the-Art*, pp. 207–251. Springer-Verlag Berlin Heidelberg.
- van Hasselt, H., Sutton, R. S. (2015). Learning to predict independent of span. ArXiv:1508.04582.
- Van Roy, B., Bertsekas, D. P., Lee, Y., Tsitsiklis, J. N. (1997). A neuro-dynamic programming approach to retailer inventory management. In *Proceedings of the 36th IEEE Conference on Decision and Control*, Vol. 4, pp. 4052–4057.
- van Seijen, H. (2011). Reinforcement Learning under Space and Time Constraints. University of Amsterdam PhD thesis. Hague: TNO.
- van Seijen, H. (2016). Effective multi-step temporal-difference learning for non-linear function approximation. ArXiv:1608.05151.
- van Seijen, H., Sutton, R. S. (2013). Efficient planning in MDPs by small backups. In: *Proceedings of the 30th International Conference on Machine Learning*, pp. 361–369.
- van Seijen, H., Sutton, R. S. (2014). True online TD(λ). In *Proceedings of the 31st International Conference on Machine Learning*, pp. 692–700. JMLR W&CP 32(1),
- van Seijen, H., Mahmood, A. R., Pilarski, P. M., Machado, M. C., Sutton, R. S. (2016). True online temporal-difference learning. *Journal of Machine Learning Research*, 17(145):1–40.
- van Seijen, H., Van Hasselt, H., Whiteson, S., Wiering, M. (2009). A theoretical and empirical analysis of Expected Sarsa. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pp. 177–184.

- van Seijen, H., Whiteson, S., van Hasselt, H., Wiering, M. (2011). Exploiting best-match equations for efficient reinforcement learning. *Journal of Machine Learning Research* 12:2045–2094.
- Varga, R. S. (1962). *Matrix Iterative Analysis*. Englewood Cliffs, NJ: Prentice-Hall.
- Vasilaki, E., Frémaux, N., Urbanczik, R., Senn, W., Gerstner, W. (2009). Spike-based reinforcement learning in continuous state and action space: when policy gradient methods fail. *PLoS Computational Biology*, 5(12).
- Viswanathan, R., Narendra, K. S. (1974). Games of stochastic automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 4(1):131–135.
- Wagner, A. R. (2008). Evolution of an elemental theory of Pavlovian conditioning. *Learning & Behavior*, 36(3):253–265.
- Walter, W. G. (1950). An imitation of life. *Scientific American*, 182(5):42–45.
- Walter, W. G. (1951). A machine that learns. *Scientific American*, 185(2):60–63.
- Waltz, M. D., Fu, K. S. (1965). A heuristic approach to reinforcement learning control systems. *IEEE Transactions on Automatic Control*, 10(4):390–398.
- Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, University of Cambridge.
- Watkins, C. J. C. H., Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4):279–292.
- Werbos, P. J. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, 22(12):25–38.
- Werbos, P. J. (1982). Applications of advances in nonlinear sensitivity analysis. In R. F. Drenick and F. Kozin (Eds.), *System Modeling and Optimization*, pp. 762–770. Springer-Verlag.
- Werbos, P. J. (1987). Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 17(1):7–20.
- Werbos, P. J. (1988). Generalization of back propagation with applications to a recurrent gas market model. *Neural Networks*, 1(4):339–356.
- Werbos, P. J. (1989). Neural networks for control and system identification. In *Proceedings of the 28th Conference on Decision and Control*, pp. 260–265. IEEE Control Systems Society.
- Werbos, P. J. (1992). Approximate dynamic programming for real-time control and neural modeling. In D. A. White and D. A. Sofge (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, pp. 493–525. Van Nostrand Reinhold, New York.
- Werbos, P. J. (1994). *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting* (Vol. 1). John Wiley and Sons.
- Wiering, M., Van Otterlo, M. (2012). *Reinforcement Learning: State-of-the-Art*. Springer-Verlag Berlin Heidelberg.
- White, A. (2015). *Developing a Predictive Approach to Knowledge*. PhD thesis, University of Alberta, Edmonton.
- White, D. J. (1969). *Dynamic Programming*. Holden-Day, San Francisco.
- White, D. J. (1985). Real applications of Markov decision processes. *Interfaces*, 15(6):73–83.
- White, D. J. (1988). Further real applications of Markov decision processes. *Interfaces*, 18(5):55–61.
- White, D. J. (1993). A survey of applications of Markov decision processes. *Journal of the Operational Research Society*, 44(11):1073–1096.
- White, A., White, M. (2016). Investigating practical linear temporal difference learning. In *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*, pp. 494–502.

- Whitehead, S. D., Ballard, D. H. (1991). Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83.
- Whitt, W. (1978). Approximations of dynamic programs I. *Mathematics of Operations Research*, 3(3):231–243.
- Whittle, P. (1982). *Optimization over Time*, vol. 1. Wiley, New York.
- Whittle, P. (1983). *Optimization over Time*, vol. 2. Wiley, New York.
- Wickens, J., Kötter, R. (1995). Cellular models of reinforcement. In J. C. Houk, J. L. Davis and D. G. Beiser (Eds.), *Models of Information Processing in the Basal Ganglia*, pp. 187–214. MIT Press, Cambridge, MA.
- Widrow, B., Gupta, N. K., Maitra, S. (1973). Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 3(5):455–465.
- Widrow, B., Hoff, M. E. (1960). Adaptive switching circuits. In *1960 WESCON Convention Record Part IV*, pp. 96–104. Institute of Radio Engineers, New York. Reprinted in J. A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, pp. 126–134. MIT Press, Cambridge, MA, 1988.
- Widrow, B., Smith, F. W. (1964). Pattern-recognizing control systems. In J. T. Tou and R. H. Wilcox (Eds.), *Computer and Information Sciences*, pp. 288–317. Spartan, Washington, DC.
- Widrow, B., Stearns, S. D. (1985). *Adaptive Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ.
- Wiener, N. (1964). *God and Golem, Inc: A Comment on Certain Points where Cybernetics Impinges on Religion*. MIT Press, Cambridge, MA.
- Wiewiora, E. (2003). Potential-based shaping and Q-value initialization are equivalent. *Journal of Artificial Intelligence Research*, 19:205–208.
- Williams, R. J. (1986). Reinforcement learning in connectionist networks: A mathematical analysis. Technical Report ICS 8605. Institute for Cognitive Science, University of California at San Diego, La Jolla.
- Williams, R. J. (1987). Reinforcement-learning connectionist systems. Technical Report NU-CCS-87-3. College of Computer Science, Northeastern University, Boston.
- Williams, R. J. (1988). On the use of backpropagation in associative reinforcement learning. In *Proceedings of the IEEE International Conference on Neural Networks*, pp. I-263–I-270. IEEE San Diego section and IEEE TAB Neural Network Committee.
- Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256.
- Williams, R. J., Baird, L. C. (1990). A mathematical analysis of actor–critic architectures for learning optimal controls through incremental dynamic programming. In *Proceedings of the Sixth Yale Workshop on Adaptive and Learning Systems*, pp. 96–101. Center for Systems Science, Dunham Laboratory, Yale University, New Haven.
- Wilson, R. C., Takahashi, Y. K., Schoenbaum, G., Niv, Y. (2014). Orbitofrontal cortex as a cognitive map of task space. *Neuron*, 81(2):267–279.
- Wilson, S. W. (1994). ZCS, A zeroth order classifier system. *Evolutionary Computation*, 2(1):1–18.
- Wise, R. A. (2004). Dopamine, learning, and motivation. *Nature Reviews Neuroscience*, 5(6):1–12.
- Witten, I. H. (1976a). Learning to Control. University of Essex PhD thesis.
- Witten, I. H. (1976b). The apparent conflict between estimation and control—A survey of the two-armed problem. *Journal of the Franklin Institute*, 301(1-2):161–189.

- Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control*, 34(4):286–295.
- Witten, I. H., Corbin, M. J. (1973). Human operators and automatic adaptive controllers: A comparative study on a particular control task. *International Journal of Man-Machine Studies*, 5(1):75–104.
- Woodbury, T., Dunn, C., and Valasek, J. (2014). Autonomous soaring using reinforcement learning for trajectory generation. In *52nd Aerospace Sciences Meeting*, p. 0990.
- Woodworth, R. S. (1938). *Experimental Psychology*. New York: Henry Holt and Company.
- Xie, X., Seung, H. S. (2004). Learning in neural networks by reinforcement of irregular spiking. *Physical Review E*, 69(4):041909.
- Xu, X., Xie, T., Hu, D., Lu, X. (2005). Kernel least-squares temporal difference learning. *International Journal of Information Technology*, 11(9):54–63.
- Yagishita, S., Hayashi-Takagi, A., Ellis-Davies, G. C. R., Urakubo, H., Ishii, S., Kasai, H. (2014). A critical time window for dopamine actions on the structural plasticity of dendritic spines. *Science*, 345(6204):1616–1619.
- Yee, R. C., Saxena, S., Utgoff, P. E., Barto, A. G. (1990). Explaining temporal differences to create useful concepts for evaluating states. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 882–888. AAAI Press, Menlo Park, CA.
- Yin, H. H., Knowlton, B. J. (2006). The role of the basal ganglia in habit formation. *Nature Reviews Neuroscience*, 7(6):464–476.
- Young, P. (1984). *Recursive Estimation and Time-Series Analysis*. Springer-Verlag, Berlin.
- Yu, H. (2010). Convergence of least squares temporal difference methods under general conditions. *International Conference on Machine Learning* 27, pp. 1207–1214.
- Yu, H. (2012). Least squares temporal difference methods: An analysis under general conditions. *SIAM Journal on Control and Optimization*, 50(6):3310–3343.
- Yu, H. (2015). On convergence of emphatic temporal-difference learning. In *Proceedings of the 28th Annual Conference on Learning Theory, JMLR W&CP 40*. Also ArXiv:1506.02582.
- Yu, H. (2016). Weak convergence properties of constrained emphatic temporal-difference learning with constant and slowly diminishing stepsize. *Journal of Machine Learning Research*, 17(220):1–58.
- Yu, H. (2017). On convergence of some gradient-based temporal-differences algorithms for off-policy learning. ArXiv:1712.09652.
- Yu, H., Mahmood, A. R., Sutton, R. S. (2017). On generalized bellman equations and temporal-difference learning. ArXiv:17041.04463. A summary appeared in *Proceedings of the Canadian Conference on Artificial Intelligence*, pp. 3–14. Springer.

Index

Page numbers in *italics* are recommended to be consulted first. Page numbers in **bold** contain boxed algorithms.

- k -armed bandits, 25–45
- absorbing state, 57
- access-control queuing example, 256
- action preferences, 322, 329, 336, 455
 - in bandit problems, 37, 42
- action-value function, *see* value function, action
- action-value methods, 321
 - for bandit problems, 27
- actor–critic, 21, 239, 321, 331–332, 338, 406
 - advantage, A2C, 338
 - one-step (episodic), **332**
 - with eligibility traces (episodic), **332**
 - with eligibility traces (continuing), **333**
 - neural, 395–415
- addiction, 409–410
- afterstates, 137, 140, 181, 182, 191, 424, 430
- agent–environment interface, 47–58, 466
- all-actions algorithm, 326
- AlphaGo, AlphaGo Zero, AlphaZero, 441–450
- Andreae, John, 17, 21, 69, 89
- ANN, *see* artificial neural networks
- applications and case studies, 421–457
- approximate dynamic programming, 15
- artificial intelligence, xvii, 1, 472, *478*
- artificial neural networks, 223–228, 238–240, 395–398, 423, 430, 436–450, 472
- associative reinforcement learning, 45, 418
- associative search, 41
- asynchronous dynamic programming, 85, 88
- Atari video game play, 436–441
- auxiliary tasks, 460–461, 468, 474
- average reward setting, 249–255, 258, 464
- averagers, 264
- backgammon, 11, 21, 182, 184, 421–426
- backpropagation, 21, 225–227, 239, 407, 424, 436, 439
- backup diagram, 60, 139
 - for dynamic programming, 59, 61, 64, **172**
 - for Monte Carlo methods, 94
 - for Q-learning, 134
 - for TD(0), 121
- for Sarsa, 129
- for Expected Sarsa, 134
- for Sarsa(λ), 304
- for TD(λ), 289
- for Q(λ), 313
- for Tree Backup(λ), 314
- for Truncated TD(λ), 296
- for n -step $Q(\sigma)$, 155
- for n -step Expected Sarsa, 146
- for n -step Sarsa, 146
- for n -step TD, 142
- for n -step Tree Backup, 152
- for Samuel’s Checker Player, 428
- compound, 288
- half backups, 62
- backward view of eligibility traces, 288, *293*
- Baird’s counterexample, 261–264, 280, 283, 285
- bandit algorithm, simple, **32**
- bandit problems, 25–45
- basal ganglia, 386
- baseline, 37–40, 329, 330, 338
- behavior policy, 103, 110, *see* off-policy learning
- Bellman equation, 14
 - for v_π , 59
 - for q_π , 78
 - for optimal value functions: v_* and q_* , 63
- differential, 250
- for options, 463
- Bellman error, 268, 270, 272, 273
 - learnability of, 274–278
 - vector, 267–269
- Bellman operator, 267–269, 286
- Bellman residual, 286, *see* Bellman error
- Bellman, Richard, 14, 71, 89, 241
- binary features, 215, 222, 245, 304, 305
- bioreactor example, 51
- blackjack example, 93–94, 99, 106
- blocking maze example, 166
- bootstrapping, 89, 189, 308
 - n -step, 141–158, 255
- and dynamic programming, 89
- and function approximation, 208, 264–274

- and Monte Carlo methods, 95
 and stability, 263–265
 and TD learning, 120
 assessment of, 124–128, 248, 264, 291, 318
 in psychology, 345, 349, 354, 355
 parameter (λ or n), 291, 307, 399
BOXES, 18, 71, 237
 branching factor, 173–177, 422
 breakfast example, 5, 22
 bucket-brigade algorithm, 19, 21, 139
- catastrophic interference, 472
 certainty-equivalence estimate, 128
 chess, 4, 20, 54, 182, 450
 classical conditioning, 20, 343–357
 blocking, 371
 and higher-order conditioning, 345–355
 delay and trace conditioning, 344
 Rescorla-Wagner model, 346–349
 TD model, 349–357
 classifier systems, 19, 21
 cliff walking example, 132, 133
 CMAC, *see* tile coding
 coarse coding, 215–220, 238
 cognitive maps, 363–364
 collective reinforcement learning, 404–407
 complex backups, *see* compound update
 compound stimulus, 345, 346–356, 371, 382
 compound update/backup, 288, 319
 conditioned/unconditioned stimulus, conditioned response (CS/US, CR), 344
 constant- α MC, 120
 contextual bandits, 41
 continuing tasks, 54, 57, 70, 124, 249, 294
 continuous action, 73, 244, 335–336
 continuous state, 73, 223, 238
 continuous time, 11, 71
 control and prediction, 342
 control theory, 4, 71
 control variates, 150–152, 155, 281
 and eligibility traces, 309–312
 credit assignment, 11, 17, 19, 47, 294, 401
 in psychology, 346, 361
 structural, 385, 405, 407
 critic, 18, 239, 346, 417, *see also* actor–critic
 cumulant, 459
 curiosity, 474
 curse of dimensionality, 4, 14, 221, 231
 cybernetics, xvii, 477
- deadly triad, 264
 deep learning, 12, 223, 441, 472–474, 479
 deep reinforcement learning, 236
 deep residual learning, 227
 delayed reinforcement, 361–363
 delayed reward, 1, 47, 249
 dimensions of reinforcement learning methods, 189–191
 direct and indirect RL, 162, 164, 192
 discounting, 55, 199, 243, 249, 282, 324, 328, 427, 459
 in pole balancing, 56
 state dependent, 307
 deprecated, 253, 256
 distribution models, 159, 185
 dopamine, 377, 381–387, 413–419
 and addiction, 409–410
 double learning, 134–136, 140
 DP, *see* dynamic programming
 driving-home example, 122–123
 Dyna architecture, 164, 161–170
 dynamic programming, 13–15, 73–90, 174, 262
 and artificial intelligence, 89
 and function approximation, 241
 and options, 463
 and the deadly triad, 264
 computational efficiency of, 87
- eligibility traces, 287–320, 350, 362, 398–403
 accumulating, 300, 306, 310
 replacing, 301, 306
 dutch, 300–303
 contingent/non-contingent, 399–403, 411
 off-policy, 309–316
 with state-dependent λ and γ , 309–316
- Emphatic-TD methods, 234–235, 315
 off-policy, 281–282
- environment, 47–58
 episodes, episodic tasks, 11, 54–58, 91
 error reduction property, 144, 288
 evaluative feedback, 17, 25, 47
 evolution, 7, 359, 374, 471
 evolutionary methods, 7, 8, 9, 11, 19
 expected approximate value, 148, 155
 Expected Sarsa, 133, *see also* Sarsa, Expected
 expected update, 75, 172–181, 189
 experience replay, 440–441
 explore/exploit dilemma, 3, 103, 472
 exploring starts, 96, 98–100, 178

- feature construction, 210–223
final time step (T), 54
Fourier basis, 211–215
function approximation, 195–200
- gambler’s example, 84
game theory, 19
gazelle calf example, 5
general value functions (GVFs), 459–463, 474
generalized policy iteration (GPI), 86–87, 92, 97, 138, 189
genetic algorithms, 19
Gittins index, 43
gliding/soaring case study, 453–457
goal, *see* reward signal
golf example, 61, 63, 66
gradient, 201
gradient descent, *see* stochastic gradient descent
Gradient-TD methods, 278–281, 314–315
greedy or ε -greedy
as exploiting, 26–28
as shortsighted, 64
 ε -greedy policies, 100
gridworld examples, 60, 65, 76, 147
cliff walking, 132
Dyna blocking maze, 166
Dyna maze, 164
Dyna shortcut maze, 167
windy, 130, 131
- habitual and goal-directed control, 364–368
hedonistic neurons, 402–404
heuristic search, 181–183, 190
as sequences of backups, 183
in Samuel’s checkers player, 426
in TD-Gammon, 425
history of reinforcement learning, 13–22
Holland, John, 19, 21, 44, 139, 241
Hull, Clark, 16, 359, 360, 362–363
- importance sampling, 103–117, 151, 257
ratio, 104, 148, 258
weighted and ordinary, 105, 106
and eligibility traces, 309–312
and infinite variance, 106
discounting aware, 112–113
incremental implementation, 109
per-decision, 114–115
- n -step, 148–156
incremental implementation
of averages, 30–33
of weighted averages, 109
instrumental conditioning, 357–361, *see also*
Law of Effect
and motivation, 360–361
Thorndike’s puzzle boxes, 358
interest and emphasis, 234–235, 282, 316
inverse reinforcement learning, 470
- Jack’s car rental example, 81–82, 137, 210
- kernel-based function approximation, 232–233
Klopf, A. Harry, xv, xvii, 19–21, 402–404, 411
- latent learning, 192, 363, 366
Law of Effect, 15–16, 45, 343, 358–361, 417
learning automata, 18
Least Mean Square (LMS) algorithm, 279, 301
Least-Squares TD (LSTD), 228–229
linear function approx., 204–209, 266–269
linear programming, 87, 90
local and global optima, 200
- Markov decision process (MDP), 2, 14, 47–71
Markov property, 49, 115, 465–468
Markov reward process (MRP), 125
maximization bias, 134–136
maximum-likelihood estimate, 128
MC, *see* Monte Carlo methods
Mean Square
Bellman Error, \overline{BE} , 268
Projected Bellman Error, $\overline{PB\overline{E}}$, 269
Return Error, \overline{RE} , 275
TD Error, \overline{TDE} , 270
Value Error, \overline{VE} , 199–200
memory-based function approx., 230–232
Michie, Donald, 17, 71, 117
Minsky, Marvin, 16, 17, 20, 89
model of the environment, 7, 159
model-based and model-free methods, 7, 159
in animal learning, 363–368
model-based reinforcement learning, 159–193
in neuroscience, 407–409
Monte Carlo methods, 91–117
first- and every-visit MC, 92
first-visit MC control, 101
first-visit MC prediction, 92

- gradient method for v_π , **202**
 Monte Carlo ES (Exploring Starts), **99**
 off-policy control, **111**, 110–112
 off-policy prediction, 103–109, **110**
 Monte Carlo Tree Search (MCTS), 185–188
 motivation, 360–361
 mountain car example, 244–248, 305, 306
 multi-armed bandits, 25–45
- n*-step methods, 141–158
 $Q(\sigma)$, **156**
 Sarsa, **147**, **247**
 differential, **255**
 off-policy, **149**
 TD, **144**
 Tree Backup, **154**
 truncated λ -return, 295
 naughts and crosses, *see* tic-tac-toe
 neural networks, *see* artificial neural networks
 neurodynamic programming, 15
 neuroeconomics, 413, 419
 neuroscience, 4, 21, 377–419
 nonstationarity, 30, 32–36, 44, 255
 inherent, 91, 198
 notation, xiii, *xix*
- observations, 464
 off-policy methods, 257–286
 vs on-policy methods, 100, 103
 Monte Carlo, 103–115
 Q-learning, **131**
 Expected Sarsa, 133–134
 n-step, 148–156
 n-step $Q(\sigma)$, **156**
 n-step Sarsa, **149**
 n-step Tree Backup, **154**
 and eligibility traces, 309–316
 Emphatic-TD(λ), 315
 GQ(λ), 315
 GTD(λ), 314
 HTD(λ), 315
 Q(λ), 312–314
 Tree Backup(λ), 312–314
 reducing variance, 283–284
 on-policy distribution, 175, 199, 208, 258, 262, 281, 282
 vs uniform distribution, 176
 on-policy methods, 100
 actor-critic, **332**, **333**
- approximate
 control, **244**, **247**, **251**, **255**
 prediction, **202**, **203**, **209**
 Monte Carlo, 101, 100–103, **328**, **330**
n-step, **144**, **147**
 Sarsa, **130**, 129–131
 TD(0), **120**, 119–128
 with eligibility traces, **293**, **300**, **305**, **307**
 operant conditioning, *see* instrumental learning
 optimal control, 2, 14–15, 21
 optimistic initial values, 34–35, 192
 optimizing memory control, 432–436
 options, 461–464
 models of, 462
- pain and pleasure, 6, 16, 413
 Partially Observable MDPs (POMDPs), **467**
 Pavlov, Ivan, 16, 343–345, 362
 Pavlovian
 conditioning, *see* classical conditioning
 control, 343, 371, 373, 478
 personalizing web services, 450–453
 planning, 3, 5, 7, 11, 138, 159–193
 in psychology, 363, 364, 366
 with learned models, 161–168, 473
 with options, 461, 463
- policy, 6, 41, 58
 hierarchical, 462
 soft and ϵ -soft, 100–103, 110
 policy approximation, 321–324
 policy evaluation, 74–76, *see also* prediction
 iterative, **75**
 policy gradient methods, 321–338
 REINFORCE, **328**, **330**
 actor-critic, **332**, **333**
 policy gradient theorem, 324–326
 proof, episodic case, 325
 proof, continuing case, 334
 policy improvement, 76–80
 theorem, 78, 101
 policy iteration, 14, **80**, 80–82
 polynomial basis, 210–211
 prediction, 74–76, *see also* policy evaluation
 and control, 342
 Monte Carlo, 92–97
 off-policy, 103–108
 TD, 119–126
 with approximation, 197–242
 prior knowledge, 12, 34, 54, 137, 236, 324, 471

- prioritized sweeping, 170, 168–171
 projected Bellman error, 285
 vector, 267, 269
 proximal TD methods, 286
 pseudo termination, 282, 308
 psychology, 4, 13, 19, 20, 341–376
- $Q(\lambda)$, Watkins’s, 312–314
 Q -function, *see* action-value function
 Q -learning, 21, 131, 131–135
 double, 136
 Q -planning, 161
 $Q(\sigma)$, 156, 154–156
 queuing example, 252
- R-learning, 256
 racetrack exercise, 111
 radial basis functions (RBFs), 221–222
 random walk, 95
 5-state, 125, 126, 127
 19-state, 144, 291
 TD(λ) results on, 294, 295, 299
 1000-state, 203–209, 217, 218
 Fourier and polynomial bases, 214
 real-time dynamic programming, 177–180
 recycling robot example, 52
 REINFORCE, 328, 326–331
 with baseline, 330
 reinforcement learning, 1–22
 reinforcement signal, 380
 representation learning, 473
 residual-gradient algorithm, 272–274, 277
 naive, 270, 271
 return, 54–58
 n-step, 143
 for $Q(\sigma)$, 155
 for action values, 146
 for Expected Sarsa, 148
 for Tree Backup, 153
 with control variates, 150, 151
 with function approximation, 209
 differential, 250, 255, 334
 flat partial, 113
 with state-dependent termination, 308
 λ -return, 288–291
 truncated, 296
 reward prediction error hypothesis, 381–383, 387–395
 reward signal, 1, 6, 48, 53, 361, 380, 383, 397
- and reinforcement, 373–375, 380–381
 design of, 469–472, 477
 intrinsic, 474
 sparse, 469–470
 rod maneuvering example, 171
 rollout algorithms, 183–185
 root mean square (RMS) error, 125
- safety, 434, 478
 sample and expected updates, 121, 170–174
 sample or simulation model, 115
 sample-average method, 27
 Samuel’s checkers player, 20, 241, 426–429
 Sarsa, 130, 129–131, 244
 vs Q-learning, 132
 differential, one-step, 251
 Expected, 133–134, 140
 n-step, 148
 n-step off-policy, 150
 double, 136
 n-step, 147, 145–148, 247
 differential, 255
 off-policy, 149
 Sarsa(λ), 305, 303–307
 true online, 307
 Schultz, Wolfram, 387–395, 410
 search control, 163
 secondary reinforcement, 20, 346, 354, 369
 selective bootstrap adaptation, 239
 semi-gradient methods, 202, 258–259
 SGD, *see* stochastic gradient descent
 Shannon, Claude, 16, 20, 71, 426
 shaping, 360, 470
 Skinner, B. F., 359–360, 375, 470, 479
 soap bubble example, 95
 soft and ε -soft policies, 100–103, 110
 soft-max, 322–323, 329, 336, 400, 445, 455
 for bandits, 37, 45
 spike-timing-dependent plasticity (STDP), 401
 state, 7, 48, 49
 kth-order history approach, 468
 and observations, 464–468
 Markov property, 465–468
 belief, 467
 latent, 467
 observable operator models (OOMs), 467
 partially observable MDPs, 14, 467
 predictive state representations, 467
 state-update function, 465

- state aggregation, 203–204
 state-update function, 465
 step-size parameter, 10, 31–33, 120, 125, 126
 automatic adaptation, 238
 in DQN, 439, 440
 in psychological models, 347, 348
 selecting manually, 222–223
 with coarse coding, 216
 with Fourier features, 213
 with tile coding, 217, 223
 stochastic approx. convergence conditions, 33
 stochastic gradient descent (SGD), 200–204
 in the Bellman error, 269–278
 strong and weak methods, 4
 supervised learning, xvii, 2, 17–19, 198
 sweeps, 75, 160, *see also* prioritized sweeping
 synaptic plasticity, 379
 Hebbian, 400
 two-factor and three factor, 400
 system identification, 364
 tabular solution methods, 23
 target
 policy, 103, 110
 of update, 31, 143, 198
 TD, *see* temporal-difference learning
 TD error, 121
 n-step, 255
 differential, 250
 with function approximation, 270
 TD(λ), 293, 292–295
 truncated, 295–297
 true online, 300, 299–301
 TD-Gammon, 21, 421–426
 temporal abstraction, 461–464
 temporal-difference learning, 10, 119–140
 history of, 20–21
 advantages of, 124–126
 optimality of, 126–128
 TD(0), 120, 203
 TD(1), 294
 TD(λ), 293, 292–295
 true online, 300, 299–301
 λ -return methods
 off-line, 290
 online, 297–299
 n-step, 144, 141–158, 209
 termination function, 307, 459
 Thompson sampling, 43, 45
 Thorndike, Edward, *see* Law of Effect
 tic-tac-toe, 8–13, 17, 137
 tile coding, 217–221, 223, 238, 246, 434, 435
 Tolman, Edward, 364, 408
 trace-decay parameter (λ), 287, 289, 290, 292
 state dependent, 307
 trajectory sampling, 174–177
 transition probabilities, 49
 Tree Backup
 n-step, 152–153, 154
 Tree-Backup(λ), 312–314
 trial-and-error, 1, 7, 15–21, 403, 404, *see also*
 instrumental conditioning
 true online TD(λ), 300, 299–301
 Tsitsiklis and Van Roy's Counterexample, 263
 undiscounted continuing tasks, *see* average re-
 ward setting
 unsupervised learning, 2, 226
 value, 6, 26, 47
 value function, 6, 58–67
 for a given policy: v_π and q_π , 58
 for an optimal policy: v_* and q_* , 62
 action, 58, 63, 65, 71, 129, 131
 approximate action values: $\hat{q}(s, a, \mathbf{w})$, 243
 approximate state values: $\hat{v}(s, \mathbf{w})$, 197
 differential, 243
 vs evolutionary methods, 11
 value iteration, 83, 82–84
 value-function approximation, 198
 Watkins, Chris, 15, 21, 89, 320
 Watson (*Jeopardy!* player), 429–432
 Werbos, Paul, 14, 21, 70, 89, 139, 239
 Witten, Ian, 21, 70

Adaptive Computation and Machine Learning

Francis Bach, Editor

Bioinformatics: The Machine Learning Approach, Pierre Baldi and Søren Brunak

Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto

Graphical Models for Machine Learning and Digital Communication, Brendan J. Frey

Learning in Graphical Models, Michael I. Jordan

Causation, Prediction, and Search, second edition, Peter Spirtes, Clark Glymour, and Richard Scheines

Principles of Data Mining, David Hand, Heikki Mannila, and Padhraic Smyth

Bioinformatics: The Machine Learning Approach, second edition, Pierre Baldi and Søren Brunak

Learning Kernel Classifiers: Theory and Algorithms, Ralf Herbrich

Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond, Bernhard Schölkopf and Alexander J. Smola

Introduction to Machine Learning, Ethem Alpaydin

Gaussian Processes for Machine Learning, Carl Edward Rasmussen and Christopher K.I. Williams

Semi-Supervised Learning, Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien, Eds.

The Minimum Description Length Principle, Peter D. Grünwald

Introduction to Statistical Relational Learning, Lise Getoor and Ben Taskar, Eds.

Probabilistic Graphical Models: Principles and Techniques, Daphne Koller and Nir Friedman

Introduction to Machine Learning, second edition, Ethem Alpaydin

Machine Learning in Non-Stationary Environments: Introduction to Covariate Shift Adaptation, Masashi Sugiyama and Motoaki Kawanabe

Boosting: Foundations and Algorithms, Robert E. Schapire and Yoav Freund

Machine Learning: A Probabilistic Perspective, Kevin P. Murphy

Foundations of Machine Learning, Mehryar Mohri, Afshin Rostami, and Ameet Talwalkar

Introduction to Machine Learning, third edition, Ethem Alpaydin

Deep Learning, Ian Goodfellow, Yoshua Bengio, and Aaron Courville

Elements of Causal Inference, Jonas Peters, Dominik Janzing, and Bernhard Schölkopf

Machine Learning for Data Streams, with Practical Examples in MOA, Albert Bifet, Ricard Gavaldà, Geoffrey Holmes, Bernhard Pfahringer

Incorporating Search Algorithms into RTS Game Agents

David Churchill and Michael Buro

University of Alberta
Computing Science Department
Edmonton, Alberta, Canada, T6G 2E8

Abstract

Real-time strategy (RTS) games are known to be one of the most complex game genres for humans to play, as well as one of the most difficult games for computer AI agents to play well. To tackle the task of applying AI to RTS games, recent techniques have focused on a divide-and-conquer approach, splitting the game into strategic components, and developing separate systems to solve each. This trend gives rise to a new problem: how to tie these systems together into a functional real-time strategy game playing agent. In this paper we discuss the architecture of UAlbertaBot, our entry into the 2011/2012 StarCraft AI competitions, and the techniques used to include heuristic search based AI systems for the intelligent automation of both build order planning and unit control for combat scenarios.

Introduction

Traditional games such as Chess and Go have for centuries been regarded as the most strategically difficult games to play at a top level. High-level play involves complex strategic decisions based on knowledge obtained through study and training, combined with online analysis of the pieces on the current board. Top players are able to “look ahead” a dozen or more moves into the future to decide on an action, often under strict time constraints, with clocks for each player ticking away as they think make their decision. Let us now imagine a genre of game in which the playing field is 256 times as large, contains up to several hundred pieces per player, with pieces able to be created or destroyed at any moment. On top of this, players may move any number of pieces simultaneously in real-time, with the only limit being their own dexterity. What we have just described is a real-time strategy (RTS) game, which combines the complex strategic elements of traditional games with the real-time actions of a modern video game.

A relatively new genre, the first RTS games started to appear in the early 1990s with titles such as Dune II, WarCraft, and Command and Conquer. Originally introduced as a single-player war simulation, their popularity exploded as the internet allowed for players to compete against each other in multiplayer scenarios. With the creation of StarCraft in 1998, RTS games had reached a level of strategy unseen in other video game genres. Tournaments such as the World

Cyber Games had prize pools large enough to create the first professional players by the year 2000, with prize totals in 2011 totalling in the millions of dollars.

In recent years, the field of AI for RTS games has grown as it has shown to be an excellent test-bed for AI algorithms. Due to the enormous size and complexity of the RTS domain, as well as its harsh real-time processing requirements, RTS AI agents are still quite weak with respect to their human counterparts, when compared to the successes of AI agents in traditional games like chess. For example, the winning entry of both the 2011 AIIDE StarCraft AI Competition and the 2011 CIG StarCraft AI Competition, Skynet, was later trivially defeated by a skilled amateur player in a man-machine showmatch.

Due to the complexity of AI algorithms for RTS games (Furtak and Buro 2010), current state of the art RTS AI agents consist primarily of large rule-based systems (like finite state machines) which implement hard-coded strategies derived from expert knowledge. Recently however some agents have started to incorporate dynamic AI solutions in an attempt to improve performance in certain areas of their play. As more and more AI solutions are developed, the task of integrating them smoothly into an existing agent can be a difficult one.

In this paper we address the problem of incorporating AI algorithms into RTS game agents which face uncertainty about the durations of actions as they are executed by the game engine for which no source code is available. Moreover, game engine interfaces may limit access to essential state information, such as the last time opponent units fired, which is crucial for planning actions for combat units. In the following sections we first describe the StarCraft programming interface and how to extract relevant game data from it. Then we discuss our RTS game agent’s hierarchical AI structure and its novel search-based AI modules for build order planning and unit micro-management, and encountered complications when integrating the search procedures into our StarCraft agent. We conclude the paper by experimental results and remarks on future work in this research area.

StarCraft Programming

In the development of any game-playing agent, two requirements are crucial:

- Access to game specific data to perform strategic analysis both on and offline

- Access to play the game itself

Unlike traditional games like chess, real-time strategy games are much more difficult to simulate, requiring systems that govern all aspects of play such as unit movements and attacking. Because StarCraft is a retail game for which no publicly available source code exists, we have two options: either model an approximation of the game in a system we construct, or construct a programming interface to the game to allow us to play the exact version. Luckily, there are tools available for us that do both.

BWAPI

BWAPI (<http://code.google.com/p/bwapi/>) is a programming interface written in C++ which allows users to read data and send game commands to a StarCraft: BroodWar game client. BWAPI contains all functionality necessary for the creation of a competitive AI agent. Examples of BWAPI functionality are:

- Perform unit actions, i.e.: Attack, Move, Build
- Obtain current data about any visible unit, such as Position, HP, Mana, isIdle
- Obtain offline data about any unit type, such as MaxSpeed, Damage, MaxHP, Size, isFlyer

Programs written with BWAPI are compiled into a Windows dynamically linked library (dll) which is injected into the StarCraft game process via a third-party program called “ChaosLauncher”. BWAPI allows the user to perform any of the above functionality while the game is running, after each logic frame update within the game’s software. After each logic frame, BWAPI interrupts the StarCraft process and allows the user to read game data and issue commands, which are stored in a queue to be executed during the game’s next logic frame. No further StarCraft game logic or animation updates are allowed until all sequential user code has finished executing, a fact we will be concerned about when writing resource intensive AI systems.

As no StarCraft source code has been released by its producer (Blizzard), BWAPI has been created via a process of reverse engineering, with all functionality arising from the getting and setting of data in the StarCraft process’ memory space. Although BWAPI provides much functionality, it currently does not give access to some specific data such as unit attack animation frame durations, which are necessary for optimal unit control (discussed in section 5).

PyICE

To extract additional data, we use PyICE: part of the PyMS (<http://www.broodwarai.com/index.php?page=pyms>) modding suite, it is a tool which allows the reading and editing of StarCraft IScript bin files, which are used by the game to control various aspects of StarCraft such as unit behaviour, sprite animations, attack damage timings, and sound triggers. By analyzing the game script files we are able to extract all additional information necessary to (in theory) carry out unit commands.

Architecture

Several architecture models exist for the construction of game playing agents. (Wintermute, Xu, and Laird 2007) use

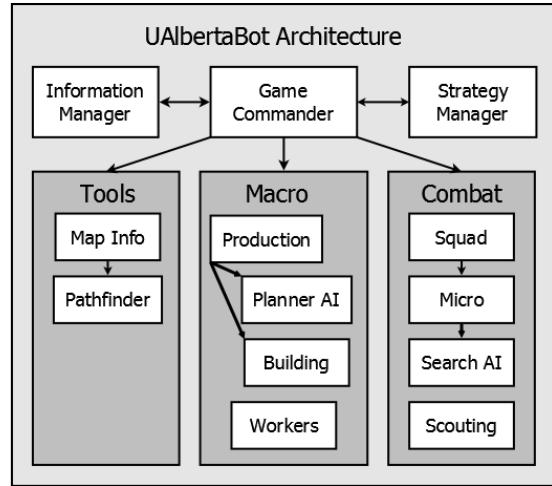


Figure 1: UAlbertaBot Class Diagram

the SOAR cognitive architecture to implement an RTS agent for the ORTS RTS game (ORTS 2010). They implement a finite state machine based approach to respond to perceptual data from the game environment. (Jaidee, Muoz-Avila, and Aha 2011), use a case-based reasoning architecture to play Wargus, an open source clone of Warcraft II. Their method selects goals based on ‘discrepancy’ events which differ from expected behaviours. (McCoy and Mateas 2008) designed a Wargus agent using ABL (A Behavioural Language), a language designed for reactive planning. Their system uses multiple manager modules for controlling sub-tasks of more high-level goals.

Like many RTS game AI authors before, we have designed UAlbertaBot’s architecture in a modular and hierarchical fashion (Figure 1) for two reasons: to retain an intuitive command hierarchy, and to avoid overlap in any computations that need to be performed. Inspired by military command structures, tasks are partitioned among modules by their intuitive strategic meaning (combat, economy, etc.), with vertical communication being performed on a “need to know” basis. High level strategy decisions are made by the game commander by compiling all known information about the current game state. Commands are then given to combat and macro sub-commanders, which in turn give commands to their sub-commanders which are directly in charge of completing the low-level task.

To illustrate how this system works, we will use an example of combat unit micro-management. When the game commander decides to attack a location on the map, it passes this decision onto the combat manager. The combat manager then decides on a squad of attack units to send and moves them toward the location. Once these units enter a given radius of where the battle is to take place, the units are then controlled by the micro manager. The micro manager packages the information about the current combat scenario and then passes that information to a function which will return the actions we should perform for each unit.

The advantage of this architecture is that this function can now be of any level of sophistication, whether it be hand-coded script or a complex search-based AI system. By implementing this design, we can start with a system which

may be full of scripted modules, and gradually replace them by more sophisticated AI systems as we develop them without changing the overall architecture of the agent. For example in 2010, UAlbertaBot used all scripted solutions for each module it contained. For the 2011 version we developed a heuristic search algorithm for build order planning, which simply replaced the existing scripted build order module. We have now done the same thing with unit micro-management by replacing our previously hand-scripted unit actions with our AI based approach.

Resource Allocation

In most cases, the more resources given to an AI agent, the stronger the results that will be returned. This can be problematic for AI systems in RTS games due to the harsh real-time constraints of the game. In the case of BWAPI, the interface will halt the progression of the game until all calculations in the agent's `onFrame()` method have been performed. If a user runs an algorithm for one minute to decide on an action for a given frame, the game will be effectively paused for both players for the full minute. In the 2011 AIIDE StarCraft AI Competition, bots were given 55 ms per game frame to perform all calculations, with game loss penalties for going over this threshold a set number of times. Therefore, it is critical to come up with an efficient scheme for resource allocation when integrating an AI system into an agent that has to act in real-time.

BWAPI allows an agent to gain frame-accurate details about a game state. Therefore, an ideal system would process the game state at every frame and decide on a new set of actions to be carried out for the following frame. This however may not be possible in all cases due to resource budgeting, so one of three situations must arise:

1. Resources (such as processing time) given to an AI system must be reduced to produce an action for the following frame. This will typically result in decreased performance from the AI.
2. The AI system will use more time than is remaining in a single frame, either by staggering the computations across multiple frames, or by performing them in a separate thread. This will yield delayed results from the algorithm because they were given input from a frame at some point in the past.
3. Similar to the previous situation, however one can attempt to estimate the input of a future state to alleviate the side effects of delayed effects. This method then relies on having a good future estimation.

UAlbertaBot uses method 1 and method 2 for its AI systems. In the case of unit micro-management, method 1 is used and the heuristic search algorithm is given the remaining resources for a particular frame to perform its calculations. This is done due to the nature of real-time combat, in which a delayed action may be the difference between life or death. For the build order planning system method 2 is used and calculations are staggered over multiple frames (up to a maximum of 50 frames). The reason for this is that the build order planner may produce a plan which ends up spanning several minutes into the future, so we typically see that the shorter plans produced by allocating more resources

Algorithm 1 Game Commander `onFrame`

```

1: procedure onFrame ()
2:   Clock.start();
3:   Tools.update();
4:   ... Tools.Map.update();
5:   ... Tools.Map.Pathfinder.update();
6:   Macro.update();
7:   ... Macro.WorkerManager.update();
8:   ... Macro.ProductionManager.update();
9:   Combat.update();
10:  ... Combat.ScoutManager.update();
11:  ... Combat.SquadManager.update();
12:  ... Combat.MicroManager.update();
13:  BuildPlannerAI.run(TimeLimit-Clock.elapsed());
14:  MicroAISystem.run(TimeLimit-Clock.elapsed());

```

far outweigh delaying its execution for a few seconds. In the cases where both AI systems are required to function on the same frame, we simply split the available resources equally among both systems, which can be done either in sequence or in parallel by spawning a separate thread whose duration matches the time remaining in the frame. To accurately determine how much time to allocate to the AI systems we delay their execution until after all other bot tasks have been completed for this frame. To illustrate how this is done, pseudo code for game commander's `onFrame()` method is shown in Algorithm 1.

AI-Specific Implementation Details

When integrating an AI system into an RTS agent, especially those designed to work with retail gaming software such as StarCraft, we may encounter different issues depending on the role of the AI we are implementing. In this section we will discuss some of the implementation details and difficulties associated with the two AI systems we have incorporated into UAlbertaBot.

Build Order Planning

In any RTS game there is an initial phase in which players must gather resources in order to construct training facilities, which in turn construct their armies which will engage in combat. An ordered sequence of actions which produces a given set of goal units is called a *build order*. Due to its exponential time complexity, build order planning requires an approximate AI solution for their construction. We have developed a build order planning using an any-time depth first branch and bound algorithm (Churchill and Buro 2011).

When implementing this system into UAlbertaBot, no real difficulties arose. The system is treated as a black-box function which when given an input set of unit goals, returns a sequence of actions to be executed by the agent. This function simply replaced the previous build order system, which was a rule based finite state machine. Due to the nature of the the build order problem, all actions produced by the algorithm are at a macro scale, such as "construct building" and "gather minerals", with no fine-grain motion or knowledge of the StarCraft engine required.

This is not the case however for actions at a micro scale, where delaying execution by even a few frames may cause disastrous effects.

Table 1: Sequence of events occurring after an attack command has been given in StarCraft. Also listed are the associated BWAPI `unit.isAttacking()` and `unit.isAttackFrame()` return values for the given step.

Attack Sequence	isAttacking	isAttackFrame	Additional Notes
1. Unit is Idle	False	False	Unit may be idle or performing another command (i.e.: move)
2. Issue Attack Cmd	False	False	Player gives order to attack a target unit
3. Turn to Face Target	False	False	May have 0 duration if already facing target
4. Approach Target	False	False	May have 0 duration if already in range of target
5. Stop Moving	False	False	Some units require unit to come to complete stop before firing
6. Begin Attack Anim	True	True	Attack animation starts, damage not yet dealt
7. Anim Until Damage	True	True	Animation frames until projectile released
8. Mandatory Anim	True	True	Extra animation frames after damage (may be 0)
9. Optional Anim	True	True	Other command can be issued to cancel extraneous frames
10. Wait for Reload	True	False	Unit may be given other commands until it can shoot again
11. Goto Step 3	False	False	Repeat the attack

Unit Micro-Management

RTS battles are complex adversarial encounters which can be classified as two-player zero-sum simultaneous move games (Churchill and Buro 2011). As in most complex games, no simple scripted strategy performs well in all possible scenarios. We implemented the ABCD (Alpha-Beta search Considering Durations) search algorithm for use in RTS combat settings that is described in (Churchill, Saffidine, and Buro), and integrated it into UAlbertaBot. ABCD search takes action durations into account and approximates game-theoretic solutions of simultaneous move games by sequentializing moves using simple policies such as Max-Min-Max-Min or Min-Max-Min-Max. Applied to RTS game combat scenarios, the paper describes various state evaluation methods ranging from simple static evaluation to simulation-based approaches in which games are played to completion by executing scripted policies. Some details of our implementation are discussed later. For more information about ABCD search we refer the reader to (Churchill, Saffidine, and Buro).

Similar to the build order planning system, our ABCD search based combat AI system acts as a function which takes a given combat scenario as input, and as output produces individual unit actions. Unlike the previous case however, these actions must be carried out with extreme precision in order to guarantee good results. Despite BWAPI’s comprehensive interface into the StarCraft game engine, there are still some intuitively simple tasks which require non-trivial effort to implement. Take for example the case of issuing an attack command to a unit in the game. To carry out frame-perfect unit micro-management we will require knowledge of the exact frame in which the unit has fired its weapon and dealt its damage. This is important because StarCraft’s game engine will cancel an attack command if another command is given before damage has been dealt, resulting in less damage being done by the unit over time. Currently, there is no functionality in BWAPI which can give us this exact information, so it must be extracted via a combination of reverse-engineered game logic and animation script data obtained via PyICE.

BWAPI gives us access to two separate functions to help determine if a unit is currently attacking: `unit.isAttacking()`, which returns true if the unit is currently firing at a unit with intent to continue firing, and `unit.isAttackFrame()`, which returns true if the unit

is current animating with an attack animation frame. Table 1 shows the sequence of events which take place after issuing a `unit.attack()` command in StarCraft. Steps 1-5 deal with the unit moving into a position and heading at which it can fire, steps 6-9 deal with the actual firing time of the unit, and step 10 is a period of time where the unit is waiting until it can fire again. We can see by this sequence that neither function gives us the exact time when the unit dealt its damage, due to steps 8 and 9, which are steps in which these functions return true, but after damage has already been inflicted. We must therefore use the information extracted from PyICE to determine the frame when damage has been dealt (the end of step 7). For a given unit, we extract the duration of steps 6-9 from PyICE and call this value *atkFrames*.

To determine this timing, we will keep track of the unit after we have given an attack command to make sure no other commands are given before the end of step 7. We record the first frame after the attack command was given for which the `unit.isAttackFrame()` returns true (the beginning of step 6), and call this value *startAtk*. We then calculate the frame in the future when the unit will have dealt its damage by:

$$damageFrame = startAtk + atkFrames$$

By issuing subsequent commands to the unit only after *damageFrame* we ensure that no attacks will be interrupted, while allowing the unit to perform other commands between attacks for as long as possible. For example, a Protoss Dragoon unit has an attack cooldown of 23 frames, but an *atkFrames* value of 7, which means it has 16 frames after firing that it is free to move around before it fires again, which can be useful for strategic attack sequences such as *kiting*, a technique used against units with short range weapons to avoid taking damage by fleeing outside of its weapon range while waiting to reload.

However, despite this effort which should work in theory, in practice the StarCraft engine does not behave in a strict deterministic fashion, and work is still being done to perfect this model so that a higher level of precise combat unit control can be obtained.

Experiments

In order to evaluate the success of our architecture, we will perform experiments that demonstrate two points:

- That the AI systems have been successfully integrated into the bot, and that the resource allocation strategy we

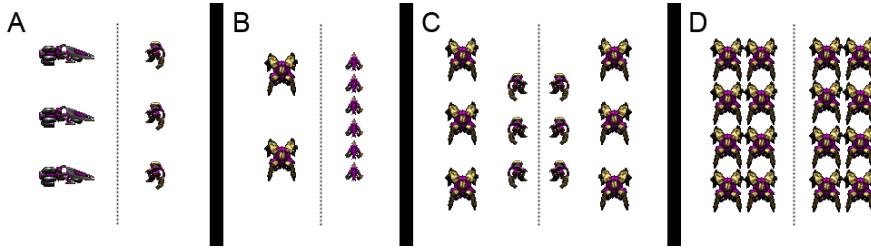


Figure 2: Micro search experiment scenarios. A) 3 ranged Vultures vs. 3 melee Zealot. B) 2 ranged Dragoons vs. 6 fast melee Zerglings. C) 3 Dragoon + 3 Zealots in symmetric formation. D) 8 Dragoons in symmetric two-column formation.

- chose did not cause any issues in bot performance
- That the AI system contributed to the success of the agent as a whole, and is an improvement over existing techniques

Build Order Planner Performance

To evaluate our build order planning AI we use the results from the 2011 AIIDE StarCraft AI Competition. UAlbertaBot incorporated the build order planning system described in section 5.1. All building actions were planned by this system, and re-planning was triggered by the following events: at the start of the game, when the current build queue is exhausted, or when a worker unit or building unit is destroyed.

As shown in Figure 1, the build order planner was controlled by the production manager of the bot. When one of the above events was triggered, the production manager would pass a new unit composition goal into the planner, and it would return an ordered list of actions to be carried out by the building manager.

Resource management was implemented as shown in algorithm 1. The game commander’s main `onFrame()` function keeps track of remaining resources as each of the other systems perform their tasks. Remaining time resources for the current frame were then given to the planner, which ran for the given time limit during the current frame. At the end of this frame time limit, the search state was saved, and the computation resumed during the next frame, until an overall time limit was reached. For our experiment, this overall time limit was set to 5 seconds. The bot ran single-threaded on an Intel E8500 CPU with 4 GB of RAM, with less than 1 MB of memory used for the planner itself.

UAlbertaBot came in 2nd place in both the 2011 AIIDE StarCraft AI Competition, as well as the 2011 CIG StarCraft AI Competition. All build orders (several dozen per game) were planned using the AI system in real-time. Over the course of 30 rounds of round robin competition, UAlbertaBot won 79.4% of its 360 matches, with no game losses due to processing time-outs. While these results cannot conclude whether or not the planner contributed to the bot’s success, they do confirm that the architecture and resource management strategies were sound. In addition, it was the only build order planning AI present in any competition of 2011, with all other entries using rule-based orderings.

Combat Micro AI Performance

To evaluate our combat search AI system, we implemented a simplified version of UAlbertaBot in BWAPI which only

performs combat scenarios. Unfortunately, due to time constraints our micro AI system has not yet been implemented into the full version of UAlbertaBot. For this reason we will only allocate 5 ms per frame to our AI’s search algorithm in order to simulate a harsh environment in which the full bot is executing each frame. In 2011, UAlbertaBot’s micro-management system involved a policy of “Attack Weakest Enemy Unit in Range”, with an option for game commander to retreat the squad from combat for various strategic purposes. For this experiment no retreat was allowed — combat is performed until one team has been eliminated or a time limit of 5000 frames (208 seconds) is reached.

In this experiment we construct several StarCraft test maps which contain pre-positioned combat scenarios. To evaluate our AI system (Search) we will do a comparison of its performance to that of the micro-management system present in the 2011 version of UAlbertaBot (AttackWeakest). Due to the desire to avoid issues with network latency (necessary to play one combat policy against the other directly) we instead chose to perform combat with both methods vs. the default StarCraft AI, and then compare the obtained scores. The default StarCraft AI’s combat policy is not explicitly known, however it is thought to be approximately equal to AttackWeakest, however it does not appear to be deterministic. To test our BWAPI implementation itself we will also run the same experiments inside a simulated model of the StarCraft combat engine as described in (Churchill, Saffidine, and Buro). Games were played against the AttackWeakest scripted policy, which is the closest known to that of the default StarCraft AI.

The scenarios we construct are designed to be realistic early-mid game combat scenarios which could be found in an actual StarCraft game. They have also been designed specifically to showcase a variety of scenarios for which no single scripted combat policy can perform well under all cases, and can be seen in Figure 2. Units for each player are shown separated by a dotted line, with the default AI units placed to the right of this line. Unit positions were fixed to the formations shown at the start of each trial, but units were allowed to freely move about the map if they are instructed to do so. For each method, 200 combat trials were performed in each of the scenarios.

Scenario A is designed such that the quicker, ranged Vulture units start within firing range of the zealots, and must adopt a “run and shoot” strategy (known as kiting) to defeat the slower but stronger melee Zealot units. Scenario B is similar to A. However, two strong ranged Dragoons must also kite a swarm of weaker melee Zerglings to survive. Sce-

Table 2: Results from the micro AI experiment. Shown are scores for Micro Search, AttackWeakest, and Kiter decision policies each versus the built-in StarCraft AI for each scenario. Scores are shown for both the micro simulator (Sim) and the actual BWAPI-based implementation (Game).

Combat Decision Settings						
	Search (5 ms)		AtkWeakest		Kiter	
	Sim	Game	Sim	Game	Sim	Game
A	1.00	0.81	0	0	1.00	0.99
A'	1.00	0.78	0	0	1.00	0.99
B	1.00	0.65	0	0	1.00	0.94
B'	1.00	0.68	0	0	1.00	0.89
C	1.00	0.95	0.50	0.56	0	0.14
C'	1.00	0.94	0.50	0.61	0	0.09
D	1.00	0.96	0.50	0.58	0	0.11
D'	1.00	0.97	0.50	0.55	0	0.08
Avg	1.00	0.84	0.25	0.29	0.50	0.53

nario C is symmetric, with initial positions allowing the Dragoons to reach the opponent zealots, but not the opponent Dragoons. Scenario D is also symmetric, wherein each unit is within firing range of each other unit. Therefore, a good targeting policy will perform well.

In addition to the scenarios shown, four more scenarios A', B', C', and D' were tested, each having a similar formation to the previously listed scenarios. However, their positions will be perturbed slightly to break their perfect line formations. In the case of C and D, symmetry was maintained for fairness. These experiments were performed on hardware similar to the build-order planning hardware, with 1 MB total memory used for the search routine and 2 MB for the transposition table.

The results from the micro search experiment are presented in Table 2. Shown are scores for a given combat method, which are defined as: $score = wins + draws/2$. We can see from these results that it is possible (through expert knowledge) to design a scripted combat policy (such as Kiter) which will perform well in scenarios where it is beneficial to move out of shorter enemy attack range like in scenarios A/B, but will fail in scenarios where excess movement is detrimental as it imposes an small delay on firing like in scenarios C/D. Scripts such as AttackWeakest perform better than Kiter in scenarios in which its better targeting policy and lack of movement allow for more effect damage output, but fail completely in situations such as A/B where standing still in range of powerful melee enemies spells certain death. By implementing a dynamic AI solution for combat micro problems, we have dramatically improved overall performance over a wide range of scenarios, even while under the extremely small time constraint of 5 ms per frame.

Also of note in these results is the fact that although the scripted strategies are deterministic, the outcome in the actual BWAPI implementation was not always the same for each trial. In a true deterministic and controllable RTS game model (such as our simulator), each of the scripted results should either be all wins, losses, or draws. This surprising result must be due to the nature of the StarCraft engine itself, for which we do not have an exact model. It

is known that the StarCraft engine does have a small level of stochastic behaviour both in its unit hit chance mechanism (<http://code.google.com/p/bwapi/wiki/StarcraftGuide>) and its random starting unit heading direction. It is unknown whether or not the default combat policy contains non-deterministic elements. It also highlights an additional frustration of implementing an RTS game bot in a real-world scenario: that results may not always be exactly repeatable, so robust designs are necessary.

Conclusion and Future Work

In this paper we have described the architecture of our RTS game playing agent and how we integrated various AI components in a hierarchical fashion. The current focus of our research is on adding AI search components for build order planning and unit micro management. While integrating build order planning is straight forward because precise action timing isn't crucial, we have encountered problems when incorporating our combat search module into a StarCraft bot that have yet to be overcome. The challenge is to accurately model complex action timings — many of them undocumented but crucial for combat situations — based on limited information provided by BWAPI, the programming interface to the StarCraft game engine. Stated another way, we need to better align the fidelity of our unit micro-management search with an environment in which exact data is not available. This can possibly be achieved by measuring (average) action times in controlled experiments or by using a two-step search procedure in which a high-level search considers sequences of low-level macro actions that are restricted to those that can be efficiently executed without slowing down attacks and unit motion. The promising experimental results we presented make us feel confident that in time for the 2012 AIIDE StarCraft AI competition our unit micro-management will generally outperform the built-in combat scripts and — hopefully — most of the competitors.

References

- Churchill, D., and Buro, M. 2011. Build order optimization in StarCraft. In *Proceedings of AIIDE*.
- Churchill, D.; Saffidine, A.; and Buro, M. Fast heuristic search for RTS game combat scenarios. In *Proceedings of AIIDE*, (pre-print available at www.cs.ualberta.ca/~mburo/ps/aiide12-combat.pdf).
- Furtak, T., and Buro, M. 2010. On the complexity of two-player attrition games played on graphs. In Youngblood, G. M., and Bulitko, V., eds., *Proceedings of AIIDE*.
- Jaidee, U.; Muoz-Avila, H.; and Aha, D. W. 2011. Case-based learning in goal-driven autonomy agents for real-time strategy combat tasks. In *Proceedings of the ICCBR Workshop on Computer Games*, 43–52.
- McCoy, J., and Mateas, M. 2008. An integrated agent for playing real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence*. Chicago, Illinois: AAAI Press, 1313–1318.
- ORTS. 2010. ORTS - A Free Software RTS Game Engine. <http://www.cs.ualberta.ca/~mburo/orts/>.
- Wintermute, S.; Xu, J.; and Laird, J. E. 2007. Sorts: A human-level approach to real-time strategy ai. In *Proceedings of the Third AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE 2007.

Foundations and Trends® in Machine Learning

An Introduction to Deep Reinforcement Learning

Suggested Citation: Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare and Joelle Pineau (2018), "An Introduction to Deep Reinforcement Learning", Foundations and Trends® in Machine Learning: Vol. 11, No. 3-4, pp 219–354. DOI: 10.1561/2200000071.

Vincent François-Lavet
McGill University
vincent.francois-lavet@mcgill.ca

Peter Henderson
McGill University
peter.henderson@mail.mcgill.ca

Riashat Islam
McGill University
riashat.islam@mail.mcgill.ca

Marc G. Bellemare
Google Brain
bellemare@google.com

Joelle Pineau
Facebook, McGill University
jpineau@cs.mcgill.ca

This article may be used only for the purpose of research, teaching, and/or private study. Commercial use or systematic downloading (by robots or other automatic processes) is prohibited without explicit Publisher approval.

now
the essence of knowledge
Boston — Delft

Contents

1	Introduction	220
1.1	Motivation	220
1.2	Outline	221
2	Machine learning and deep learning	224
2.1	Supervised learning and the concepts of bias and overfitting	225
2.2	Unsupervised learning	227
2.3	The deep learning approach	228
3	Introduction to reinforcement learning	233
3.1	Formal framework	234
3.2	Different components to learn a policy	238
3.3	Different settings to learn a policy from data	239
4	Value-based methods for deep RL	242
4.1	Q-learning	242
4.2	Fitted Q-learning	243
4.3	Deep Q-networks	245
4.4	Double DQN	246
4.5	Dueling network architecture	247
4.6	Distributional DQN	249
4.7	Multi-step learning	250

4.8 Combination of all DQN improvements and variants of DQN	252
5 Policy gradient methods for deep RL	254
5.1 Stochastic Policy Gradient	255
5.2 Deterministic Policy Gradient	257
5.3 Actor-Critic Methods	258
5.4 Natural Policy Gradients	260
5.5 Trust Region Optimization	261
5.6 Combining policy gradient and Q-learning	262
6 Model-based methods for deep RL	264
6.1 Pure model-based methods	264
6.2 Integrating model-free and model-based methods	267
7 The concept of generalization	271
7.1 Feature selection	276
7.2 Choice of the learning algorithm and function approximator selection	277
7.3 Modifying the objective function	279
7.4 Hierarchical learning	280
7.5 How to obtain the best bias-overfitting tradeoff	281
8 Particular challenges in the online setting	284
8.1 Exploration/Exploitation dilemma	284
8.2 Managing experience replay	289
9 Benchmarking Deep RL	291
9.1 Benchmark Environments	291
9.2 Best practices to benchmark deep RL	296
9.3 Open-source software for Deep RL	298
10 Deep reinforcement learning beyond MDPs	299
10.1 Partial observability and the distribution of (related) MDPs	299
10.2 Transfer learning	304
10.3 Learning without explicit reward function	307
10.4 Multi-agent systems	309

11 Perspectives on deep reinforcement learning	312
11.1 Successes of deep reinforcement learning	312
11.2 Challenges of applying reinforcement learning to real-world problems	313
11.3 Relations between deep RL and neuroscience	314
12 Conclusion	317
12.1 Future development of deep RL	317
12.2 Applications and societal impact of deep RL	318
Appendices	321
A Appendix	322
A.1 Deep RL frameworks	322
References	324

An Introduction to Deep Reinforcement Learning

Vincent François-Lavet¹, Peter Henderson², Riashat Islam³, Marc G. Bellemare⁴ and Joelle Pineau⁵

¹ McGill University; vincent.francois-lavet@mcgill.ca

² McGill University; peter.henderson@mail.mcgill.ca

³ McGill University; riashat.islam@mail.mcgill.ca

⁴ Google Brain; bellemare@google.com

⁵ Facebook, McGill University; jpineau@cs.mcgill.ca

ABSTRACT

Deep reinforcement learning is the combination of reinforcement learning (RL) and deep learning. This field of research has been able to solve a wide range of complex decision-making tasks that were previously out of reach for a machine. Thus, deep RL opens up many new applications in domains such as healthcare, robotics, smart grids, finance, and many more. This manuscript provides an introduction to deep reinforcement learning models, algorithms and techniques. Particular focus is on the aspects related to generalization and how deep RL can be used for practical applications. We assume the reader is familiar with basic machine learning concepts.

1

Introduction

1.1 Motivation

A core topic in machine learning is that of sequential decision-making. This is the task of deciding, from experience, the sequence of actions to perform in an uncertain environment in order to achieve some goals. Sequential decision-making tasks cover a wide range of possible applications with the potential to impact many domains, such as robotics, healthcare, smart grids, finance, self-driving cars, and many more.

Inspired by behavioral psychology (see e.g., Sutton, 1984), reinforcement learning (RL) proposes a formal framework to this problem. The main idea is that an artificial agent may learn by interacting with its environment, similarly to a biological agent. Using the experience gathered, the artificial agent should be able to optimize some objectives given in the form of cumulative rewards. This approach applies in principle to any type of sequential decision-making problem relying on past experience. The environment may be stochastic, the agent may only observe partial information about the current state, the observations may be high-dimensional (e.g., frames and time series), the agent may freely gather experience in the environment or, on the contrary, the data

may be may be constrained (e.g., not access to an accurate simulator or limited data).

Over the past few years, RL has become increasingly popular due to its success in addressing challenging sequential decision-making problems. Several of these achievements are due to the combination of RL with deep learning techniques (LeCun *et al.*, 2015; Schmidhuber, 2015; Goodfellow *et al.*, 2016). This combination, called deep RL, is most useful in problems with high dimensional state-space. Previous RL approaches had a difficult design issue in the choice of features (Munos and Moore, 2002; Bellemare *et al.*, 2013). However, deep RL has been successful in complicated tasks with lower prior knowledge thanks to its ability to learn different levels of abstractions from data. For instance, a deep RL agent can successfully learn from visual perceptual inputs made up of thousands of pixels (Mnih *et al.*, 2015). This opens up the possibility to mimic some human problem solving capabilities, even in high-dimensional space — which, only a few years ago, was difficult to conceive.

Several notable works using deep RL in games have stood out for attaining super-human level in playing Atari games from the pixels (Mnih *et al.*, 2015), mastering Go (Silver *et al.*, 2016b) or beating the world’s top professionals at the game of Poker (Brown and Sandholm, 2017; Moravčík *et al.*, 2017). Deep RL also has potential for real-world applications such as robotics (Levine *et al.*, 2016; Gandhi *et al.*, 2017; Pinto *et al.*, 2017), self-driving cars (You *et al.*, 2017), finance (Deng *et al.*, 2017) and smart grids (François-Lavet, 2017), to name a few. Nonetheless, several challenges arise in applying deep RL algorithms. Among others, exploring the environment efficiently or being able to generalize a good behavior in a slightly different context are not straightforward. Thus, a large array of algorithms have been proposed for the deep RL framework, depending on a variety of settings of the sequential decision-making tasks.

1.2 Outline

The goal of this introduction to deep RL is to guide the reader towards effective use and understanding of core methods, as well as provide

references for further reading. After reading this introduction, the reader should be able to understand the key different deep RL approaches and algorithms and should be able to apply them. The reader should also have enough background to investigate the scientific literature further and pursue research on deep RL.

In Chapter 2, we introduce the field of machine learning and the deep learning approach. The goal is to provide the general technical context and explain briefly where deep learning is situated in the broader field of machine learning. We assume the reader is familiar with basic notions of supervised and unsupervised learning; however, we briefly review the essentials.

In Chapter 3, we provide the general RL framework along with the case of a Markov Decision Process (MDP). In that context, we examine the different methodologies that can be used to train a deep RL agent. On the one hand, learning a value function (Chapter 4) and/or a direct representation of the policy (Chapter 5) belong to the so-called model-free approaches. On the other hand, planning algorithms that can make use of a learned model of the environment belong to the so-called model-based approaches (Chapter 6).

We dedicate Chapter 7 to the notion of generalization in RL. Within either a model-based or a model-free approach, we discuss the importance of different elements: (i) feature selection, (ii) function approximator selection, (iii) modifying the objective function and (iv) hierarchical learning. In Chapter 8, we present the main challenges of using RL in the online setting. In particular, we discuss the exploration-exploitation dilemma and the use of a replay memory.

In Chapter 9, we provide an overview of different existing benchmarks for evaluation of RL algorithms. Furthermore, we present a set of best practices to ensure consistency and reproducibility of the results obtained on the different benchmarks.

In Chapter 10, we discuss more general settings than MDPs: (i) the Partially Observable Markov Decision Process (POMDP), (ii) the distribution of MDPs (instead of a given MDP) along with the notion of transfer learning, (iii) learning without explicit reward function and (iv) multi-agent systems. We provide descriptions of how deep RL can be used in these settings.

In Chapter 11, we present broader perspectives on deep RL. This includes a discussion on applications of deep RL in various domains, along with the successes achieved and remaining challenges (e.g. robotics, self driving cars, smart grids, healthcare, etc.). This also includes a brief discussion on the relationship between deep RL and neuroscience.

Finally, we provide a conclusion in Chapter 12 with an outlook on the future development of deep RL techniques, their future applications, as well as the societal impact of deep RL and artificial intelligence.

2

Machine learning and deep learning

Machine learning provides automated methods that can detect patterns in data and use them to achieve some tasks (Christopher, 2006; Murphy, 2012). Three types of machine learning tasks can be considered:

- *Supervised learning* is the task of inferring a classification or regression from labeled training data.
- *Unsupervised learning* is the task of drawing inferences from datasets consisting of input data without labeled responses.
- *Reinforcement learning* (RL) is the task of learning how agents ought to take sequences of actions in an environment in order to maximize cumulative rewards.

To solve these machine learning tasks, the idea of function approximators is at the heart of machine learning. There exist many different types of function approximators: linear models (Anderson *et al.*, 1958), SVMs (Cortes and Vapnik, 1995), decisions tree (Liaw, Wiener, *et al.*, 2002; Geurts *et al.*, 2006), Gaussian processes (Rasmussen, 2004), deep learning (LeCun *et al.*, 2015; Schmidhuber, 2015; Goodfellow *et al.*, 2016), etc.

In recent years, mainly due to recent developments in deep learning, machine learning has undergone dramatic improvements when learning from high-dimensional data such as time series, images and videos. These improvements can be linked to the following aspects: (i) an exponential increase of computational power with the use of GPUs and distributed computing (Krizhevsky *et al.*, 2012), (ii) methodological breakthroughs in deep learning (Srivastava *et al.*, 2014; Ioffe and Szegedy, 2015; He *et al.*, 2016; Szegedy *et al.*, 2016; Klambauer *et al.*, 2017), (iii) a growing eco-system of softwares such as Tensorflow (Abadi *et al.*, 2016) and datasets such as ImageNet (Russakovsky *et al.*, 2015). All these aspects are complementary and, in the last few years, they have lead to a virtuous circle for the development of deep learning.

In this chapter, we discuss the supervised learning setting along with the key concepts of bias and overfitting. We briefly discuss the unsupervised setting with tasks such as data compression and generative models. We also introduce the deep learning approach that has become key to the whole field of machine learning. Using the concepts introduced in this chapter, we cover the reinforcement learning setting in later chapters.

2.1 Supervised learning and the concepts of bias and overfitting

In its most abstract form, supervised learning consists in finding a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that takes as input $x \in \mathcal{X}$ and gives as output $y \in \mathcal{Y}$ (\mathcal{X} and \mathcal{Y} depend on the application):

$$y = f(x). \quad (2.1)$$

A supervised learning algorithm can be viewed as a function that maps a dataset D_{LS} of learning samples $(x, y) \stackrel{\text{i.i.d.}}{\sim} (X, Y)$ into a model. The prediction of such a model at a point $x \in \mathcal{X}$ of the input space is denoted by $f(x | D_{LS})$. Assuming a random sampling scheme, s.t. $D_{LS} \sim \mathcal{D}_{LS}$, $f(x | D_{LS})$ is a random variable, and so is its average error over the input space. The expected value of this quantity is given by:

$$I[f] = \mathbb{E}_X \mathbb{E}_{D_{LS}} \mathbb{E}_{Y|X} L(Y, f(X | D_{LS})), \quad (2.2)$$

where $L(\cdot, \cdot)$ is the loss function. If $L(y, \hat{y}) = (y - \hat{y})^2$, the error decomposes naturally into a sum of a bias term and a variance term¹. This bias-variance decomposition can be useful because it highlights a tradeoff between an error due to erroneous assumptions in the model selection/learning algorithm (the bias) and an error due to the fact that only a finite set of data is available to learn that model (the parametric variance). Note that the parametric variance is also called the overfitting error². Even though there is no such direct decomposition for other loss functions (James, 2003), there is always a tradeoff between a sufficiently rich model (to reduce the model bias, which is present even when the amount of data would be unlimited) and a model not too complex (so as to avoid overfitting to the limited amount of data). Figure 2.1 provides an illustration.

Without knowing the joint probability distribution, it is impossible to compute $I[f]$. Instead, we can compute the empirical error on a sample of data. Given n data points (x_i, y_i) , the empirical error is

$$I_S[f] = \frac{1}{n} \sum_{i=1}^n L(y_i, f(x_i)).$$

The *generalization error* is the difference between the error on a sample set (used for training) and the error on the underlying joint probability distribution. It is defined as

$$G = I[f] - I_S[f].$$

¹ The bias-variance decomposition (Geman *et al.*, 1992) is given by:

$$\mathbb{E}_{D_{LS}} \mathbb{E}_{Y|X} (Y - f(X | D_{LS}))^2 = \sigma^2(x) + \text{bias}^2(x), \quad (2.3)$$

where

$$\begin{aligned} \text{bias}^2(x) &\triangleq (\mathbb{E}_{Y|x}(Y) - \mathbb{E}_{D_{LS}} f(x | D_{LS}))^2, \\ \sigma^2(x) &\triangleq \underbrace{\mathbb{E}_{Y|x} (Y - \mathbb{E}_{Y|x}(Y))^2}_{\text{Internal variance}} + \underbrace{\mathbb{E}_{D_{LS}} (f(x | D_{LS}) - \mathbb{E}_{D_{LS}} f(x | D_{LS}))^2}_{\text{Parametric variance}}, \end{aligned} \quad (2.4)$$

²For any given model, the parametric variance goes to zero with an arbitrary large dataset by considering the strong law of convergence.

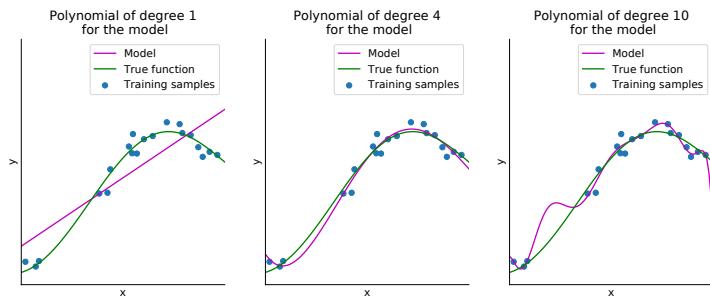


Figure 2.1: Illustration of overfitting and underfitting for a simple 1D regression task in supervised learning (based on one example from the library scikit-learn (Pedregosa *et al.*, 2011)). In this illustration, the data points (x, y) are noisy samples from a true function represented in green. In the left figure, the degree 1 approximation is underfitting, which means that it is not a good model, even for the training samples; on the right, the degree 10 approximation is a very good model for the training samples but is overly complex and fails to provide a good generalization.

In machine learning, the complexity of the function approximator provides upper bounds on the generalization error. The generalization error can be bounded by making use of complexity measures, such as the Rademacher complexity (Bartlett and Mendelson, 2002), or the VC-dimension (Vapnik, 1998). However, even though it lacks strong theoretical foundations, it has become clear in practice that the strength of deep neural networks is their generalization capability, even with a high number of parameters (hence a potentially high complexity) (Zhang *et al.*, 2016).

2.2 Unsupervised learning

Unsupervised learning is a branch of machine learning that learns from data that do not have any label. It relates to using and identifying patterns in the data for tasks such as data compression or generative models.

Data compression or dimensionality reduction involve encoding information using a smaller representation (e.g., fewer bits) than the original representation. For instance, an auto-encoder consists of an encoder and a decoder. The encoder maps the original image $x_i \in \mathbb{R}^M$

onto a low-dimensional representation $z_i = e(x_i; \theta_e) \in \mathbb{R}^m$, where $m << M$; the decoder maps these features back to a high-dimensional representation $d(z_i; \theta_d) \approx e^{-1}(z_i; \theta_e)$. Auto-encoders can be trained by optimizing for the reconstruction of the input through supervised learning objectives.

Generative models aim at approximating the true data distribution of a training set so as to generate new data points from the distribution. Generative adversarial networks (Goodfellow *et al.*, 2014) use an adversarial process, in which two models are trained simultaneously: a generative model G captures the data distribution, while a discriminative model D estimates whether a sample comes from the training data rather than G. The training procedure corresponds to a minimax two-player game.

2.3 The deep learning approach

Deep learning relies on a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ parameterized with $\theta \in \mathbb{R}^{n_\theta}$ ($n_\theta \in \mathbb{N}$):

$$y = f(x; \theta). \quad (2.5)$$

A deep neural network is characterized by a succession of multiple processing layers. Each layer consists in a non-linear transformation and the sequence of these transformations leads to learning different levels of abstraction (Erhan *et al.*, 2009; Olah *et al.*, 2017).

First, let us describe a very simple neural network with one fully-connected hidden layer (see Fig 2.2). The first layer is given the input values (i.e., the input features) x in the form of a column vector of size n_x ($n_x \in \mathbb{N}$). The values of the next hidden layer are a transformation of these values by a non-linear parametric function, which is a matrix multiplication by W_1 of size $n_h \times n_x$ ($n_h \in \mathbb{N}$), plus a bias term b_1 of size n_h , followed by a non-linear transformation:

$$h = A(W_1 \cdot x + b_1), \quad (2.6)$$

where A is the *activation function*. This non-linear activation function is what makes the transformation at each layer non-linear, which ultimately provides the expressivity of the neural network. The hidden layer h of

size n_h can in turn be transformed to other sets of values up to the last transformation that provides the output values y . In this case:

$$y = (W_2 \cdot h + b_2), \quad (2.7)$$

where W_2 is of size $n_y \times n_h$ and b_2 is of size n_y ($n_y \in \mathbb{N}$).

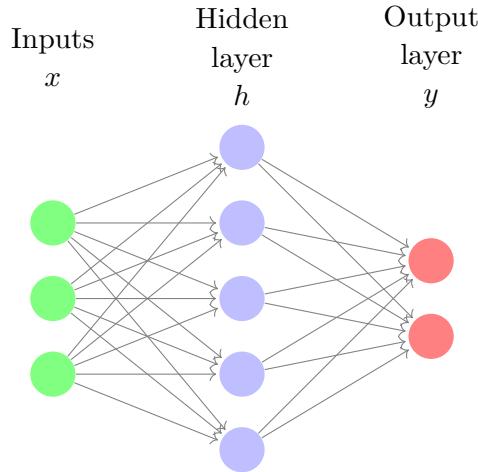


Figure 2.2: Example of a neural network with one hidden layer.

All these layers are trained to minimize the empirical error $I_S[f]$. The most common method for optimizing the parameters of a neural network is based on gradient descent via the backpropagation algorithm (Rumelhart *et al.*, 1988). In the simplest case, at every iteration, the algorithm changes its internal parameters θ so as to fit the desired function:

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} I_S[f], \quad (2.8)$$

where α is the learning rate.

In current applications, many different types of neural network layers have appeared beyond the simple feedforward networks just introduced. Each variation provides specific advantages, depending on the application (e.g., good tradeoff between bias and overfitting in a supervised learning setting). In addition, within one given neural network, an arbitrarily large number of layers is possible, and the trend

in the last few years is to have an ever-growing number of layers, with more than 100 in some supervised learning tasks (Szegedy *et al.*, 2017). We merely describe here two types of layers that are of particular interest in deep RL (and in many other tasks).

Convolutional layers (LeCun, Bengio, *et al.*, 1995) are particularly well suited for images and sequential data (see Fig 2.3), mainly due to their translation invariance property. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field and which apply a convolution operation to the input, passing the result to the next layer. As a result, the network learns filters that activate when it detects some specific features. In image classification, the first layers learn how to detect edges, textures and patterns; then the following layers are able to detect parts of objects and whole objects (Erhan *et al.*, 2009; Olah *et al.*, 2017). In fact, a convolutional layer is a particular kind of feedforward layer, with the specificity that many weights are set to 0 (not learnable) and that other weights are shared.

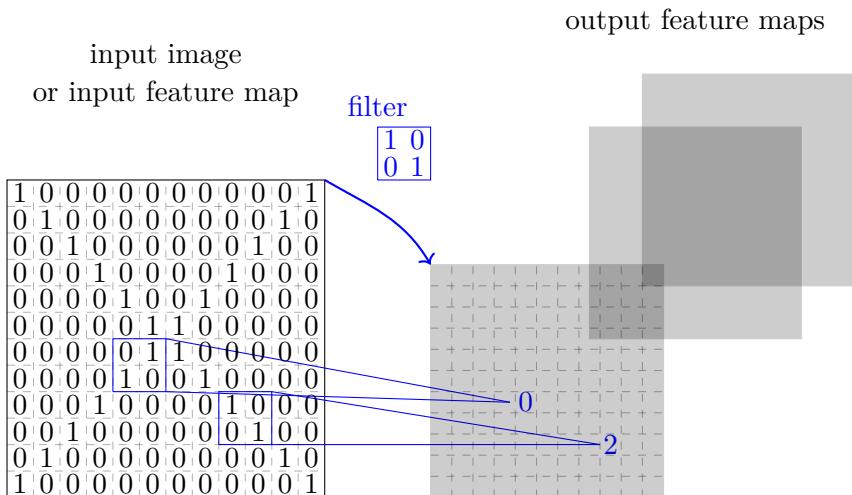


Figure 2.3: Illustration of a convolutional layer with one input feature map that is convolved by different filters to yield the output feature maps. The parameters that are learned for this type of layer are those of the filters. For illustration purposes, some results are displayed for one of the output feature maps with a given filter (in practice, that operation is followed by a non-linear activation function).

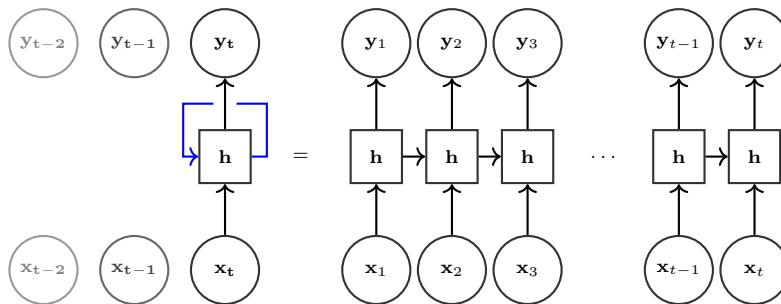


Figure 2.4: Illustration of a simple recurrent neural network. The layer denoted by "h" may represent any non linear function that takes two inputs and provides two outputs. On the left is the simplified view of a recurrent neural network that is applied recursively to (x_t, y_t) for increasing values of t and where the blue line presents a delay of one time step. On the right, the neural network is unfolded with the implicit requirement of presenting all inputs and outputs simultaneously.

Recurrent layers are particularly well suited for sequential data (see Fig 2.4). Several different variants provide particular benefits in different settings. One such example is the long short-term memory network (LSTM) (Hochreiter and Schmidhuber, 1997), which is able to encode information from long sequences, unlike a basic recurrent neural network. Neural Turing Machines (NTMs) (Graves *et al.*, 2014) are another such example. In such systems, a differentiable "external memory" is used for inferring even longer-term dependencies than LSTMs with low degradation.

Several other specific neural network architectures have also been studied to improve generalization in deep learning. For instance, it is possible to design an architecture in such a way that it automatically focuses on only some parts of the inputs with a mechanism called attention (Xu *et al.*, 2015; Vaswani *et al.*, 2017). Other approaches aim to work with symbolic rules by learning to create programs (Reed and De Freitas, 2015; Neelakantan *et al.*, 2015; Johnson *et al.*, 2017; Chen *et al.*, 2017).

To be able to actually apply the deep RL methods described in the later chapters, the reader should have practical knowledge of applying deep learning methods in simple supervised learning settings (e.g., MNIST classification). For information on topics such as the importance

of input normalizations, weight initialization techniques, regularization techniques and the different variants of gradient descent techniques, the reader can refer to several reviews on the subject (LeCun *et al.*, 2015; Schmidhuber, 2015; Goodfellow *et al.*, 2016) as well as references therein.

In the following chapters, the focus is on reinforcement learning, in particular on methods that scale to deep neural network function approximators. These methods allows for learning a wide variety of challenging sequential decision-making tasks directly from rich high-dimensional inputs.

3

Introduction to reinforcement learning

Reinforcement learning (RL) is the area of machine learning that deals with sequential decision-making. In this chapter, we describe how the RL problem can be formalized as an agent that has to make decisions in an environment to optimize a given notion of cumulative rewards. It will become clear that this formalization applies to a wide variety of tasks and captures many essential features of artificial intelligence such as a sense of cause and effect as well as a sense of uncertainty and nondeterminism. This chapter also introduces the different approaches to learning sequential decision-making tasks and how deep RL can be useful.

A key aspect of RL is that an agent *learns* a good behavior. This means that it modifies or acquires new behaviors and skills incrementally. Another important aspect of RL is that it uses trial-and-error *experience* (as opposed to e.g., dynamic programming that assumes full knowledge of the environment *a priori*). Thus, the RL agent does not require complete knowledge or control of the environment; it only needs to be able to interact with the environment and collect information. In an *offline* setting, the experience is acquired *a priori*, then it is used as a batch for learning (hence the offline setting is also called batch RL).

This is in contrast to the *online* setting where data becomes available in a sequential order and is used to progressively update the behavior of the agent. In both cases, the core learning algorithms are essentially the same but the main difference is that in an online setting, the agent can influence how it gathers experience so that it is the most useful for learning. This is an additional challenge mainly because the agent has to deal with the *exploration/exploitation* dilemma while learning (see §8.1 for a detailed discussion). But learning in the online setting can also be an advantage since the agent is able to gather information specifically on the most interesting part of the environment. For that reason, even when the environment is fully known, RL approaches may provide the most computationally efficient approach in practice as compared to some dynamic programming methods that would be inefficient due to this lack of specificity.

3.1 Formal framework

The reinforcement learning setting

The general RL problem is formalized as a discrete time stochastic control process where an agent interacts with its environment in the following way: the agent starts, in a given state within its environment $s_0 \in \mathcal{S}$, by gathering an initial observation $\omega_0 \in \Omega$. At each time step t , the agent has to take an action $a_t \in \mathcal{A}$. As illustrated in Figure 3.1, it follows three consequences: (i) the agent obtains a reward $r_t \in \mathcal{R}$, (ii) the state transitions to $s_{t+1} \in \mathcal{S}$, and (iii) the agent obtains an observation $\omega_{t+1} \in \Omega$. This control setting was first proposed by Bellman, 1957b and later extended to learning by Barto *et al.*, 1983. Comprehensive treatment of RL fundamentals are provided by Sutton and Barto, 2017. Here, we review the main elements of RL before delving into deep RL in the following chapters.

The Markov property

For the sake of simplicity, let us consider first the case of Markovian stochastic control processes (Norris, 1998).

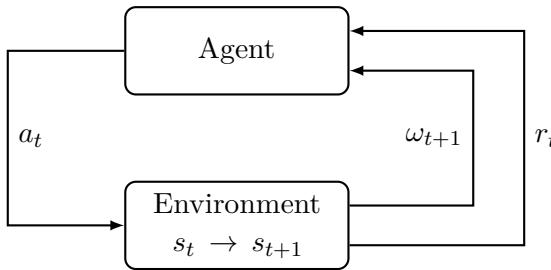


Figure 3.1: Agent-environment interaction in RL.

Definition 3.1. A discrete time stochastic control process is Markovian (i.e., it has the Markov property) if

- $\mathbb{P}(\omega_{t+1} | \omega_t, a_t) = \mathbb{P}(\omega_{t+1} | \omega_t, a_t, \dots, \omega_0, a_0)$, and
- $\mathbb{P}(r_t | \omega_t, a_t) = \mathbb{P}(r_t | \omega_t, a_t, \dots, \omega_0, a_0)$.

The Markov property means that the future of the process only depends on the current observation, and the agent has no interest in looking at the full history.

A Markov Decision Process (MDP) (Bellman, 1957a) is a discrete time stochastic control process defined as follows:

Definition 3.2. An MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ where:

- \mathcal{S} is the state space,
- \mathcal{A} is the action space,
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition function (set of conditional transition probabilities between states),
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R}$ is the reward function, where \mathcal{R} is a continuous set of possible rewards in a range $R_{\max} \in \mathbb{R}^+$ (e.g., $[0, R_{\max}]$),
- $\gamma \in [0, 1]$ is the discount factor.

The system is fully observable in an MDP, which means that the observation is the same as the state of the environment: $\omega_t = s_t$. At each time step t , the probability of moving to s_{t+1} is given by the state

transition function $T(s_t, a_t, s_{t+1})$ and the reward is given by a bounded reward function $R(s_t, a_t, s_{t+1}) \in \mathcal{R}$. This is illustrated in Figure 3.2. Note that more general cases than MDPs are introduced in Chapter 10.

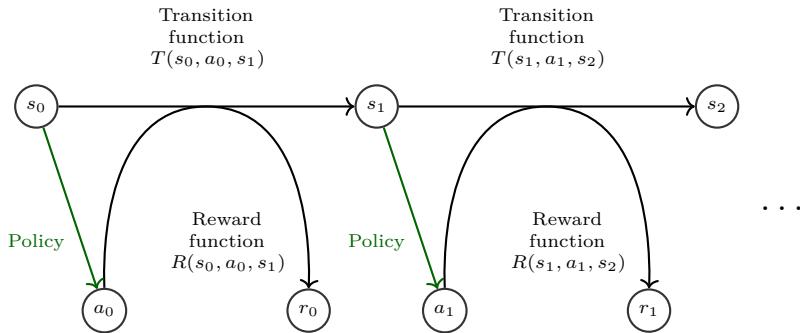


Figure 3.2: Illustration of a MDP. At each step, the agent takes an action that changes its state in the environment and provides a reward.

Different categories of policies

A policy defines how an agent selects actions. Policies can be categorized under the criterion of being either stationary or non-stationary. A non-stationary policy depends on the time-step and is useful for the finite-horizon context where the cumulative rewards that the agent seeks to optimize are limited to a finite number of future time steps (Bertsekas *et al.*, 1995). In this introduction to deep RL, infinite horizons are considered and the policies are stationary¹.

Policies can also be categorized under a second criterion of being either deterministic or stochastic:

- In the deterministic case, the policy is described by $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$.
- In the stochastic case, the policy is described by $\pi(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ where $\pi(s, a)$ denotes the probability that action a may be chosen in state s .

¹The formalism can be directly extended to the finite horizon context. In that case, the policy and the cumulative expected returns should be time-dependent.

The expected return

Throughout this survey, we consider the case of an RL agent whose goal is to find a policy $\pi(s, a) \in \Pi$, so as to optimize an expected return $V^\pi(s) : \mathcal{S} \rightarrow \mathbb{R}$ (also called V-value function) such that

$$V^\pi(s) = \mathbb{E} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, \pi \right], \quad (3.1)$$

where:

- $r_t = \mathbb{E}_{a \sim \pi(s_t, \cdot)} R(s_t, a, s_{t+1})$,
- $\mathbb{P}(s_{t+1} | s_t, a_t) = T(s_t, a_t, s_{t+1})$ with $a_t \sim \pi(s_t, \cdot)$,

From the definition of the expected return, the optimal expected return can be defined as:

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s). \quad (3.2)$$

In addition to the V-value function, a few other functions of interest can be introduced. The Q-value function $Q^\pi(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is defined as follows:

$$Q^\pi(s, a) = \mathbb{E} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a, \pi \right]. \quad (3.3)$$

This equation can be rewritten recursively in the case of an MDP using Bellman's equation:

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} T(s, a, s') (R(s, a, s') + \gamma Q^\pi(s', a = \pi(s'))). \quad (3.4)$$

Similarly to the V-value function, the optimal Q-value function $Q^*(s, a)$ can also be defined as

$$Q^*(s, a) = \max_{\pi \in \Pi} Q^\pi(s, a). \quad (3.5)$$

The particularity of the Q-value function as compared to the V-value function is that the optimal policy can be obtained directly from $Q^*(s, a)$:

$$\pi^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q^*(s, a). \quad (3.6)$$

The optimal V-value function $V^*(s)$ is the expected discounted reward when in a given state s while following the policy π^* thereafter. The

optimal Q-value $Q^*(s, a)$ is the expected discounted return when in a given state s and for a given action a while following the policy π^* thereafter.

It is also possible to define the advantage function

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s). \quad (3.7)$$

This quantity describes how good the action a is, as compared to the expected return when following directly policy π .

Note that one straightforward way to obtain estimates of either $V^\pi(s)$, $Q^\pi(s, a)$ or $A^\pi(s, a)$ is to use Monte Carlo methods, i.e. defining an estimate by performing several simulations from s while following policy π . In practice, we will see that this may not be possible in the case of limited data. In addition, even when it is possible, we will see that other methods should usually be preferred for computational efficiency.

3.2 Different components to learn a policy

An RL agent includes one or more of the following components:

- a representation of a *value function* that provides a prediction of how good each state or each state/action pair is,
- a direct representation of the *policy* $\pi(s)$ or $\pi(s, a)$, or
- a *model* of the environment (the estimated transition function and the estimated reward function) in conjunction with a planning algorithm.

The first two components are related to what is called *model-free* RL and are discussed in Chapters 4, 5. When the latter component is used, the algorithm is referred to as *model-based* RL, which is discussed in Chapter 6. A combination of both and why using the combination can be useful is discussed in §6.2. A schema with all possible approaches is provided in Figure 3.3.

For most problems approaching real-world complexity, the state space is high-dimensional (and possibly continuous). In order to learn an estimate of the model, the value function or the policy, there are two main advantages for RL algorithms to rely on deep learning:

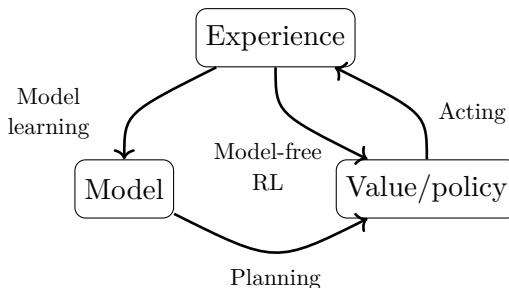


Figure 3.3: General schema of the different methods for RL. The direct approach uses a representation of either a value function or a policy to act in the environment. The indirect approach makes use of a model of the environment.

- Neural networks are well suited for dealing with high-dimensional sensory inputs (such as times series, frames, etc.) and, in practice, they do not require an exponential increase of data when adding extra dimensions to the state or action space (see Chapter 2).
- In addition, they can be trained incrementally and make use of additional samples obtained as learning happens.

3.3 Different settings to learn a policy from data

We now describe key settings that can be tackled with RL.

3.3.1 Offline and online learning

Learning a sequential decision-making task appears in two cases: (i) in the offline learning case where only limited data on a given environment is available and (ii) in an online learning case where, in parallel to learning, the agent gradually gathers experience in the environment. In both cases, the core learning algorithms introduced in Chapters 4 to 6 are essentially the same. The specificity of the batch setting is that the agent has to learn from limited data without the possibility of interacting further with the environment. In that case, the idea of generalization introduced in Chapter 7 is the main focus. In the online setting, the learning problem is more intricate and learning without requiring a large amount of data (sample efficiency) is not only influenced by the

capability of the learning algorithm to generalize well from the limited experience. Indeed, the agent has the possibility to gather experience via an *exploration/exploitation strategy*. In addition, it can use a *replay memory* to store its experience so that it can be reprocessed at a later time. Both the exploration and the replay memory will be discussed in Chapter 8). In both the batch and the online settings, a supplementary consideration is also the computational efficiency, which, among other things, depends on the efficiency of a given gradient descent step. All these elements will be introduced with more details in the following chapters. A general schema of the different elements that can be found in most deep RL algorithms is provided in Figure 3.4.

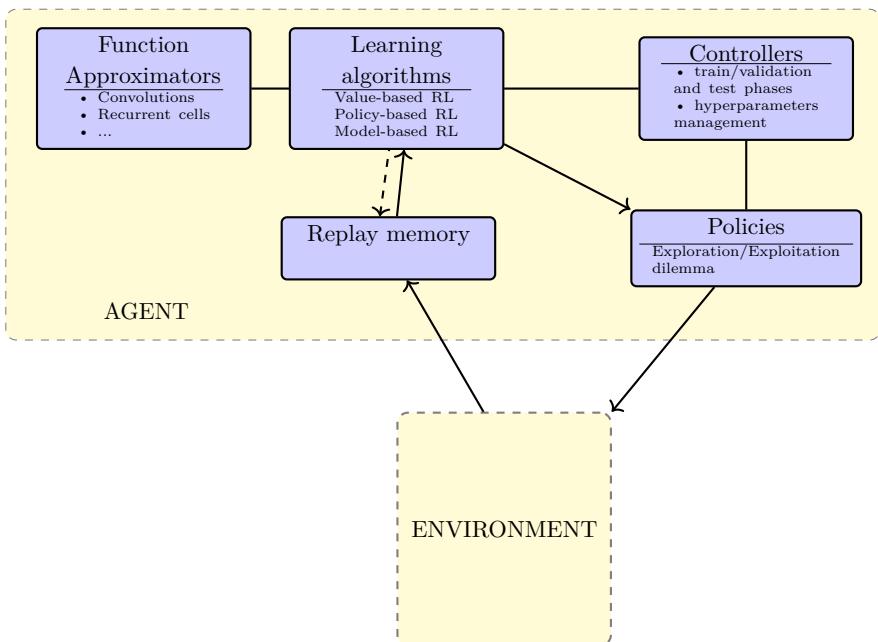


Figure 3.4: General schema of deep RL methods.

3.3.2 Off-policy and on-policy learning

According to Sutton and Barto, 2017, « on-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas

off-policy methods evaluate or improve a policy different from that used to generate the data ». In off-policy based methods, learning is straightforward when using trajectories that are not necessarily obtained under the current policy, but from a different behavior policy $\beta(s, a)$. In those cases, experience replay allows reusing samples from a different behavior policy. On the contrary, on-policy based methods usually introduce a bias when used with a replay buffer as the trajectories are usually not obtained solely under the current policy π . As will be discussed in the following chapters, this makes off-policy methods sample efficient as they are able to make use of any experience; in contrast, on-policy methods would, if specific care is not taken, introduce a bias when using off-policy trajectories.

4

Value-based methods for deep RL

The value-based class of algorithms aims to build a value function, which subsequently lets us define a policy. We discuss hereafter one of the simplest and most popular value-based algorithms, the Q-learning algorithm (Watkins, 1989) and its variant, the fitted Q-learning, that uses parameterized function approximators (Gordon, 1996). We also specifically discuss the main elements of the deep Q-network (DQN) algorithm (Mnih *et al.*, 2015) which has achieved superhuman-level control when playing ATARI games from the pixels by using neural networks as function approximators. We then review various improvements of the DQN algorithm and provide resources for further details. At the end of this chapter and in the next chapter, we discuss the intimate link between value-based methods and policy-based methods.

4.1 Q-learning

The basic version of Q-learning keeps a lookup table of values $Q(s, a)$ (Equation 3.3) with one entry for every state-action pair. In order to learn the optimal Q-value function, the Q-learning algorithm makes use

of the Bellman equation for the Q-value function (Bellman and Dreyfus, 1962) whose unique solution is $Q^*(s, a)$:

$$Q^*(s, a) = (\mathcal{B}Q^*)(s, a), \quad (4.1)$$

where \mathcal{B} is the Bellman operator mapping any function $K : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ into another function $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ and is defined as follows:

$$(\mathcal{B}K)(s, a) = \sum_{s' \in S} T(s, a, s') \left(R(s, a, s') + \gamma \max_{a' \in \mathcal{A}} K(s', a') \right). \quad (4.2)$$

By Banach's theorem, the fixed point of the Bellman operator \mathcal{B} exists since it is a contraction mapping¹. In practice, one general proof of convergence to the optimal value function is available (Watkins and Dayan, 1992) under the conditions that:

- the state-action pairs are represented discretely, and
- all actions are repeatedly sampled in all states (which ensures sufficient exploration, hence not requiring access to the transition model).

This simple setting is often inapplicable due to the high-dimensional (possibly continuous) state-action space. In that context, a parameterized value function $Q(s, a; \theta)$ is needed, where θ refers to some parameters that define the Q-values.

4.2 Fitted Q-learning

Experiences are gathered in a given dataset D in the form of tuples $< s, a, r, s' >$ where the state at the next time-step s' is drawn from $T(s, a, \cdot)$ and the reward r is given by $R(s, a, s')$. In fitted Q-learning (Gordon, 1996), the algorithm starts with some random initialization of the Q-values $Q(s, a; \theta_0)$ where θ_0 refers to the initial parameters (usually

¹The Bellman operator is a contraction mapping because it can be shown that for any pair of bounded functions $K, K' : S \times A \rightarrow \mathbb{R}$, the following condition is respected:

$$\|TK - TK'\|_\infty \leq \gamma \|K - K'\|_\infty.$$

such that the initial Q-values should be relatively close to 0 so as to avoid slow learning). Then, an approximation of the Q-values at the k^{th} iteration $Q(s, a; \theta_k)$ is updated towards the target value

$$Y_k^Q = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a'; \theta_k), \quad (4.3)$$

where θ_k refers to some parameters that define the Q-values at the k^{th} iteration.

In neural fitted Q-learning (NFQ) (Riedmiller, 2005), the state can be provided as an input to the Q-network and a different output is given for each of the possible actions. This provides an efficient structure that has the advantage of obtaining the computation of $\max_{a' \in \mathcal{A}} Q(s', a'; \theta_k)$ in a single forward pass in the neural network for a given s' . The Q-values are parameterized with a neural network $Q(s, a; \theta_k)$ where the parameters θ_k are updated by stochastic gradient descent (or a variant) by minimizing the square loss:

$$\mathcal{L}_{DQN} = \left(Q(s, a; \theta_k) - Y_k^Q \right)^2. \quad (4.4)$$

Thus, the Q-learning update amounts in updating the parameters:

$$\theta_{k+1} = \theta_k + \alpha \left(Y_k^Q - Q(s, a; \theta_k) \right) \nabla_{\theta_k} Q(s, a; \theta_k), \quad (4.5)$$

where α is a scalar step size called the learning rate. Note that using the square loss is not arbitrary. Indeed, it ensures that $Q(s, a; \theta_k)$ should tend without bias to the expected value of the random variable Y_k^Q ². Hence, it ensures that $Q(s, a; \theta_k)$ should tend to $Q^*(s, a)$ after many iterations in the hypothesis that the neural network is well-suited for the task and that the experience gathered in the dataset D is sufficient (more details will be given in Chapter 7).

When updating the weights, one also changes the target. Due to the generalization and extrapolation abilities of neural networks, this approach can build large errors at different places in the state-action space³. Therefore, the contraction mapping property of the Bellman

²The minimum of $\mathbb{E}[(Z - c)^2]$ occurs when the constant c equals the expected value of the random variable Z .

³Note that even fitted value iteration with linear regression can diverge (Boyan and Moore, 1995). However, this drawback does not happen when using linear

operator in Equation 4.2 is not enough to guarantee convergence. It is verified experimentally that these errors may propagate with this update rule and, as a consequence, convergence may be slow or even unstable (Baird, 1995; Tsitsiklis and Van Roy, 1997; Gordon, 1999; Riedmiller, 2005). Another related damaging side-effect of using function approximators is the fact that Q-values tend to be overestimated due to the max operator (Van Hasselt *et al.*, 2016). Because of the instabilities and the risk of overestimation, specific care has been taken to ensure proper learning.

4.3 Deep Q-networks

Leveraging ideas from NFQ, the deep Q-network (DQN) algorithm introduced by Mnih *et al.* (2015) is able to obtain strong performance in an online setting for a variety of ATARI games, directly by learning from the pixels. It uses two heuristics to limit the instabilities:

- The target Q-network in Equation 4.3 is replaced by $Q(s', a'; \theta_k^-)$ where its parameters θ_k^- are updated only every $C \in \mathbb{N}$ iterations with the following assignment: $\theta_k^- = \theta_k$. This prevents the instabilities to propagate quickly and it reduces the risk of divergence as the target values Y_k^Q are kept fixed for C iterations. The idea of target networks can be seen as an instantiation of fitted Q-learning, where each period between target network updates corresponds to a single fitted Q-iteration.
- In an online setting, the replay memory (Lin, 1992) keeps all information for the last $N_{\text{replay}} \in \mathbb{N}$ time steps, where the experience is collected by following an ϵ -greedy policy⁴. The updates are then made on a set of tuples $< s, a, r, s' >$ (called mini-batch) selected randomly within the replay memory. This

function approximators that only have interpolation abilities such as kernel-based regressors (k-nearest neighbors, linear and multilinear interpolation, etc.) (Gordon, 1999) or tree-based ensemble methods (Ernst *et al.*, 2005). However, these methods have not proved able to handle successfully high-dimensional inputs.

⁴It takes a random action with probability ϵ and follows the policy given by $\arg\max_{a \in \mathcal{A}} Q(s, a; \theta_k)$ with probability $1 - \epsilon$.

technique allows for updates that cover a wide range of the state-action space. In addition, one mini-batch update has less variance compared to a single tuple update. Consequently, it provides the possibility to make a larger update of the parameters, while having an efficient parallelization of the algorithm.

A sketch of the algorithm is given in Figure 4.1.

In addition to the target Q-network and the replay memory, DQN uses other important heuristics. To keep the target values in a reasonable scale and to ensure proper learning in practice, rewards are clipped between -1 and +1. Clipping the rewards limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games (however, it introduces a bias). In games where the player has multiple lives, one trick is also to associate a terminal state to the loss of a life such that the agent avoids these terminal states (in a terminal state the discount factor is set to 0).

In DQN, many deep learning specific techniques are also used. In particular, a preprocessing step of the inputs is used to reduce the input dimensionality, to normalize inputs (it scales pixels value into [-1,1]) and to deal with some specificities of the task. In addition, convolutional layers are used for the first layers of the neural network function approximator and the optimization is performed using a variant of stochastic gradient descent called RMSprop (Tieleman, 2012).

4.4 Double DQN

The max operation in Q-learning (Equations 4.2, 4.3) uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values in case of inaccuracies or noise, resulting in overoptimistic value estimates. Therefore, the DQN algorithm induces an upward bias. The double estimator method uses two estimates for each variable, which allows for the selection of an estimator and its value to be uncoupled (Hasselt, 2010). Thus, regardless of whether errors in the estimated Q-values are due to stochasticity in the environment, function approximation, non-stationarity, or any other source, this allows for the removal of the positive bias in estimating the action

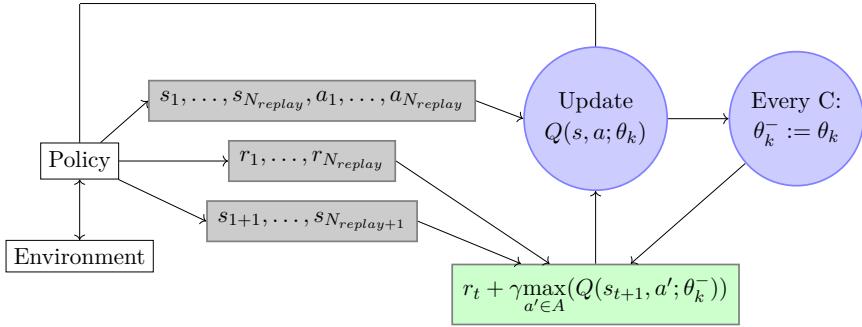


Figure 4.1: Sketch of the DQN algorithm. $Q(s, a; \theta_k)$ is initialized to random values (close to 0) everywhere in its domain and the replay memory is initially empty; the target Q-network parameters θ_k^- are only updated every C iterations with the Q-network parameters θ_k and are held fixed between updates; the update uses a mini-batch (e.g., 32 elements) of tuples $< s, a >$ taken randomly in the replay memory along with the corresponding mini-batch of target values for the tuples.

values. In Double DQN, or DDQN (Van Hasselt *et al.*, 2016), the target value Y_k^Q is replaced by

$$Y_k^{DDQN} = r + \gamma \max_{a \in \mathcal{A}} Q(s', a; \theta_k; \theta_k^-), \quad (4.6)$$

which leads to less overestimation of the Q-learning values, as well as improved stability, hence improved performance. As compared to DQN, the target network with weights θ_t^- are used for the evaluation of the current greedy action. Note that the policy is still chosen according to the values obtained by the current weights θ .

4.5 Dueling network architecture

In (Wang *et al.*, 2015), the neural network architecture decouples the value and advantage function $A^\pi(s, a)$ (Equation 3.7), which leads to improved performance. The Q-value function is given by

$$\begin{aligned} Q(s, a; \theta^{(1)}, \theta^{(2)}, \theta^{(3)}) &= V(s; \theta^{(1)}, \theta^{(3)}) \\ &+ \left(A(s, a; \theta^{(1)}, \theta^{(2)}) - \max_{a' \in \mathcal{A}} A(s, a'; \theta^{(1)}, \theta^{(2)}) \right). \end{aligned} \quad (4.7)$$

Now, for $a^* = \arg\max_{a' \in \mathcal{A}} Q(s, a'; \theta^{(1)}, \theta^{(2)}, \theta^{(3)})$, we obtain $Q(s, a^*; \theta^{(1)}, \theta^{(2)}, \theta^{(3)}) = V(s; \theta^{(1)}, \theta^{(3)})$. As illustrated in Figure 4.2,

the stream $V(s; \theta^{(1)}, \theta^{(3)})$ provides an estimate of the value function, while the other stream produces an estimate of the advantage function. The learning update is done as in DQN and it is only the structure of the neural network that is modified.

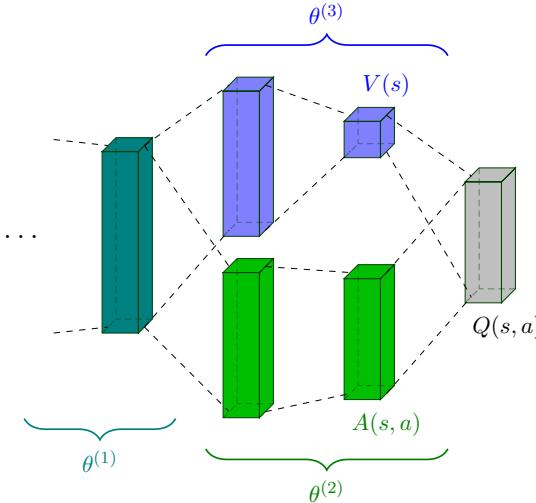


Figure 4.2: Illustration of the dueling network architecture with the two streams that separately estimate the value $V(s)$ and the advantages $A(s, a)$. The boxes represent layers of a neural network and the grey output implements equation 4.7 to combine $V(s)$ and $A(s, a)$.

In fact, even though it loses the original semantics of V and A , a slightly different approach is preferred in practice because it increases the stability of the optimization:

$$\begin{aligned} Q(s, a; \theta^{(1)}, \theta^{(2)}, \theta^{(3)}) &= V(s; \theta^{(1)}, \theta^{(3)}) \\ &+ \left(A(s, a; \theta^{(1)}, \theta^{(2)}) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a'; \theta^{(1)}, \theta^{(2)}) \right). \end{aligned} \quad (4.8)$$

In that case, the advantages only need to change as fast as the mean, which appears to work better in practice (Wang *et al.*, 2015).

4.6 Distributional DQN

The approaches described so far in this chapter all directly approximate the expected return in a value function. Another approach is to aim for a richer representation through a value distribution, i.e. the distribution of possible cumulative returns (Jaquette *et al.*, 1973; Morimura *et al.*, 2010). This value distribution provides more complete information of the intrinsic randomness of the rewards and transitions of the agent within its environment (note that it is not a measure of the agent's uncertainty about the environment).

The value distribution Z^π is a mapping from state-action pairs to distributions of returns when following policy π . It has an expectation equal to Q^π :

$$Q^\pi(s, a) = \mathbb{E}Z^\pi(s, a).$$

This random return is also described by a recursive equation, but one of a distributional nature:

$$Z^\pi(s, a) = R(s, a, S') + \gamma Z^\pi(S', A'), \quad (4.9)$$

where we use capital letters to emphasize the random nature of the next state-action pair (S', A') and $A' \sim \pi(\cdot | S')$. The distributional Bellman equation states that the distribution of Z is characterized by the interaction of three random variables: the reward $R(s, a, S')$, the next state-action (S', A') , and its random return $Z^\pi(S', A')$.

It has been shown that such a distributional Bellman equation can be used in practice, with deep learning as the function approximator (Bellemare *et al.*, 2017; Dabney *et al.*, 2017; Rowland *et al.*, 2018). This approach has the following advantages:

- It is possible to implement risk-aware behavior (see e.g., Morimura *et al.*, 2010).
- It leads to more performant learning in practice. This may appear surprising since both DQN and the distributional DQN aim to maximize the expected return (as illustrated in Figure 4.3). One of the main elements is that the distributional perspective naturally provides a richer set of training signals than a scalar value function

$Q(s, a)$. These training signals that are not a priori necessary for optimizing the expected return are known as *auxiliary tasks* (Jaderberg *et al.*, 2016) and lead to an improved learning (this is discussed in §7.2.1).

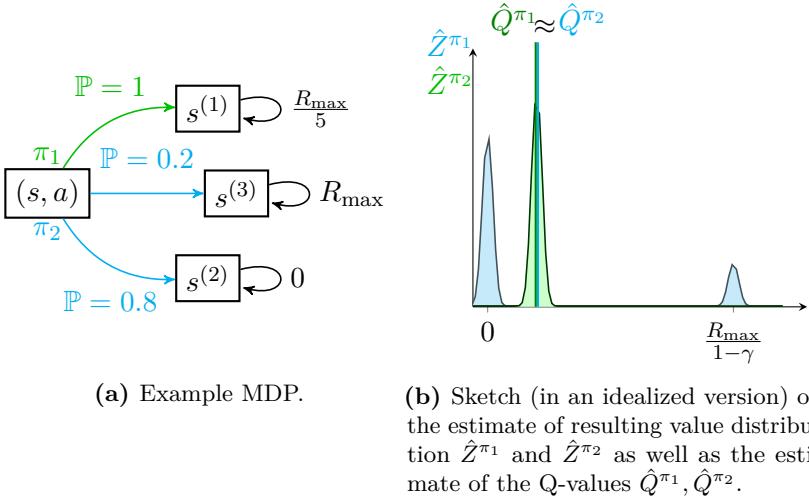


Figure 4.3: For two policies illustrated on Fig (a), the illustration on Fig (b) gives the value distribution $Z^{(\pi)}(s, a)$ as compared to the expected value $Q^\pi(s, a)$. On the left figure, one can see that π_1 moves with certainty to an absorbing state with reward at every step $\frac{R_{\max}}{5}$, while π_2 moves with probability 0.2 and 0.8 to absorbing states with respectively rewards at every step R_{\max} and 0. From the pair (s, a) , the policies π_1 and π_2 have the same expected return but different value distributions.

4.7 Multi-step learning

In DQN, the target value used to update the Q-network parameters (given in Equation 4.3) is estimated as the sum of the immediate reward and a contribution of the following steps in the return. That contribution is estimated based on its own value estimate at the next time-step. For that reason, the learning algorithm is said to *bootstrap* as it recursively uses its own value estimates (Sutton, 1988).

This method of estimating a target value is not the only possibility. Non-bootstrapping methods learn directly from returns (Monte Carlo)

and an intermediate solution is to use a multi-step target (Sutton, 1988; Watkins, 1989; Peng and Williams, 1994; Singh and Sutton, 1996). Such a variant in the case of DQN can be obtained by using the n-step target value given by:

$$Y_k^{Q,n} = \sum_{t=0}^{n-1} \gamma^t r_t + \gamma^n \max_{a' \in A} Q(s_n, a'; \theta_k) \quad (4.10)$$

where $(s_0, a_0, r_0, \dots, s_{n-1}, a_{n-1}, r_{n-1}, s_n)$ is any trajectory of $n+1$ time steps with $s = s_0$ and $a = a_0$. A combination of different multi-steps targets can also be used:

$$Y_k^{Q,n} = \sum_{i=0}^{n-1} \lambda_i \left(\sum_{t=0}^i \gamma^t r_t + \gamma^{i+1} \max_{a' \in A} Q(s_{i+1}, a'; \theta_k) \right) \quad (4.11)$$

with $\sum_{i=0}^{n-1} \lambda_i = 1$. In the method called $TD(\lambda)$ (Sutton, 1988), $n \rightarrow \infty$ and λ_i follow a geometric law: $\lambda_i \propto \lambda^i$ where $0 \leq \lambda \leq 1$.

To bootstrap or not to bootstrap? Bootstrapping has both advantages and disadvantages. On the negative side, using pure bootstrapping methods (such as in DQN) are prone to instabilities when combined with function approximation because they make recursive use of their own value estimate at the next time-step. On the contrary, methods such as n-step Q-learning rely less on their own value estimate because the estimate used is decayed by γ^n for the n^{th} step backup. In addition, methods that rely less on bootstrapping can propagate information more quickly from delayed rewards as they learn directly from returns (Sutton, 1996). Hence they might be more computationally efficient.

Bootstrapping also has advantages. The main advantage is that using value bootstrap allows learning from off-policy samples. Indeed, methods that do not use pure bootstrapping, such as n-step Q-learning with $n > 1$ or $TD(\lambda)$, are in principle on-policy based methods that would introduce a bias when used with trajectories that are not obtained solely under the behavior policy μ (e.g., stored in a replay buffer).

The conditions required to learn efficiently and safely with eligibility traces from off-policy experience are provided by Munos *et al.*, 2016; Harutyunyan *et al.*, 2016. In the control setting, the retrace operator (Munos *et al.*, 2016) considers a sequence of target policies π that depend

on the sequence of Q-functions (such as ϵ -greedy policies), and seek to approximate Q^* (if π is greedy or becomes increasingly greedy w.r.t. the Q estimates). It leads to the following target:

$$Y = Q(s, a) + \left[\sum_{t \geq 0} \gamma^t \left(\prod_{c_s=1}^t c_s \right) (r_t + \gamma \mathbb{E}_\pi Q(s_{t+1}, a') - Q(s_t, a_t)) \right] \quad (4.12)$$

where $c_s = \lambda \min \left(1, \frac{\pi(s, a)}{\mu(s, a)} \right)$ with $0 \leq \lambda \leq 1$ and μ is the behavior policy (estimated from observed samples). This way of updating the Q-network has guaranteed convergence, does not suffer from a high variance and it does not cut the traces unnecessarily when π and μ are close. Nonetheless, one can note that estimating the target is more expansive to compute as compared to the one-step target (such as in DQN) because the Q-value function has to be estimated on more states.

4.8 Combination of all DQN improvements and variants of DQN

The original DQN algorithm can combine the different variants discussed in §4.4 to §4.7 (as well as some discussed in Chapter 8.1) and that has been studied by Hessel *et al.*, 2017. Their experiments show that the combination of all the previously mentioned extensions to DQN provides state-of-the-art performance on the Atari 2600 benchmarks, both in terms of sample efficiency and final performance. Overall, a large majority of Atari games can be solved such that the deep RL agents surpass the human level performance.

Some limitations remain with DQN-based approaches. Among others, these types of algorithms are not well-suited to deal with large and/or continuous action spaces. In addition, they cannot explicitly learn stochastic policies. Modifications that address these limitations will be discussed in the following Chapter 5, where we discuss policy-based approaches. Actually, the next section will also show that value-based and policy-based approaches can be seen as two facets of the same model-free approach. Therefore, the limitations of discrete action spaces and deterministic policies are only related to DQN.

One can also note that value-based or policy-based approaches do not make use of any model of the environment, which limits their sample

efficiency. Ways to combine model-free and model-based approaches will be discussed in Chapter 6.

5

Policy gradient methods for deep RL

This section focuses on a particular family of reinforcement learning algorithms that use policy gradient methods. These methods optimize a performance objective (typically the expected cumulative reward) by finding a good policy (e.g a neural network parameterized policy) thanks to variants of stochastic gradient ascent with respect to the policy parameters. Note that policy gradient methods belong to a broader class of policy-based methods that includes, among others, evolution strategies. These methods use a learning signal derived from sampling instantiations of policy parameters and the set of policies is developed towards policies that achieve better returns (e.g., Salimans *et al.*, 2017).

In this chapter, we introduce the stochastic and deterministic gradient theorems that provide gradients on the policy parameters in order to optimize the performance objective. Then, we present different RL algorithms that make use of these theorems.

5.1 Stochastic Policy Gradient

The expected return of a stochastic policy π starting from a given state s_0 from Equation 3.1 can be written as (Sutton *et al.*, 2000):

$$V^\pi(s_0) = \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \pi(s, a) R'(s, a) da ds, \quad (5.1)$$

where $R'(s, a) = \int_{s' \in \mathcal{S}} T(s, a, s') R(s, a, s')$ and $\rho^\pi(s)$ is the discounted state distribution defined as

$$\rho^\pi(s) = \sum_{t=0}^{\infty} \gamma^t Pr\{s_t = s | s_0, \pi\}.$$

For a differentiable policy π_w , the fundamental result underlying these algorithms is the policy gradient theorem (Sutton *et al.*, 2000):

$$\nabla_w V^{\pi_w}(s_0) = \int_{\mathcal{S}} \rho^{\pi_w}(s) \int_{\mathcal{A}} \nabla_w \pi_w(s, a) Q^{\pi_w}(s, a) da ds. \quad (5.2)$$

This result allows us to adapt the policy parameters w : $\Delta w \propto \nabla_w V^{\pi_w}(s_0)$ from experience. This result is particularly interesting since the policy gradient does not depend on the gradient of the state distribution (even though one might have expected it to). The simplest way to derive the policy gradient estimator (i.e., estimating $\nabla_w V^{\pi_w}(s_0)$ from experience) is to use a *score function gradient estimator*, commonly known as the REINFORCE algorithm (Williams, 1992). The likelihood ratio trick can be exploited as follows to derive a general method of estimating gradients from expectations:

$$\begin{aligned} \nabla_w \pi_w(s, a) &= \pi_w(s, a) \frac{\nabla_w \pi_w(s, a)}{\pi_w(s, a)} \\ &= \pi_w(s, a) \nabla_w \log(\pi_w(s, a)). \end{aligned} \quad (5.3)$$

Considering Equation 5.3, it follows that

$$\nabla_w V^{\pi_w}(s_0) = \mathbb{E}_{s \sim \rho^{\pi_w}, a \sim \pi_w} [\nabla_w (\log \pi_w(s, a)) Q^{\pi_w}(s, a)]. \quad (5.4)$$

Note that, in practice, most policy gradient methods effectively use undiscounted state distributions, without hurting their performance (Thomas, 2014).

So far, we have shown that policy gradient methods should include a policy evaluation followed by a policy improvement. On the one hand, the policy evaluation estimates Q^{π_w} . On the other hand, the policy improvement takes a gradient step to optimize the policy $\pi_w(s, a)$ with respect to the value function estimation. Intuitively, the policy improvement step increases the probability of the actions proportionally to their expected return.

The question that remains is how the agent can perform the policy evaluation step, i.e., how to obtain an estimate of $Q^{\pi_w}(s, a)$. The simplest approach to estimating gradients is to replace the Q function estimator with a cumulative return from entire trajectories. In the Monte-Carlo policy gradient, we estimate the $Q^{\pi_w}(s, a)$ from rollouts on the environment while following policy π_w . The Monte-Carlo estimator is an unbiased well-behaved estimate when used in conjunction with the back-propagation of a neural network policy, as it estimates returns until the end of the trajectories (without instabilities induced by bootstrapping). However, the main drawback is that the estimate requires on-policy rollouts and can exhibit high variance. Several rollouts are typically needed to obtain a good estimate of the return. A more efficient approach is to instead use an estimate of the return given by a value-based approach, as in actor-critic methods discussed in §5.3.

We make two additional remarks. First, to prevent the policy from becoming deterministic, it is common to add an entropy regularizer to the gradient. With this regularizer, the learnt policy can remain stochastic. This ensures that the policy keeps exploring.

Second, instead of using the value function Q^{π_w} in Eq. 5.4, an advantage value function A^{π_w} can also be used. While $Q^{\pi_w}(s, a)$ summarizes the performance of each action for a given state under policy π_w , the advantage function $A^{\pi_w}(s, a)$ provides a measure of comparison for each action to the expected return at the state s , given by $V^{\pi_w}(s)$. Using $A^{\pi_w}(s, a) = Q^{\pi_w}(s, a) - V^{\pi_w}(s)$ has usually lower magnitudes than $Q^{\pi_w}(s, a)$. This helps reduce the variance of the gradient estimator $\nabla_w V^{\pi_w}(s_0)$ in the policy improvement step, while not modifying the

expectation¹. In other words, the value function $V^{\pi_w}(s)$ can be seen as a *baseline* or *control variate* for the gradient estimator. When updating the neural network that fits the policy, using such a baseline allows for improved numerical efficiency – i.e. reaching a given performance with fewer updates – because the learning rate can be bigger.

5.2 Deterministic Policy Gradient

The policy gradient methods may be extended to deterministic policies. The Neural Fitted Q Iteration with Continuous Actions (NFQCA) (Hafner and Riedmiller, 2011) and the Deep Deterministic Policy Gradient (DDPG) (Silver *et al.*, 2014; Lillicrap *et al.*, 2015) algorithms introduce the direct representation of a policy in such a way that it can extend the NFQ and DQN algorithms to overcome the restriction of discrete actions.

Let us denote by $\pi(s)$ the deterministic policy: $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$. In discrete action spaces, a direct approach is to build the policy iteratively with:

$$\pi_{k+1}(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q^{\pi_k}(s, a), \quad (5.5)$$

where π_k is the policy at the k^{th} iteration. In continuous action spaces, a greedy policy improvement becomes problematic, requiring a global maximisation at every step. Instead, let us denote by $\pi_w(s)$ a differentiable deterministic policy. In that case, a simple and computationally attractive alternative is to move the policy in the direction of the gradient of Q , which leads to the Deep Deterministic Policy Gradient (DDPG) algorithm (Lillicrap *et al.*, 2015):

$$\nabla_w V^{\pi_w}(s_0) = \mathbb{E}_{s \sim \rho^{\pi_w}} \left[\nabla_w (\pi_w) \nabla_a (Q^{\pi_w}(s, a)) |_{a=\pi_w(s)} \right]. \quad (5.6)$$

This equation implies relying on $\nabla_a (Q^{\pi_w}(s, a))$ (in addition to $\nabla_w \pi_w$), which usually requires using actor-critic methods (see §5.3).

¹Indeed, subtracting a baseline that only depends on s to $Q^{\pi_w}(s, a)$ in Eq. 5.2 does not change the gradient estimator because $\forall s, \int_{\mathcal{A}} \nabla_w \pi_w(s, a) da = 0$.

5.3 Actor-Critic Methods

As we have seen in §5.1 and §5.2, a policy represented by a neural network can be updated by gradient ascent for both the deterministic and the stochastic case. In both cases, the policy gradient typically requires an estimate of a value function for the current policy. One common approach is to use an actor-critic architecture that consists of two parts: an actor and a critic (Konda and Tsitsiklis, 2000). The actor refers to the policy and the critic to the estimate of a value function (e.g., the Q-value function). In deep RL, both the actor and the critic can be represented by non-linear neural network function approximators (Mnih *et al.*, 2016). The actor uses gradients derived from the policy gradient theorem and adjusts the policy parameters w . The critic, parameterized by θ , estimates the approximate value function for the current policy π : $Q(s, a; \theta) \approx Q^\pi(s, a)$.

The critic

From a (set of) tuples $< s, a, r, s' >$, possibly taken from a replay memory, the simplest off-policy approach to estimating the critic is to use a pure bootstrapping algorithm $TD(0)$ where, at every iteration, the current value $Q(s, a; \theta)$ is updated towards a target value:

$$Y_k^Q = r + \gamma Q(s', a = \pi(s'); \theta) \quad (5.7)$$

This approach has the advantage of being simple, yet it is not computationally efficient as it uses a pure bootstrapping technique that is prone to instabilities and has a slow reward propagation backwards in time (Sutton, 1996). This is similar to the elements discussed in the value-based methods in §4.7.

The ideal is to have an architecture that is

- sample-efficient such that it should be able to make use of both off-policy and on-policy trajectories (i.e., it should be able to use a replay memory), and
- computationally efficient: it should be able to profit from the stability and the fast reward propagation of on-policy methods for samples collected from near on-policy behavior policies.

There are many methods that combine on- and off-policy data for policy evaluation (Precup, 2000). The algorithm *Retrace*(λ) (Munos *et al.*, 2016) has the advantages that (i) it can make use of samples collected from any behavior policy without introducing a bias and (ii) it is efficient as it makes the best use of samples collected from near on-policy behavior policies. That approach was used in actor-critic architectures described by Wang *et al.* (2016b) and Gruslys *et al.* (2017). These architectures are sample-efficient thanks to the use of a replay memory, and computationally efficient since they use multi-step returns which improves the stability of learning and increases the speed of reward propagation backwards in time.

The actor

From Equation 5.4, the off-policy gradient in the policy improvement phase for the stochastic case is given as:

$$\nabla_w V^{\pi_w}(s_0) = \mathbb{E}_{s \sim \rho^{\pi_\beta}, a \sim \pi_\beta} [\nabla_\theta (\log \pi_w(s, a)) Q^{\pi_w}(s, a)]. \quad (5.8)$$

where β is a behavior policy generally different than π , which makes the gradient generally biased. This approach usually behaves properly in practice but the use of a biased policy gradient estimator makes difficult the analysis of its convergence without the GLIE assumption (Munos *et al.*, 2016; Gruslys *et al.*, 2017)².

In the case of actor-critic methods, an approach to perform the policy gradient on-policy without experience replay has been investigated with the use of asynchronous methods, where multiple agents are executed in parallel and the actor-learners are trained asynchronously (Mnih *et al.*, 2016). The parallelization of agents also ensures that each agent experiences different parts of the environment at a given time step. In that case, n-step returns can be used without introducing a bias. This simple idea can be applied to any learning algorithm that requires

²Greedy in the Limit with Infinite Exploration (GLIE) means that the behavior policies are required to become greedy (no exploration) in the limit of an online learning setting where the agent has gathered an infinite amount of experience. It is required that « (i) each action is executed infinitely often in every state that is visited infinitely often, and (ii) in the limit, the learning policy is greedy with respect to the Q-value function with probability 1 »(Singh *et al.*, 2000).

on-policy data and it removes the need to maintain a replay buffer. However, this asynchronous trick is not sample efficient.

An alternative is to combine off-policy and on-policy samples to trade-off both the sample efficiency of off-policy methods and the stability of on-policy gradient estimates. For instance, Q-Prop (Gu *et al.*, 2017b) uses a Monte Carlo on-policy gradient estimator, while reducing the variance of the gradient estimator by using an off-policy critic as a control variate. One limitation of Q-Prop is that it requires using on-policy samples for estimating the policy gradient.

5.4 Natural Policy Gradients

Natural policy gradients are inspired by the idea of natural gradients for the updates of the policy. Natural gradients can be traced back to the work of Amari, 1998 and has been later adapted to reinforcement learning (Kakade, 2001).

Natural policy gradient methods use the steepest direction given by the Fisher information metric, which uses the manifold of the objective function. In the simplest form of steepest ascent for an objective function $J(w)$, the update is of the form $\Delta w \propto \nabla_w J(w)$. In other words, the update follows the direction that maximizes $(J(w) - J(w + \Delta w))$ under a constraint on $\|\Delta w\|_2$. In the hypothesis that the constraint on Δw is defined with another metric than L_2 , the first-order solution to the constrained optimization problem typically has the form $\Delta w \propto B^{-1} \nabla_w J(w)$ where B is an $n_w \times n_w$ matrix. In natural gradients, the norm uses the Fisher information metric, given by a local quadratic approximation to the KL divergence $D_{KL}(\pi^w || \pi^{w+\Delta w})$. The natural gradient ascent for improving the policy π_w is given by

$$\Delta w \propto F_w^{-1} \nabla_w V^{\pi^w}(\cdot), \quad (5.9)$$

where F_w is the Fisher information matrix given by

$$F_w = \mathbb{E}_{\pi_w} [\nabla_w \log \pi_w(s, \cdot) (\nabla_w \log \pi_w(s, \cdot))^T]. \quad (5.10)$$

Policy gradients following $\nabla_w V^{\pi^w}(\cdot)$ are often slow because they are prone to getting stuck in local plateaus. Natural gradients, however, do not follow the usual steepest direction in the parameter space, but the

steepest direction with respect to the Fisher metric. Note that, as the angle between natural and ordinary gradient is never larger than ninety degrees, convergence is also guaranteed when using natural gradients.

The caveat with natural gradients is that, in the case of neural networks and their large number of parameters, it is usually impractical to compute, invert, and store the Fisher information matrix (Schulman *et al.*, 2015). This is the reason why natural policy gradients are usually not used in practice for deep RL; however alternatives inspired by this idea have been found and they are discussed in the following section.

5.5 Trust Region Optimization

As a modification to the natural gradient method, policy optimization methods based on a *trust region* aim at improving the policy while changing it in a controlled way. These constraint-based policy optimization methods focus on restricting the changes in a policy using the KL divergence between the action distributions. By bounding the size of the policy update, trust region methods also bound the changes in state distributions guaranteeing improvements in policy.

TRPO (Schulman *et al.*, 2015) uses constrained updates and advantage function estimation to perform the update, resulting in the reformulated optimization given by

$$\max_{\Delta w} \mathbb{E}_{s \sim \rho^{\pi_w}, a \sim \pi_w} \left[\frac{\pi_{w+\Delta w}(s, a)}{\pi_w(s, a)} A^{\pi_w}(s, a) \right] \quad (5.11)$$

subject to $\mathbb{E} D_{\text{KL}}(\pi_w(s, \cdot) || \pi_{w+\Delta w}(s, \cdot)) \leq \delta$, where $\delta \in \mathbb{R}$ is a hyperparameter. From empirical data, TRPO uses a conjugate gradient with KL constraint to optimize the objective function.

Proximal Policy Optimization (PPO) (Schulman *et al.*, 2017b) is a variant of the TRPO algorithm, which formulates the constraint as a penalty or a clipping objective, instead of using the KL constraint. Unlike TRPO, PPO considers modifying the objective function to penalize changes to the policy that move $r_t(w) = \frac{\pi_{w+\Delta w}(s, a)}{\pi_w(s, a)}$ away from 1. The clipping objective that PPO maximizes is given by

$$\mathbb{E}_{s \sim \rho^{\pi_w}, a \sim \pi_w} \left[\min \left(r_t(w) A^{\pi_w}(s, a), \text{clip}(r_t(w), 1 - \epsilon, 1 + \epsilon) A^{\pi_w}(s, a) \right) \right] \quad (5.12)$$

where $\epsilon \in \mathbb{R}$ is a hyperparameter. This objective function clips the probability ratio to constrain the changes of r_t in the interval $[1 - \epsilon, 1 + \epsilon]$.

5.6 Combining policy gradient and Q-learning

Policy gradient is an efficient technique for improving a policy in a reinforcement learning setting. As we have seen, this typically requires an estimate of a value function for the current policy and a sample efficient approach is to use an actor-critic architecture that can work with off-policy data.

These algorithms have the following properties unlike the methods based on DQN discussed in Chapter 4:

- They are able to work with continuous action spaces. This is particularly interesting in applications such as robotics, where forces and torques can take a continuum of values.
- They can represent stochastic policies, which is useful for building policies that can explicitly explore. This is also useful in settings where the optimal policy is a stochastic policy (e.g., in a multi-agent setting where the Nash equilibrium is a stochastic policy).

However, another approach is to combine policy gradient methods directly with off-policy Q-learning (O'Donoghue *et al.*, 2016). In some specific settings, depending on the loss function and the entropy regularization used, value-based methods and policy-based methods are equivalent (Fox *et al.*, 2015; O'Donoghue *et al.*, 2016; Haarnoja *et al.*, 2017; Schulman *et al.*, 2017a). For instance, when adding an entropy regularization, Eq. 5.4 can be written as

$$\nabla_w V^{\pi_w}(s_0) = \mathbb{E}_{s,a} [\nabla_w (\log \pi_w(s, a)) Q^{\pi_w}(s, a)] + \alpha \mathbb{E}_s \nabla_w H^{\pi_w}(s). \quad (5.13)$$

where $H^{\pi}(s) = -\sum_a \pi(s, a) \log \pi(s, a)$. From this, one can note that an optimum is satisfied by the following policy: $\pi_w(s, a) = \exp(A^{\pi_w}(s, a)/\alpha - H^{\pi_w}(s))$. Therefore, we can use the policy to derive an estimate of the advantage function: $\tilde{A}^{\pi_w}(s, a) = \alpha(\log \pi_w(s, a) + H^{\pi}(s))$.

We can thus think of all model-free methods as different facets of the same approach.

One remaining limitation is that both value-based and policy-based methods are model-free and they do not make use of any model of the environment. The next chapter describes algorithms with a model-based approach.

6

Model-based methods for deep RL

In Chapters 4 and 5, we have discussed the model-free approach that rely either on a value-based or a policy-based method. In this chapter, we introduce the model-based approach that relies on a model of the environment (dynamics and reward function) in conjunction with a planning algorithm. In §6.2, the respective strengths of the model-based versus the model-free approaches are discussed, along with how the two approaches can be integrated.

6.1 Pure model-based methods

A model of the environment is either explicitly given (e.g., in the game of Go for which all the rules are known a priori) or learned from experience. To learn the model, yet again function approximators bring significant advantages in high-dimensional (possibly partially observable) environments (Oh *et al.*, 2015; Mathieu *et al.*, 2015; Finn *et al.*, 2016a; Kalchbrenner *et al.*, 2016; Duchesne *et al.*, 2017; Nagabandi *et al.*, 2018). The model can then act as a proxy for the actual environment.

When a model of the environment is available, planning consists in interacting with the model to recommend an action. In the case of discrete actions, lookahead search is usually done by generating

potential trajectories. In the case of a continuous action space, trajectory optimization with a variety of controllers can be used.

6.1.1 Lookahead search

A lookahead search in an MDP iteratively builds a decision tree where the current state is the root node. It stores the obtained returns in the nodes and focuses attention on promising potential trajectories. The main difficulty in sampling trajectories is to balance *exploration* and *exploitation*. On the one hand, the purpose of exploration is to gather more information on the part of the search tree where few simulations have been performed (i.e., where the expected value has a high variance). On the other hand, the purpose of exploitation is to refine the expected value of the most promising moves.

Monte-Carlo tree search (MCTS) techniques (Browne *et al.*, 2012) are popular approaches to lookahead search. Among others, they have gained popularity thanks to prolific achievements in the challenging task of computer Go (Brügmann, 1993; Gelly *et al.*, 2006; Silver *et al.*, 2016b). The idea is to sample multiple trajectories from the current state until a terminal condition is reached (e.g., a given maximum depth) (see Figure 6.1 for an illustration). From those simulation steps, the MCTS algorithm then recommends an action to take.

Recent works have developed strategies to directly learn end-to-end the model, along with how to make the best use of it, without relying on explicit tree search techniques (Pascanu *et al.*, 2017). These approaches show improved sample efficiency, performance, and robustness to model misspecification compared to the separated approach (simply learning the model and then relying on it during planning).

6.1.2 Trajectory optimization

Lookahead search techniques are limited to discrete actions, and alternative techniques have to be used for the case of continuous actions. If the model is differentiable, one can directly compute an analytic policy gradient by backpropagation of rewards along trajectories (Nguyen and Widrow, 1990). For instance, PILCO (Deisenroth and Rasmussen, 2011) uses Gaussian processes to learn a probabilistic model of the

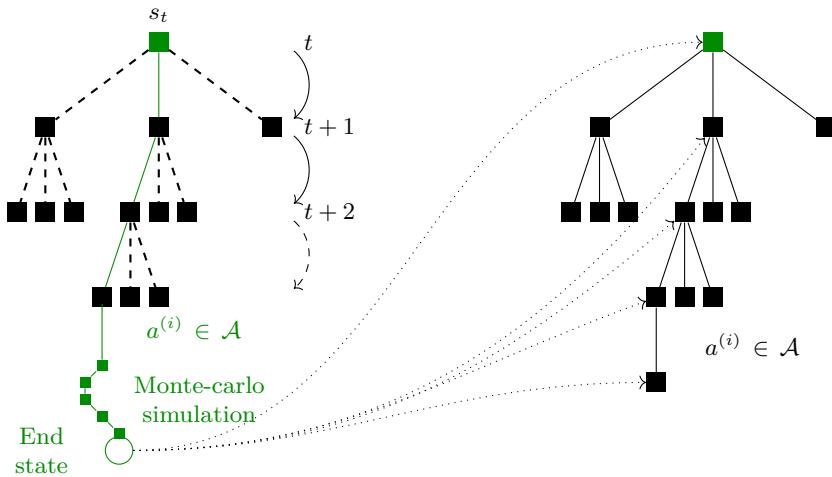


Figure 6.1: Illustration of how a MCTS algorithm performs a Monte-Carlo simulation and builds a tree by updating the statistics of the different nodes. Based on the statistics gathered for the current node s_t , the MCTS algorithm chooses an action to perform on the actual environment.

dynamics. It can then explicitly use the uncertainty for planning and policy evaluation in order to achieve a good sample efficiency. However, the gaussian processes have not been able to scale reliably to high-dimensional problems.

One approach to scale planning to higher dimensions is to aim at leveraging the generalization capabilities of deep learning. For instance, Wahlström *et al.* (2015) uses a deep learning model of the dynamics (with an auto-encoder) along with a model in a latent state space. Model-predictive control (Morari and Lee, 1999) can then be used to find the policy by repeatedly solving a finite-horizon optimal control problem in the latent space. It is also possible to build a probabilistic generative model in a latent space with the objective that it possesses a locally linear dynamics, which allows control to be performed more efficiently (Watter *et al.*, 2015). Another approach is to use the trajectory optimizer as a teacher rather than a demonstrator: guided policy search (Levine and Koltun, 2013) takes a few sequences of actions suggested by another controller. It then learns to adjust the policy from these sequences. Methods that leverage trajectory optimization have

demonstrated many capabilities, for instance in the case of simulated 3D bipeds and quadrupeds (e.g., Mordatch *et al.*, 2015).

6.2 Integrating model-free and model-based methods

The respective strengths of the model-free versus model-based approaches depend on different factors. First, the best suited approach depends on whether the agent has access to a model of the environment. If that's not the case, the learned model usually has some inaccuracies that should be taken into account. Note that learning the model can share the hidden-state representations with a value-based approach by sharing neural network parameters (Li *et al.*, 2015).

Second, a model-based approach requires working in conjunction with a planning algorithm (or controller), which is often computationally demanding. The time constraints for computing the policy $\pi(s)$ via planning must therefore be taken into account (e.g., for applications with real-time decision-making or simply due to resource limitations).

Third, for some tasks, the structure of the policy (or value function) is the easiest one to learn, but for other tasks, the model of the environment may be learned more efficiently due to the particular structure of the task (less complex or with more regularity). Thus, the most performant approach depends on the structure of the model, policy, and value function (see the coming Chapter 7 for more details on generalization). Let us consider two examples to better understand this key consideration. In a labyrinth where the agent has full observability, it is clear how actions affect the next state and the dynamics of the model may easily be generalized by the agent from only a few tuples (for instance, the agent is blocked when trying to cross a wall of the labyrinth). Once the model is known, a planning algorithm can then be used with high performance. Let us now discuss another example where, on the contrary, planning is more difficult: an agent has to cross a road with random events happening everywhere on the road. Let us suppose that the best policy is simply to move forward except when an object has just appeared in front of the agent. In that case, the optimal policy may easily be captured by a model-free approach, while a model-based approach would be more difficult (mainly due to the stochasticity of the model which

leads to many different possible situations, even for one given sequence of actions).

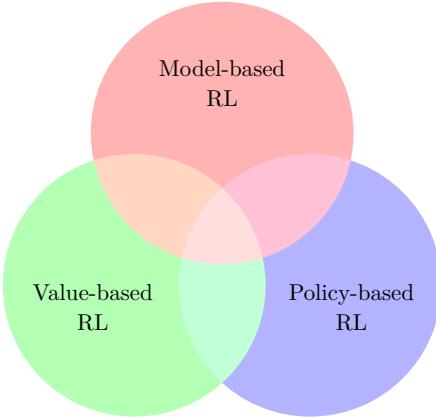


Figure 6.2: Venn diagram in the space of possible RL algorithms.

We now describe how it is possible to obtain advantages from both worlds by integrating learning and planning into one end-to-end training procedure so as to obtain an efficient algorithm both in performance (sample efficient) and in computation time. A Venn diagram of the different combinations is given in Figure 6.2.

When the model is available, one direct approach is to use tree search techniques that make use of both value and policy networks (e.g., Silver *et al.*, 2016b). When the model is not available and under the assumption that the agent has only access to a limited number of trajectories, the key property is to have an algorithm that generalizes well (see Chapter 7 for a discussion on generalization). One possibility is to build a model that is used to generate additional samples for a model-free reinforcement learning algorithm (Gu *et al.*, 2016b). Another possibility is to use a model-based approach along with a controller such as MPC to perform basic tasks and use model-free fine-tuning in order to achieve task success (Nagabandi *et al.*, 2017).

Other approaches build neural network architectures that combine both model-free and model-based elements. For instance, it is possible to combine a value function with steps of back-propagation through a model (Heess *et al.*, 2015). The VIN architecture (Tamar *et al.*, 2016)

is a fully differentiable neural network with a planning module that learns to plan from model-free objectives (given by a value function). It works well for tasks that involve planning-based reasoning (navigation tasks) from one initial position to one goal position and it demonstrates strong generalization in a few different domains.

In the same spirit, the predictron (Silver *et al.*, 2016a) is aimed at developing a more generally applicable algorithm that is effective in the context of planning. It works by implicitly learning an internal model in an abstract state space, which is used for policy evaluation. The predictron is trained end-to-end to learn, from the abstract state space, (i) the immediate reward and (ii) value functions over multiple planning depths. The predictron architecture is limited to policy evaluation, but the idea was extended to an algorithm that can learn an optimal policy in an architecture called VPN (Oh *et al.*, 2017). Since VPN relies on n-step Q-learning, it requires however on-policy data.

Other works have proposed architectures that combine model-based and model-free approaches. Schema Networks (Kansky *et al.*, 2017) learn the dynamics of an environment directly from data by enforcing some relational structure. The idea is to use a richly structured architecture such that it provides robust generalization thanks to an object-oriented approach for the model.

I2As (Weber *et al.*, 2017) does not use the model to directly perform planning but it uses the predictions as additional context in deep policy networks. The proposed idea is that I2As could learn to interpret predictions from the learned model to construct implicit plans.

TreeQN (Farquhar *et al.*, 2017) constructs a tree by recursively applying an implicit transition model in an implicitly learned abstract state space, built by estimating Q-values. Farquhar *et al.* (2017) also propose ATreeC, which is an actor-critic variant that augments TreeQN with a softmax layer to form a stochastic policy network.

The CRAR agent explicitly learns both a value function and a model via a shared low-dimensional learned encoding of the environment, which is meant to capture summarized abstractions and allow for efficient planning (François-Lavet *et al.*, 2018). By forcing an expressive representation, the CRAR approach creates an interpretable low-

dimensional representation of the environment, even far temporally from any rewards or in the absence of model-free objectives.

Improving the combination of model-free and model-based ideas is one key area of research for the future development of deep RL algorithms. We therefore expect to see smarter and richer structures in that domain.

7

The concept of generalization

Generalization is a central concept in the field of machine learning, and reinforcement learning is no exception. In an RL algorithm (model-free or model-based), generalization refers to either

- the capacity to achieve good performance in an environment where limited data has been gathered, or
- the capacity to obtain good performance in a related environment.

In the former case, the agent must learn how to behave in a test environment that is identical to the one it has been trained on. In that case, the idea of generalization is directly related to the notion of *sample efficiency* (e.g., when the state-action space is too large to be fully visited). In the latter case, the test environment has common patterns with the training environment but can differ in the dynamics and the rewards. For instance, the underlying dynamics may be the same but a transformation on the observations may have happened (e.g., noise, shift in the features, etc.). That case is related to the idea of transfer learning (discussed in §10.2) and meta-learning (discussed in §10.1.2).

Note that, in the online setting, one mini-batch gradient update is usually done at every step. In that case, the community has also used the term sample efficiency to refer to how fast the algorithm learns, which is measured in terms of performance for a given number of steps (number of learning steps=number of transitions observed). However, in that context, the result depends on many different elements. It depends on the learning algorithm and it is, for instance, influenced by the possible variance of the target in a model-free setting. It also depends on the exploration/exploitation, which will be discussed in §8.1 (e.g, instabilities may be good). Finally, it depends on the actual generalization capabilities.

In this chapter, the goal is to study specifically the aspect of generalization. We are not interested in the number of mini-batch gradient descent steps that are required but rather in the performance that a deep RL algorithm can have in the offline case where the agent has to learn from limited data. Let us consider the case of a finite dataset D obtained on the exact same task as the test environment. Formally, a dataset available to the agent $D \sim \mathcal{D}$ can be defined as a set of four-tuples $\langle s, a, r, s' \rangle \in \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times \mathcal{S}$ gathered by sampling independently and identically (i.i.d.)¹

- a given number of state-action pairs (s, a) from some fixed distribution with $\mathbb{P}(s, a) > 0, \forall (s, a) \in \mathcal{S} \times \mathcal{A}$,
- a next state $s' \sim T(s, a, \cdot)$, and
- a reward $r = R(s, a, s')$.

We denote by D_∞ the particular case of a dataset D where the number of tuples tends to infinity.

A learning algorithm can be seen as a mapping of a dataset D into a policy π_D (independently of whether the learning algorithm has a

¹That i.i.d. assumption can, for instance, be obtained from a given distribution of initial states by following a stochastic sampling policy that ensures a non-zero probability of taking any action in any given state. That sampling policy should be followed during at least H time steps with the assumption that all states of the MDP can be reached in a number of steps smaller than H from the given distribution of initial states.

model-based or a model-free approach). In that case, we can decompose the suboptimality of the expected return as follows:

$$\begin{aligned}
 \mathbb{E}_{D \sim \mathcal{D}} [V^{\pi^*}(s) - V^{\pi_D}(s)] &= \mathbb{E}_{D \sim \mathcal{D}} [V^{\pi^*}(s) - V^{\pi_{D^\infty}}(s) + V^{\pi_{D^\infty}}(s) - V^{\pi_D}(s)] \\
 &= \underbrace{(V^{\pi^*}(s) - V^{\pi_{D^\infty}}(s))}_{\text{asymptotic bias}} \\
 &\quad + \underbrace{\mathbb{E}_{D \sim \mathcal{D}} [V^{\pi_{D^\infty}}(s) - V^{\pi_D}(s)]}_{\text{error due to finite size of the dataset } D} .
 \end{aligned} \tag{7.1}$$

This decomposition highlights two different terms: (i) an asymptotic bias which is independent of the quantity of data and (ii) an overfitting term directly related to the fact that the amount of data is limited. The goal of building a policy π_D from a dataset D is to obtain the lowest overall suboptimality. To do so, the RL algorithm should be well adapted to the task (or the set of tasks).

In the previous section, two different types of approaches (model-based and model-free) have been discussed, as well as how to combine them. We have discussed the algorithms that can be used for different approaches but we have in fact left out many important elements that have an influence on the bias-overfitting tradeoff (e.g., Zhang *et al.*, 2018c; Zhang *et al.*, 2018a for illustrations of overfitting in deep RL).

As illustrated in Figure 7.1, improving generalization can be seen as a tradeoff between (i) an error due to the fact that the algorithm trusts completely the frequentist assumption (i.e., discards any uncertainty on the limited data distribution) and (ii) an error due to the bias introduced to reduce the risk of overfitting. For instance, the function approximator can be seen as a form of structure introduced to force some generalization, at the risk of introducing a bias. When the quality of the dataset is low, the learning algorithm should favor more robust policies (i.e., consider a smaller class of policies with stronger generalization capabilities). When the quality of the dataset increases, the risk of overfitting is lower and the learning algorithm can trust the data more, hence reducing the asymptotic bias.

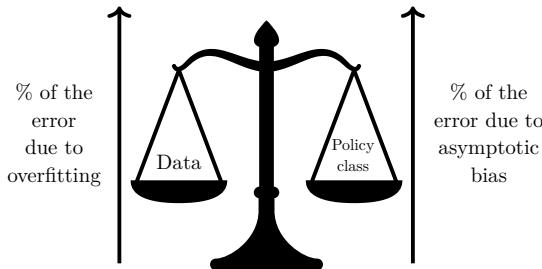


Figure 7.1: Schematic representation of the bias-overfitting tradeoff.

As we will see, for many algorithmic choices, there is in practice a tradeoff to be made between asymptotic bias and overfitting that we simply call "bias-overfitting tradeoff". In this section, we discuss the following key elements that are at stake when one wants to improve generalization in deep RL:

- the state representation,
- the learning algorithm (type of function approximator and model-free vs model-based),
- the objective function (e.g., reward shaping, tuning the training discount factor), and
- using hierarchical learning.

Throughout those discussions, a simple example is considered. This example is, by no means, representative of the complexity of real-world problems but it is enlightening to simply illustrate the concepts that will be discussed. Let us consider an MDP with N_S states, $N_S = 11$ and N_A actions, $N_A = 4$. Let us suppose that the main part of the environment is a square 3×3 grid world (each represented by a tuple (x, y) with $x = \{0, 1, 2\}, y = \{0, 1, 2\}$), such as illustrated in Figure 7.2. The agent starts in the central state $(1, 1)$. In every state, it selects one of the 4 actions corresponding to 4 cardinal directions (up, down, left and right), which leads the agent to transition deterministically in a state immediately next to it, except when it tries to move out of the domain. On the upper part and lower part of the domain, the agent is

stuck in the same state if it tries to move out of the domain. On the left, the agent transitions deterministically to a given state, which will provide a reward of 0.6 for any action at the next time step. On the right side of the square, the agent transitions with a probability 25% to another state that will provide, at the next time step, a reward of 1 for any action (the rewards are 0 for all other states). When a reward is obtained, the agent transitions back to the central state.

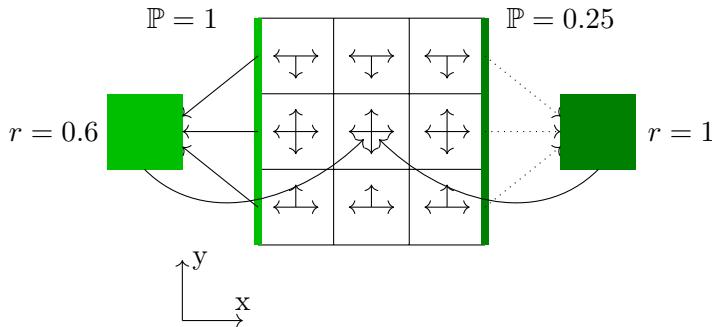


Figure 7.2: Representation of a simple MDP that illustrates the need of generalization.

In this example, if the agent has perfect knowledge of its environment, the best expected cumulative reward (for a discount factor close to 1) would be to always go to the left direction and repeatedly gather a reward of 0.6 every 3 steps (as compared to gathering a reward of 1 every 6 steps on average). Let us now suppose that only limited information has been obtained on the MDP with only one tuple of experience $\langle s, a, r, s' \rangle$ for each couple $\langle s, a \rangle$. According to the limited data in the frequentist assumption, there is a rather high probability ($\sim 58\%$) that at least one transition from the right side seems to provide a deterministic access to $r = 1$. In those cases and for either a model-based or a model-free approach, if the learning algorithm comes up with the optimal policy in an empirical MDP built from the frequentist statistics, it would actually suffer from poor generalization as it would choose to try to obtain the reward $r = 1$.

We discuss hereafter the different aspects that can be used to avoid overfitting to limited data; we show that it is done by favoring robust

policies within the policy class, usually at the expense of introducing some bias. At the end, we also discuss how the bias-overfitting tradeoff can be used in practice to obtain the best performance from limited data.

7.1 Feature selection

The idea of selecting the right features for the task at hand is key in the whole field of machine learning and also highly prevalent in reinforcement learning (see e.g., Munos and Moore, 2002; Ravindran and Barto, 2004; Leffler *et al.*, 2007; Kroon and Whiteson, 2009; Dinculescu and Precup, 2010; Li *et al.*, 2011; Ortner *et al.*, 2014; Mandel *et al.*, 2014; Jiang *et al.*, 2015a; Guo and Brunskill, 2017; François-Lavet *et al.*, 2017). The appropriate level of abstraction plays a key role in the bias-overfitting tradeoff and one of the key advantages of using a small but rich abstract representation is to allow for improved generalization.

Overfitting When considering many features on which to base the policy (in the example the y-coordinate of the state as illustrated in Figure 7.3), an RL algorithm may take into consideration spurious correlations, which leads to overfitting (in the example, the agent may infer that the y-coordinate changes something to the expected return because of the limited data).

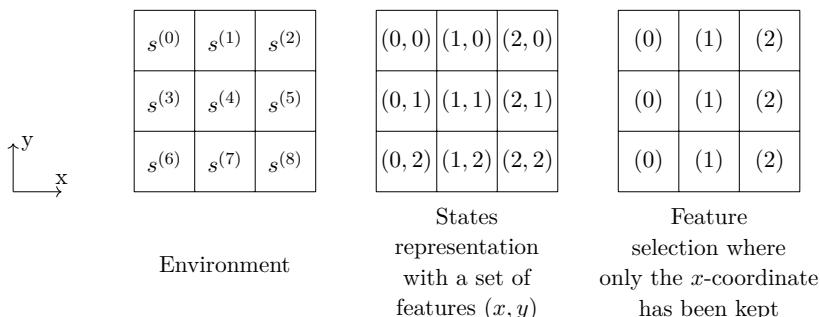


Figure 7.3: Illustration of the state representation and feature selection process. In this case, after the feature selection process, all states with the same x -coordinate are considered as indistinguishable.

7.2. Choice of the learning algorithm and function approximator selection

Asymptotic bias Removing features that discriminate states with a very different role in the dynamics introduces an asymptotic bias. Indeed, the same policy would be enforced on undistinguishable states, hence leading to a sub-optimal policy.

In deep RL, one approach is to first infer a factorized set of generative factors from the observations. This can be done for instance with an encoder-decoder architecture variant (Higgins *et al.*, 2017; Zhang *et al.*, 2018b). These features can then be used as inputs to a reinforcement learning algorithm. The learned representation can, in some contexts, greatly help for generalization as it provides a more succinct representation that is less prone to overfitting. However, an auto-encoder is often too strong of a constraint. On the one hand, some features may be kept in the abstract representation because they are important for the reconstruction of the observations, though they are otherwise irrelevant for the task at hand (e.g., the color of the cars in a self-driving car context). On the other hand, crucial information about the scene may also be discarded in the latent representation, particularly if that information takes up a small proportion of the observations x in pixel space (Higgins *et al.*, 2017). Note that in the deep RL setting, the abstraction representation is intertwined with the use of deep learning. This is discussed in detail in the following section.

7.2 Choice of the learning algorithm and function approximator selection

The function approximator in deep learning characterizes how the features will be treated into higher levels of abstraction (a fortiori it can thus give more or less weight to some features). If there is, for instance, an attention mechanism in the first layers of a deep neural network, the mapping made up of those first layers can be seen as a feature selection mechanism.

On the one hand, if the function approximator used for the value function and/or the policy and/or the model is too simple, an asymptotic bias may appear. On the other hand, when the function approximator has poor generalization, there will be a large error due to the finite size

of the dataset (overfitting). In the example above, a particularly good choice of a model-based or model-free approach associated with a good choice of a function approximator could infer that the y-coordinate of the state is less important than the x-coordinate, and generalize that to the policy.

Depending on the task, finding a performant function approximator is easier in either a model-free or a model-based approach. The choice of relying more on one or the other approach is thus also a crucial element to improve generalization, as discussed in §6.2.

One approach to mitigate non-informative features is to force the agent to acquire a set of symbolic rules adapted to the task and to reason on a more abstract level. This abstract level reasoning and the improved generalization have the potential to induce high-level cognitive functions such as transfer learning and analogical reasoning (Garnelo *et al.*, 2016). For instance, the function approximator may embed a relational learning structure (Santoro *et al.*, 2017) and thus build on the idea of relational reinforcement learning (Džeroski *et al.*, 2001).

7.2.1 Auxiliary tasks

In the context of deep reinforcement learning, it was shown by Jaderberg *et al.* (2016) that augmenting a deep reinforcement learning agent with auxiliary tasks within a jointly learned representation can drastically improve sample efficiency in learning. This is done by maximizing simultaneously many pseudo-reward functions such as immediate reward prediction ($\gamma = 0$), predicting pixel changes in the next observation, or predicting activation of some hidden unit of the agent’s neural network. The argument is that learning related tasks introduces an inductive bias that causes a model to build features in the neural network that are useful for the range of tasks (Ruder, 2017). Hence, this formation of more significant features leads to less overfitting.

In deep RL, it is possible to build an abstract state such that it provides sufficient information for simultaneously fitting an internal meaningful dynamics as well as the estimation of the expected value of an optimal policy. By explicitly learning both the model-free and model-based components through the state representation, along with an

approximate entropy maximization penalty, the CRAR agent (François-Lavet *et al.*, 2018) shows how it is possible to learn a low-dimensional representation of the task. In addition, this approach can directly make use of a combination of model-free and model-based, with planning happening in a smaller latent state space.

7.3 Modifying the objective function

In order to improve the policy learned by a deep RL algorithm, one can optimize an objective function that diverts from the actual objective. By doing so, a bias is usually introduced but this can in some cases help with generalization. The main approaches to modify the objective function are either (i) to modify the reward of the task to ease learning (reward shaping), or (ii) tune the discount factor at training time.

7.3.1 Reward shaping

Reward shaping is a heuristic for faster learning. In practice, reward shaping uses prior knowledge by giving intermediate rewards for actions that lead to desired outcome. It is usually formalized as a function $F(s, a, s')$ added to the original reward function $R(s, a, s')$ of the original MDP (Ng *et al.*, 1999). This technique is often used in deep reinforcement learning to improve the learning process in settings with sparse and delayed rewards (e.g., Lample and Chaplot, 2017).

7.3.2 Discount factor

When the model available to the agent is estimated from data, the policy found using a shorter planning horizon can actually be better than a policy learned with the true horizon (Petrik and Scherrer, 2009; Jiang *et al.*, 2015b). On the one hand, artificially reducing the planning horizon leads to a bias since the objective function is modified. On the other hand, if a long planning horizon is targeted (the discount factor γ is close to 1), there is a higher risk of overfitting. This overfitting can intuitively be understood as linked to the accumulation of the errors in the transitions and rewards estimated from data as compared to the actual transition and reward probabilities. In the example above

(Figure 7.2), in the case where the upper right or lower right states would seem to lead deterministically to $r = 1$ from the limited data, one may take into account that it requires more steps and thus more uncertainty on the transitions (and rewards). In that context, a low training discount factor would reduce the impact of rewards that are temporally distant. In the example, a discount factor close to 0 would discount the estimated rewards at three time steps much more strongly than the rewards two time steps away, hence practically discarding the potential rewards that can be obtained by going through the corners as compared to the ones that only require moving along the x-axis.

In addition to the bias-overfitting tradeoff, a high discount factor also requires specific care in value iteration algorithms as it can lead to instabilities in convergence. This effect is due to the mappings used in the value iteration algorithms with bootstrapping (e.g., Equation 4.2 for the Q-learning algorithm) that propagate errors more strongly with a high discount factor. This issue is discussed by Gordon (1999) with the notion of *non-expansion/expansion mappings*. When bootstrapping is used in a deep RL value iteration algorithm, the risk of instabilities and overestimation of the value function is empirically stronger for a discount factor close to one (François-Lavet *et al.*, 2015).

7.4 Hierarchical learning

The possibility of learning temporally extended actions (as opposed to atomic actions that last for one time-step) has been formalized under the name of options (Sutton *et al.*, 1999). Similar ideas have also been denoted in the literature as macro-actions (McGovern *et al.*, 1997) or abstract actions (Hauskrecht *et al.*, 1998). The usage of options is an important challenge in RL because it is essential when the task at hand requires working on long time scales while developing generalization capabilities and easier transfer learning between the strategies. A few recent works have brought interesting results in the context of fully differentiable (hence learnable in the context of deep RL) options discovery. In the work of Bacon *et al.*, 2016, an *option-critic* architecture is presented with the capability of learning simultaneously the internal policies and the termination conditions of

options, as well as the policy over options. In the work of Vezhnevets *et al.*, 2016, the deep recurrent neural network is made up of two main elements. The first module generates an action-plan (stochastic plan of future actions) while the second module maintains a commitment-plan which determines when the action-plan has to be updated or terminated. Many variations of these approaches are also of interest (e.g., Kulkarni *et al.*, 2016; Mankowitz *et al.*, 2016). Overall, building a learning algorithm that is able to do hierarchical learning can be a good way of constraining/favoring some policies that have interesting properties and thus improving generalization.

7.5 How to obtain the best bias-overfitting tradeoff

From the previous sections, it is clear that there is a large variety of algorithmic choices and parameters that have an influence on the bias-overfitting tradeoff (including the choice of approach between model-based and model-free). An overall combination of all these elements provides a low overall sub-optimality.

For a given algorithmic parameter setting and keeping all other things equal, the right level of complexity is the one at which the increase in bias is equivalent to the reduction of overfitting (or the increase in overfitting is equivalent to the reduction of bias). However, in practice, there is usually not an analytical way to find the right tradeoffs to be made between all the algorithmic choices and parameters. Still, there are a variety of practical strategies that can be used. We now discuss them for the batch setting case and the online setting case.

7.5.1 Batch setting

In the batch setting case, the selection of the policy parameters to effectively balance the bias-overfitting tradeoff can be done similarly to that in supervised learning (e.g., cross-validation) as long as the performance criterion can be estimated from a subset of the trajectories from the dataset D not used during training (i.e., a validation set).

One approach is to fit an MDP model to the data via regression (or simply use the frequentist statistics for finite state and action space).

The empirical MDP can then be used to evaluate the policy. This purely model-based estimator has alternatives that do not require fitting a model. One possibility is to use a policy evaluation step obtained by generating artificial trajectories from the data, without explicitly referring to a model, thus designing a Model-free Monte Carlo-like (MFMC) estimator (Fonteneau *et al.*, 2013). Another approach is to use the idea of *importance sampling* that lets us obtain an estimate of $V^\pi(s)$ from trajectories that come from a behavior policy $\beta \neq \pi$, where β is assumed to be known (Precup, 2000). That approach is unbiased but the variance usually grows exponentially in horizon, which renders the method unsuitable when the amount of data is low. A mix of the regression-based approach and the importance sampling approach is also possible (Jiang and Li, 2016; Thomas and Brunskill, 2016), and the idea is to use a *doubly-robust estimator* that is both unbiased and with a lower variance than the importance sampling estimators.

Note that there exists a particular case where the environment's dynamics are known to the agent, but contain a dependence on an exogenous time series (e.g., trading in energy markets, weather-dependent dynamics) for which the agent only has finite data. In that case, the exogenous signal can be broken down in training time series and validation time series (François-Lavet *et al.*, 2016). This allows training on the environment with the training time series and this allows estimating any policy on the environment with the validation time series.

7.5.2 Online setting

In the online setting, the agent continuously gathers new experience. The bias-overfitting tradeoff still plays a key role at each stage of the learning process in order to achieve good sampling efficiency. Indeed, a performant policy from given data is part of the solution to an efficient exploration/exploitation tradeoff. For that reason, progressively fitting a function approximator as more data becomes available can in fact be understood as a way to obtain a good bias-overfitting tradeoff throughout learning. With the same logic, progressively increasing the discount factor allows optimizing the bias-overfitting tradeoff through

learning (François-Lavet *et al.*, 2015). Besides, optimizing the bias-overfitting tradeoff also suggests the possibility to dynamically adapt the feature space and/or the function approximator. For example, this can be done through ad hoc regularization, or by adapting the neural network architecture, using for instance the NET2NET transformation (Chen *et al.*, 2015).

8

Particular challenges in the online setting

As discussed in the introduction, reinforcement learning can be used in two main settings: (i) the batch setting (also called offline setting), and (ii) the online setting. In a batch setting, the whole set of transitions (s, a, r, s') to learn the task is fixed. This is in contrast to the *online* setting where the agent can gather new experience gradually. In the online setting, two specific elements have not yet been discussed in depth. First, the agent can influence how to gather experience so that it is the most useful for learning. This is the *exploration/exploitation* dilemma that we discuss in Section 8.1. Second, the agent has the possibility to use a replay memory (Lin, 1992) that allows for a good data-efficiency. We discuss in Section 8.2 what experience to store and how to reprocess that experience.

8.1 Exploration/Exploitation dilemma

The exploration-exploitation dilemma is a well-studied tradeoff in RL (e.g., Thrun, 1992). Exploration is about obtaining information about the environment (transition model and reward function) while exploitation is about maximizing the expected return given the current knowledge. As an agent starts accumulating knowledge about its environment, it

has to make a tradeoff between learning more about its environment (exploration) or pursuing what seems to be the most promising strategy with the experience gathered so far (exploitation).

8.1.1 Different settings in the exploration/exploitation dilemma

There exist mainly two different settings. In the first setting, the agent is expected to perform well without a separate training phase. Thus, an explicit tradeoff between exploration versus exploitation appears so that the agent should explore only when the learning opportunities are valuable enough for the future to compensate what direct exploitation can provide. The sub-optimality $\mathbb{E}_{s_0} V^*(s_0) - V^\pi(s_0)$ of an algorithm obtained in this context is known as the *cumulative regret*¹. The deep RL community is usually not focused on this case, except when explicitly stated such as in the works of Wang *et al.* (2016a) and Duan *et al.* (2016b).

In the more common setting, the agent is allowed to follow a *training policy* during a first phase of interactions with the environment so as to accumulate training data and hence learn a *test policy*. In the training phase, exploration is only constrained by the interactions it can make with the environment (e.g., a given number of interactions). The test policy should then be able to maximize a cumulative sum of rewards in a separate phase of interaction. The sub-optimality $\mathbb{E}_{s_0} V^*(s_0) - V^\pi(s_0)$ obtained in this case of setting is known as the *simple regret*. Note that an implicit exploration/exploitation is still important. On the one hand, the agent has to ensure that the lesser-known parts of the environment are not promising (exploration). On the other hand, the agent is interested in gathering experience in the most promising parts of the environment (which relates to exploitation) to refine the knowledge of the dynamics. For instance, in the bandit task provided in Figure 8.1, it should be clear with only a few samples that the option on the right is less promising and the agent should gather experience mainly on the two most promising arms to be able to discriminate the best one.

¹This term is mainly used in the bandit community where the agent is in only one state and where a distribution of rewards is associated to each action; see e.g., Bubeck *et al.*, 2011.

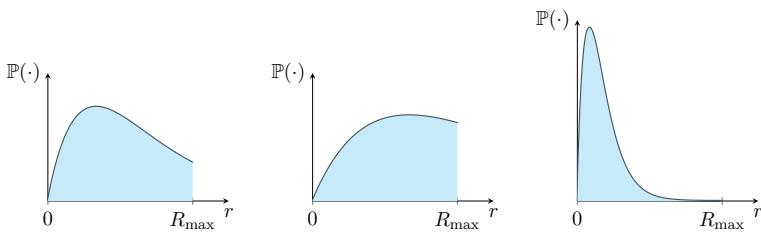


Figure 8.1: Illustration of the reward probabilities of 3 arms in a multi-armed bandit problem.

8.1.2 Different approaches to exploration

The exploration techniques are split into two main categories: (i) directed exploration and (ii) undirected exploration (Thrun, 1992).

In the undirected exploration techniques, the agent does not rely on any exploration specific knowledge of the environment (Thrun, 1992). For instance, the technique called ϵ -greedy takes a random action with probability ϵ and follows the policy that is believed to be optimal with probability $1 - \epsilon$. Other variants such as softmax exploration (also called Boltzmann exploration) takes an action with a probability that depends on the associated expected return.

Contrary to the undirected exploration, directed exploration techniques make use of a memory of the past interactions with the environment. For MDPs, directed exploration can scale polynomially with the size of the state space while undirected exploration scales in general exponentially with the size of the state space (e.g., E³ by Kearns and Singh, 2002; R-max by Brafman and Tennenholtz, 2003; ...). Inspired by the Bayesian setting, directed exploration can be done via heuristics of exploration bonus (Kolter and Ng, 2009) or by maximizing Shannon information gains (e.g., Sun *et al.*, 2011).

Directed exploration is, however, not trivially applicable in high-dimensional state spaces (e.g., Kakade *et al.*, 2003). With the development of the generalization capabilities of deep learning, some possibilities have been investigated. The key challenge is to handle, for high-dimensional spaces, the exploration/exploitation tradeoff in a principled way – with the idea to encourage the exploration of the environment

where the uncertainty due to limited data is the highest. When rewards are not sparse, a measure of the uncertainty on the value function can be used to drive the exploration (Dearden *et al.*, 1998; Dearden *et al.*, 1999). When rewards are sparse, this is even more challenging and exploration should in addition be driven by some novelty measures on the observations (or states in a Markov setting).

Before discussing the different techniques that have been proposed in the deep RL setting, one can note that the success of the first deep RL algorithms such as DQN also come from the exploration that arises naturally. Indeed, following a simple ϵ -greedy scheme online often proves to be already relatively efficient thanks to the natural instability of the Q-network that drives exploration (see Chapter 4 for why there are instabilities when using bootstrapping in a fitted Q-learning algorithm with neural networks).

Different improvements are directly built on that observation. For instance, the method of "Bootstrapped DQN" (Osband *et al.*, 2016) makes an explicit use of randomized value functions. Along similar lines, efficient exploration has been obtained by the induced stochasticity of uncertainty estimates given by a dropout Q-network (Gal and Ghahramani, 2016) or parametric noise added to its weights (Lipton *et al.*, 2016; Plappert *et al.*, 2017; Fortunato *et al.*, 2017). One specificity of the work done by Fortunato *et al.*, 2017 is that, similarly to Bayesian deep learning, the variance parameters are learned by gradient descent from the reinforcement learning loss function.

Another common approach is to have a directed scheme thanks to *exploration rewards* given to the agent via heuristics that estimate novelty (Schmidhuber, 2010; Stadie *et al.*, 2015; Houthooft *et al.*, 2016). In (Bellemare *et al.*, 2016; Ostrovski *et al.*, 2017), an algorithm provides the notion of novelty through a pseudo-count from an arbitrary density model that provides an estimate of how many times an action has been taken in similar states. This has shown good results on one of the most difficult Atari 2600 games, Montezuma's Revenge.

In (Florensa *et al.*, 2017), useful skills are learned in pre-training environments, which can then be utilized in the actual environment to improve exploration and train a high-level policy over these skills. Similarly, an agent that learns a set of auxiliary tasks may use them to

efficiently explore its environment (Riedmiller *et al.*, 2018). These ideas are also related to the creation of options studied in (Machado *et al.*, 2017a), where it is suggested that exploration may be tackled by learning options that lead to specific modifications in the state representation derived from proto-value functions.

Exploration strategies can also make use of a model of the environment along with planning. In that case, a strategy investigated in (Salge *et al.*, 2014; Mohamed and Rezende, 2015; Gregor *et al.*, 2016; Chiappa *et al.*, 2017) is to have the agent choose a sequence of actions by planning that leads to a representation of state as different as possible to the current state. In (Pathak *et al.*, 2017; Haber *et al.*, 2018), the agent optimizes both a model of its environment and a separate model that predicts the error/uncertainty of its own model. The agent can thus seek to take actions that adversarially challenge its knowledge of the environment (Savinov *et al.*, 2018).

By providing rewards on unfamiliar states, it is also possible to explore efficiently the environments. To determine the bonus, the current observation can be compared with the observations in memory. One approach is to define the rewards based on how many environment steps it takes to reach the current observation from those in memory (Savinov *et al.*, 2018). Another approach is to use a bonus positively correlated to the error of predicting features from the observations (e.g., features given by a fixed randomly initialized neural network) (Burda *et al.*, 2018).

Other approaches require either demonstrations or guidance from human demonstrators. One line of work suggests using natural language to guide the agent by providing exploration bonuses when an instruction is correctly executed (Kaplan *et al.*, 2017). In the case where demonstrations from expert agents are available, another strategy for guiding exploration in these domains is to imitate good trajectories. In some cases, it is possible to use demonstrations from experts even when they are given in an environment setup that is not exactly the same (Aytar *et al.*, 2018).

8.2 Managing experience replay

In online learning, the agent has the possibility to use a replay memory (Lin, 1992) that allows for data-efficiency by storing the past experience of the agent in order to have the opportunity to reprocess it later. In addition, a replay memory also ensures that the mini-batch updates are done from a reasonably stable data distribution kept in the replay memory (for N_{replay} sufficiently large) which helps for convergence/stability. This approach is particularly well-suited in the case of off-policy learning as using experience from past (i.e. different) policies does not introduce any bias (usually it is even good for exploration). In that context, methods based for instance on a DQN learning algorithm or model-based learning can safely and efficiently make use of a replay memory. In an online setting, the replay memory keeps all information for the last $N_{\text{replay}} \in \mathbb{N}$ time steps, where N_{replay} is constrained by the amount of memory available.

While a replay memory allows processing the transitions in a different order than they are experienced, there is also the possibility to use prioritized replay. This allows for consideration of the transitions with a different frequency than they are experienced depending on their significance (that could be which experience to store and which ones to replay). In (Schaul *et al.*, 2015b), the prioritization increases with the magnitude of the transitions' TD error, with the aim that the "unexpected" transitions are replayed more often.

A disadvantage of prioritized replay is that, in general, it also introduces a bias; indeed, by modifying the apparent probabilities of transitions and rewards, the expected return gets biased. This can readily be understood by considering the simple example illustrated in Figure 8.2 where an agent tries to estimate the expected return for a given tuple $\langle s, a \rangle$. In that example, a cumulative return of 0 is obtained with probability $1 - \epsilon$ (from next state $s^{(1)}$) while a cumulative return of $C > 0$ is obtained with probability ϵ (from next state $s^{(2)}$). In that case, using prioritized experience replay will bias the expected return towards a value higher than ϵC since any transition leading to $s^{(2)}$ will be replayed with a probability higher than ϵ .

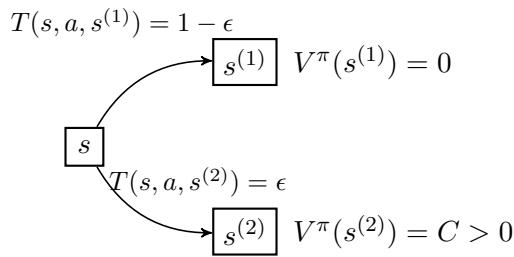


Figure 8.2: Illustration of a state s where for a given action a , the value of $Q^\pi(s, a; \theta)$ would be biased if prioritized experience replay is used ($\epsilon \ll 1$).

Note that this bias can be partly or completely corrected using weighted importance sampling, and this correction is important near convergence at the end of training (Schaul *et al.*, 2015b).

9

Benchmarking Deep RL

Comparing deep learning algorithms is a challenging problem due to the stochastic nature of the learning process and the narrow scope of the datasets examined during algorithm comparisons. This problem is exacerbated in deep reinforcement learning. Indeed, deep RL involves both stochasticity in the environment and stochasticity inherent to model learning, which makes ensuring fair comparisons and reproducibility especially difficult. To this end, simulations of many sequential decision-making tasks have been created to serve as benchmarks. In this section, we present several such benchmarks. Next, we give key elements to ensure consistency and reproducibility of experimental results. Finally, we also discuss some open-source implementations for deep RL algorithms.

9.1 Benchmark Environments

9.1.1 Classic control problems

Several classic control problems have long been used to evaluate reinforcement learning algorithms. These include balancing a pole on a cart (Cartpole) (Barto *et al.*, 1983), trying to get a car

up a mountain using momentum (Mountain Car) (Moore, 1990), swinging a pole up using momentum and subsequently balancing it (Acrobot) (Sutton and Barto, 1998). These problems have been commonly used as benchmarks for tabular RL and RL algorithms using linear function approximators (Whiteson *et al.*, 2011). Nonetheless, these simple environments are still sometimes used to benchmark deep RL algorithms (Ho and Ermon, 2016; Duan *et al.*, 2016a; Lillicrap *et al.*, 2015).

9.1.2 Games

Board-games have also been used for evaluating artificial intelligence methods for decades (Shannon, 1950; Turing, 1953; Samuel, 1959; Sutton, 1988; Littman, 1994; Schraudolph *et al.*, 1994; Tesauro, 1995; Campbell *et al.*, 2002). In recent years, several notable works have stood out in using deep RL for mastering Go (Silver *et al.*, 2016b) or Poker (Brown and Sandholm, 2017; Moravčík *et al.*, 2017).

In parallel to the achievements in board games, video games have also been used to further investigate reinforcement learning algorithms. In particular,

- many of these games have large observation space and/or large action space;
- they are often non-Markovian, which require specific care (see §10.1);
- they also usually require very long planning horizons (e.g., due to sparse rewards).

Several platforms based on video games have been popularized. The Arcade Learning Environment (ALE) (Bellemare *et al.*, 2013) was developed to test reinforcement algorithms across a wide range of different tasks. The system encompasses a suite of iconic Atari games, including Pong, Asteroids, Montezuma’s Revenge, etc. Figure 9.1 shows sample frames from some of these games. On most of the Atari games, deep RL algorithms have reached super-human level (Mnih *et al.*, 2015). Due to the similarity in state and action spaces between different Atari

games or different variants of the same game, they are also a good test-bed for evaluating generalization of reinforcement learning algorithms (Machado *et al.*, 2017b), multi-task learning (Parisotto *et al.*, 2015) and for transfer learning (Rusu *et al.*, 2015).

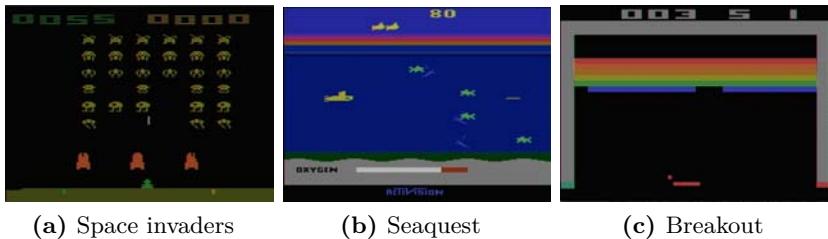


Figure 9.1: Illustration of three Atari games.

The General Video Game AI (GVGAI) competition framework (Perez-Liebana *et al.*, 2016) was created and released with the purpose of providing researchers a platform for testing and comparing their algorithms on a large variety of games and under different constraints. The agents are required to either play multiple unknown games with or without access to game simulations, or to design new game levels or rules.

VizDoom (Kempka *et al.*, 2016) implements the Doom video game as a simulated environment for reinforcement learning. VizDoom has been used as a platform for investigations of reward shaping (Lample and Chaplot, 2017), curriculum learning (Wu and Tian, 2016), predictive planning (Dosovitskiy and Koltun, 2016), and meta-reinforcement learning (Duan *et al.*, 2016b).

The open-world nature of Minecraft also provides a convenient platform for exploring reinforcement learning and artificial intelligence. Project Malmo (Johnson *et al.*, 2016) is a framework that provides easy access to the Minecraft video game. The environment and framework provide layers of abstraction that facilitate tasks ranging from simple navigation to collaborative problem solving. Due to the nature of the simulation, several works have also investigated lifelong-learning, curriculum learning, and hierarchical planning using Minecraft as a

platform (Tessler *et al.*, 2017; Matiisen *et al.*, 2017; Branavan *et al.*, 2012; Oh *et al.*, 2016).

Similarly, Deepmind Lab (Beattie *et al.*, 2016) provides a 3D platform adapted from the Quake video game. The Labyrinth maze environments provided with the framework have been used in work on hierarchical, lifelong and curriculum learning (Jaderberg *et al.*, 2016; Mirowski *et al.*, 2016; Teh *et al.*, 2017).

Finally, “StarCraft II” (Vinyals *et al.*, 2017) and “Starcraft: Broodwar” (Wender and Watson, 2012; Synnaeve *et al.*, 2016) provide similar benefits in exploring lifelong-learning, curriculum learning, and other related hierarchical approaches. In addition, real-time strategy (RTS) games – as with the Starcraft series – are also an ideal testbed for multi-agent systems. Consequently, several works have investigated these aspects in the Starcraft framework (Foerster *et al.*, 2017b; Peng *et al.*, 2017a; Brys *et al.*, 2014).

9.1.3 Continuous control systems and robotics domains

While games provide a convenient platform for reinforcement learning, the majority of those environments investigate discrete action decisions. In many real-world systems, as in robotics, it is necessary to provide frameworks for continuous control.

In that setting, the MuJoCo (Todorov *et al.*, 2012) simulation framework is used to provide several locomotion benchmark tasks. These tasks typically involve learning a gait to move a simulated robotic agent as fast as possible. The action space is the amount of torque to apply to motors on the agents’ joints, while the observations provided are typically the joint angles and positions in the 3D space. Several frameworks have built on top of these locomotion tasks to provide hierarchical task environments (Duan *et al.*, 2016a) and multi-task learning platforms (Henderson *et al.*, 2017a).

Because the MuJoCo simulator is closed-source and requires a license, an open-source initiative called Roboschool (Schulman *et al.*, 2017b) provides the same locomotion tasks along with more complex tasks involving humanoid robot simulations (such as learning to run and chase a moving flag while being hit by obstacles impeding progress).

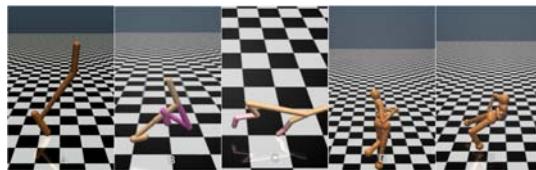


Figure 9.2: Screenshots from MuJoCo locomotion benchmark environments provided by OpenAI Gym.

These tasks allow for evaluation of complex planning in reinforcement learning algorithms.

Physics engines have also been used to investigate transfer learning to real-world applications. For instance, the Bullet physics engine (Coumans, Bai, *et al.*, 2016) has been used to learn locomotion skills in simulation, for character animation in games (Peng *et al.*, 2017b) or for being transferred to real robots (Tan *et al.*, 2018). This also includes manipulation tasks (Rusu *et al.*, 2016; Duan *et al.*, 2017) where a robotic arm stacks cubes in a given order. Several works integrate Robot Operating System (ROS) with physics engines (such as ODE, or Bullet) to provide RL-compatible access to near real-world robotic simulations (Zamora *et al.*, 2016; Ueno *et al.*, 2017). Most of them can also be run on real robotic systems using the same software.

There exists also a toolkit that leverages the Unity platform for creating simulation environments (Juliani *et al.*, 2018). This toolkit enables the development of learning environments that are rich in sensory and physical complexity and supports the multi-agent setting.

9.1.4 Frameworks

Most of the previously cited benchmarks have open-source code available. There also exists easy-to-use wrappers for accessing many different benchmarks. One such example is OpenAI Gym (Brockman *et al.*, 2016). This wrapper provides ready access to environments such as algorithmic, Atari, board games, Box2d games, classical control problems, MuJoCo robotics simulations, toy text problems, and others. Gym Retro¹ is a wrapper similar to OpenAI Gym and it provides over 1,000 games

¹<https://github.com/openai/retro>

across a variety of backing emulators. The goal is to study the ability of deep RL agents to generalize between games that have similar concepts but different appearances. Other frameworks such as μ niverse² and SerpentAI³ also provide wrappers for specific games or simulations.

9.2 Best practices to benchmark deep RL

Ensuring best practices in scientific experiments is crucial to continued scientific progress. Across various fields, investigations in reproducibility have found problems in numerous publications, resulting in several works providing experimental guidelines in proper scientific practices (Sandve *et al.*, 2013; Baker, 2016; Halsey *et al.*, 2015; Casadevall and Fang, 2010). To this end, several works have investigated proper metrics and experimental practices when comparing deep RL algorithms (Henderson *et al.*, 2017b; Islam *et al.*, 2017; Machado *et al.*, 2017b; Whiteson *et al.*, 2011).

Number of Trials, Random Seeds and Significance Testing

Stochasticity plays a large role in deep RL, both from randomness within initializations of neural networks and stochasticity in environments. Results may vary significantly simply by changing the random seed. When comparing the performance of algorithms, it is therefore important to run many trials across different random seeds.

In deep RL, it has become common to simply test an algorithm's effectiveness with an average across a few learning trials. While this is a reasonable benchmark strategy, techniques derived from significance testing (Demšar, 2006; Bouckaert and Frank, 2004; Bouckaert, 2003; Dietterich, 1998) have the advantage of providing statistically grounded arguments in favor of a given hypothesis. In practice for deep RL, significance testing can be used to take into account the standard deviation across several trials with different random seeds and environment conditions. For instance, a simple 2-sample t -test can give an idea of whether performance gains are significantly due to the algorithm performance

²<https://github.com/unixpickle/muniverse>

³<https://github.com/SerpentAI/SerpentAI>

or to noisy results in highly stochastic settings. In particular, while several works have used the top- K trials and simply presented those as performance gains, this has been argued to be inadequate for fair comparisons (Machado *et al.*, 2017b; Henderson *et al.*, 2017b).

In addition, one should be careful not to over-interpret the results. It is possible that a hypothesis can be shown to hold for one or several given environments and under one or several given set of hyperparameters, but fail in other settings.

Hyperparameter Tuning and Ablation Comparisons

Another important consideration is ensuring a fair comparison between learning algorithms. In this case, an ablation analysis compares alternate configurations across several trials with different random seeds. It is especially important to tune hyperparameters to the greatest extent possible for baseline algorithms. Poorly chosen hyperparameters can lead to an unfair comparison between a novel and a baseline algorithm. In particular, network architecture, learning rate, reward scale, training discount factor, and many other parameters can affect results significantly. Ensuring that a novel algorithm is indeed performing much better requires proper scientific procedure when choosing such hyperparameters (Henderson *et al.*, 2017b).

Reporting Results, Benchmark Environments, and Metrics

Average returns (or cumulative reward) across evaluation trajectories are often reported as a comparison metric. While some literature (Gu *et al.*, 2016a; Gu *et al.*, 2017c) has also used metrics such as average maximum return or maximum return within Z samples, these may be biased to make results for highly unstable algorithms appear more significant. For example, if an algorithm reaches a high maximum return quickly, but then diverges, such metrics would ensure this algorithm appears successful. When choosing metrics to report, it is important to select those that provide a fair comparison. If the algorithm performs better in average maximum return, but worse by using an average return metric, it is important to highlight both results and describe the benefits and shortcomings of such an algorithm (Henderson *et al.*, 2017b).

This is also applicable to the selection of which benchmark environments to report during evaluation. Ideally, empirical results should cover a large mixture of environments to determine in which settings an algorithm performs well and in which settings it does not. This is vital for determining real-world performance applications and capabilities.

9.3 Open-source software for Deep RL

A deep RL agent is composed of a learning algorithm (model-based or model-free) along with specific structure(s) of function approximator(s). In the online setting (more details are given in Chapter 8), the agent follows a specific exploration/exploitation strategy and typically uses a memory of its previous experience for sample efficiency.

While many papers release implementations of various deep RL algorithms, there also exist some frameworks built to facilitate the development of new deep RL algorithms or to apply existing algorithms to a variety of environments. We provide a list of some of the existing frameworks in Appendix A.1.

10

Deep reinforcement learning beyond MDPs

We have so far mainly discussed how an agent is able to learn how to behave in a given Markovian environment where all the interesting information (the state $s_t \in \mathcal{S}$) is obtained at every time step t . In this chapter, we discuss more general settings with (i) non-Markovian environments, (ii) transfer learning and (iii) multi-agent systems.

10.1 Partial observability and the distribution of (related) MDPs

In domains where the Markov hypothesis holds, it is straightforward to show that the policy need not depend on what happened at previous time steps to recommend an action (by definition of the Markov hypothesis). This section describes two different cases that complicate the Markov setting: the partially observable environments and the distribution of (related) environments.

Those two settings are at first sight quite different conceptually. However, in both settings, at each step in the sequential decision process, the agent may benefit from taking into account its whole observable history up to the current time step t when deciding what action to perform. In other words, a history of observations can be used as a pseudo-state (pseudo-state because that refers to a different and abstract

stochastic control process). Any missing information in the history of observations (potentially long before time t) can introduce a bias in the RL algorithm (as described in Chapter 7 when some features are discarded).

10.1.1 The partially observable scenario

In this setting, the agent only receives, at each time step, an observation of its environment that does not allow it to identify the state with certainty. A Partially Observable Markov Decision Process (POMDP) (Sondik, 1978; Kaelbling *et al.*, 1998) is a discrete time stochastic control process defined as follows:

Definition 10.1. A POMDP is a 7-tuple $(\mathcal{S}, \mathcal{A}, T, R, \Omega, O, \gamma)$ where:

- \mathcal{S} is a finite set of states $\{1, \dots, N_{\mathcal{S}}\}$,
- \mathcal{A} is a finite set of actions $\{1, \dots, N_{\mathcal{A}}\}$,
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition function (set of conditional transition probabilities between states),
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R}$ is the reward function, where \mathcal{R} is a continuous set of possible rewards in a range $R_{\max} \in \mathbb{R}^+$ (e.g., $[0, R_{\max}]$ without loss of generality),
- Ω is a finite set of observations $\{1, \dots, N_{\Omega}\}$,
- $O : \mathcal{S} \times \Omega \rightarrow [0, 1]$ is a set of conditional observation probabilities, and
- $\gamma \in [0, 1]$ is the discount factor.

The environment starts in a distribution of initial states $b(s_0)$. At each time step $t \in \mathbb{N}_0$, the environment is in a state $s_t \in \mathcal{S}$. At the same time, the agent receives an observation $\omega_t \in \Omega$ that depends on the state of the environment with probability $O(s_t, \omega_t)$, after which the agent chooses an action $a_t \in \mathcal{A}$. Then, the environment transitions to state $s_{t+1} \in \mathcal{S}$ with probability $T(s_t, a_t, s_{t+1})$ and the agent receives a reward $r_t \in \mathcal{R}$ equal to $R(s_t, a_t, s_{t+1})$.

When the full model (T , R and O) are known, methods such as Point-Based Value Iteration (PBVI) algorithm (Pineau *et al.*, 2003) for POMDP planning can be used to solve the problem. If the full POMDP model is not available, other reinforcement learning techniques have to be used.

A naive approach to building a space of candidate policies is to consider the set of mappings taking only the very last observation(s) as input. However, in a POMDP setting, this leads to candidate policies that are typically not rich enough to capture the system dynamics, thus suboptimal. In that case, the best achievable policy is stochastic (Singh *et al.*, 1994), and it can be obtained using policy gradient. The alternative is to use a history of previously observed features to better estimate the hidden state dynamics. We denote by $\mathcal{H}_t = \Omega \times (\mathcal{A} \times \mathcal{R} \times \Omega)^t$ the set of histories observed up to time t for $t \in \mathbb{N}_0$ (see Fig. 10.1), and by $\mathcal{H} = \bigcup_{t=0}^{\infty} \mathcal{H}_t$ the space of all possible observable histories.

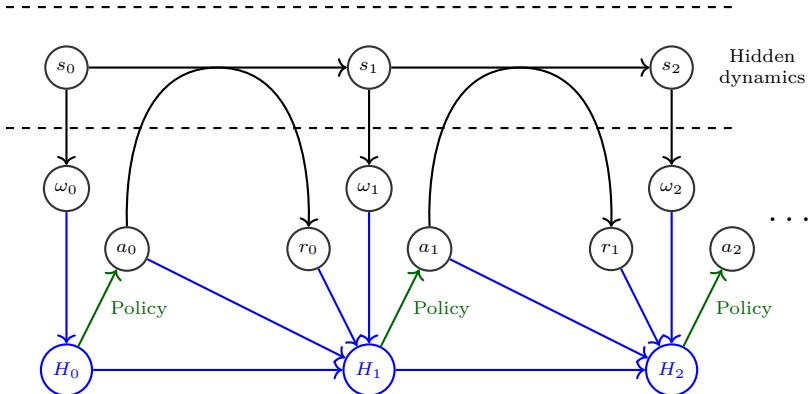


Figure 10.1: Illustration of a POMDP. The actual dynamics of the POMDP is depicted in dark while the information that the agent can use to select the action at each step is the whole history H_t depicted in blue.

A straightforward approach is to take the whole history $H_t \in \mathcal{H}$ as input (Braziuas, 2003). However, increasing the size of the set of candidate optimal policies generally implies: (i) more computation to search within this set (Singh *et al.*, 1994; McCallum, 1996) and, (ii) an increased risk of including candidate policies suffering from overfitting

due to lack of sufficient data, which thus leads to a bias-overfitting tradeoff when learning policies from data (François-Lavet *et al.*, 2017).

In the case of deep RL, the architectures used usually have a smaller number of parameters and layers than in supervised learning due to the more complex RL setting, but the trend of using ever smarter and complex architectures in deep RL happens similarly to supervised learning tasks. Architectures such as convolutional layers or recurrency are particularly well-suited to deal with a large input space because they offer interesting generalization properties. A few empirical successes on large scale POMDPs make use of convolutional layers (Mnih *et al.*, 2015) and/or recurrent layers (Hausknecht and Stone, 2015), such as LSTMs (Hochreiter and Schmidhuber, 1997).

10.1.2 The distribution of (related) environments

In this setting, the environment of the agent is a distribution of different (yet related) tasks that differ for instance in the reward function or in the probabilities of transitions from one state to another. Each task $T_i \sim \mathcal{T}$ can be defined by the observations $\omega_t \in \Omega$ (which are equal to s_t if the environments are Markov), the rewards $r_t \in \mathcal{R}$, as well as the effect of the actions $a_t \in \mathcal{A}$ taken at each step. Similarly to the partially observable context, we denote the history of observations by H_t , where $H_t \in \mathcal{H}_t = \Omega \times (\mathcal{A} \times \mathcal{R} \times \Omega)^t$. The agent aims at finding a policy $\pi(a_t|H_t; \theta)$ with the objective of maximizing its expected return, defined (in the discounted setting) as

$$\mathbb{E}_{T_i \sim \mathcal{T}} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid H_t, \pi \right].$$

An illustration of the general setting of meta learning on non-Markov environments is given in Figure 10.2.

Different approaches have been investigated in the literature. The Bayesian approach aims at explicitly modeling the distribution of the different environments, if a prior is available (Ghavamzadeh *et al.*, 2015). However, it is often intractable to compute the Bayesian-optimal strategy and one has to rely on more practical approaches that do not require an explicit model of the distribution. The concept of *meta-learning* or *learning to learn* aims at discovering, from experience, how

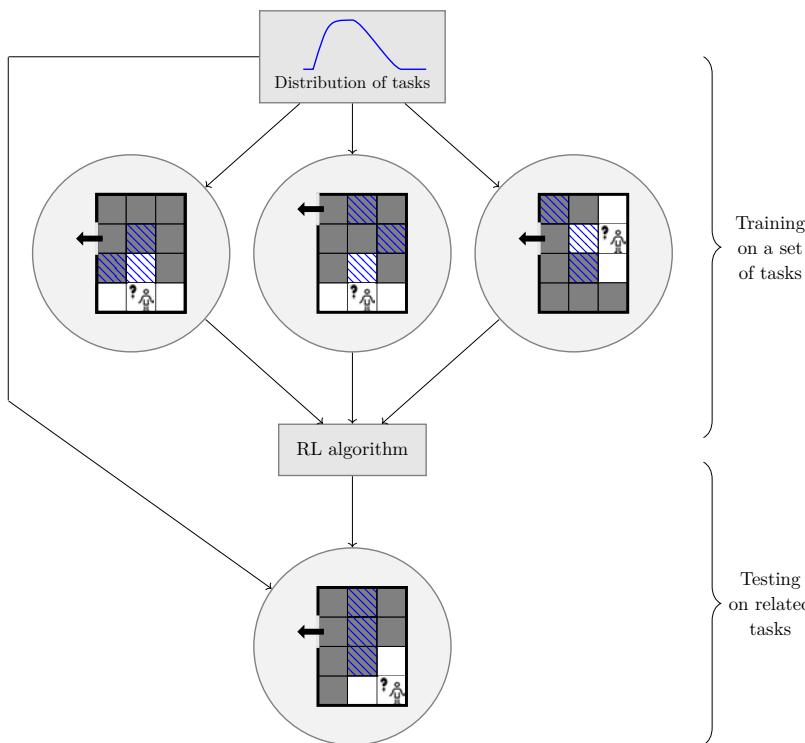


Figure 10.2: Illustration of the general setting of meta learning on POMDPs for a set of labyrinth tasks. In this illustration, it is supposed that the agent only sees the nature of the environment just one time step away from him.

to behave in a range of tasks and how to negotiate the exploration-exploitation tradeoff (Hochreiter *et al.*, 2001). In that case, deep RL techniques have been investigated by, e.g., Wang *et al.*, 2016a; Duan *et al.*, 2016b with the idea of using recurrent networks trained on a set of environments drawn i.i.d. from the distribution.

Some other approaches have also been investigated. One possibility is to train a neural network to imitate the behavior of known optimal policies on MDPs drawn from the distribution (Castronovo *et al.*, 2017). The parameters of the model can also be explicitly trained such that a small number of gradient steps in a new task from the distribution will produce fast learning on that task (Finn *et al.*, 2017).

There exists different denominations for this setting with a distribution of environments. For instance, the denomination of "multi-task setting" is usually used in settings where a short history of observations is sufficient to clearly distinguish the tasks. As an example of a multi-task setting, a deep RL agent can exceed median human performance on the set of 57 Atari games with a single set of weights (Hessel *et al.*, 2018). Other related denominations are the concepts of "contextual" policies (Da Silva *et al.*, 2012) and "universal"/"goal conditioned" value functions (Schaul *et al.*, 2015a) that refer to learning policies or value function within the same dynamics for multiple goals (multiple reward functions).

10.2 Transfer learning

Transfer learning is the task of efficiently using previous knowledge from a source environment to achieve a good performance in a target environment. In a transfer learning setting, the target environment should not be in the distribution of the source tasks. However, in practice, the concept of transfer learning is sometimes closely related to meta learning, as we discuss hereafter.

10.2.1 Zero-shot learning

The idea of zero-shot learning is that an agent should be able to act appropriately in a new task directly from experience acquired on other similar tasks. For instance, one use case is to learn a policy in a simulation environment and then use it in a real-world context where gathering experience is not possible or severely constrained (see §11.2). To achieve this, the agent must either (i) develop generalization capacities described in Chapter 7 or (ii) use specific transfer strategies that explicitly retrain or replace some of its components to adjust to new tasks.

To develop generalization capacities, one approach is to use an idea similar to data augmentation in supervised learning so as to make sense of variations that were not encountered in the training data. Exactly as in the meta-learning setting (§10.1.2), the actual (unseen) task may

appear to the agent as just another variation if there is enough data augmentation on the training data. For instance, the agent can be trained with deep RL techniques on different tasks simultaneously, and it is shown by Parisotto *et al.*, 2015 that it can generalize to new related domains where the exact state representation has never been observed. Similarly, the agent can be trained in a simulated environment while being provided with different renderings of the observations. In that case, the learned policy can transfer well to real images (Sadeghi and Levine, 2016; Tobin *et al.*, 2017). The underlying reason for these successes is the ability of the deep learning architecture to generalize between states that have similar high-level representations and should therefore have the same value function/policy in different domains. Rather than manually tuning the randomization of simulations, one can also adapt the simulation parameters by matching the policy behavior in simulation to the real world (Chebotar *et al.*, 2018). Another approach to zero-shot transfer is to use algorithms that enforce states that relate to the same underlying task but have different renderings to be mapped into an abstract state that is close (Tzeng *et al.*, 2015; François-Lavet *et al.*, 2018).

10.2.2 Lifelong learning or continual learning

A specific way of achieving transfer learning is to aim at *lifelong learning* or *continual learning*. According to Silver *et al.*, 2013, lifelong machine learning relates to the capability of a system to learn many tasks over a lifetime from one or more domains.

In general, deep learning architectures can generalize knowledge across multiple tasks by sharing network parameters. A direct approach is thus to train function approximators (e.g. policy, value function, model, etc.) sequentially in different environments. The difficulty of this approach is to find methods that enable the agent to retain knowledge in order to more efficiently learn new tasks. The problem of retaining knowledge in deep reinforcement learning is complicated by the phenomenon of catastrophic forgetting, where generalization to previously seen data is lost at later stages of learning.

The straightforward approach is to either (i) use experience replay from all previous experience (as discussed in §8.2), or (ii) retrain occasionally on previous tasks similar to the meta-learning setting (as discussed in §10.1.2).

When these two options are not available, or as a complement to the two previous approaches, one can use deep learning techniques that are robust to forgetting, such as progressive networks (Rusu *et al.*, 2016). The idea is to leverage prior knowledge by adding, for each new task, lateral connections to previously learned features (that are kept fixed). Other approaches to limiting catastrophic forgetting include slowing down learning on the weights important for previous tasks (Kirkpatrick *et al.*, 2016) and decomposing learning into skill hierarchies (Stone and Veloso, 2000; Tessler *et al.*, 2017).

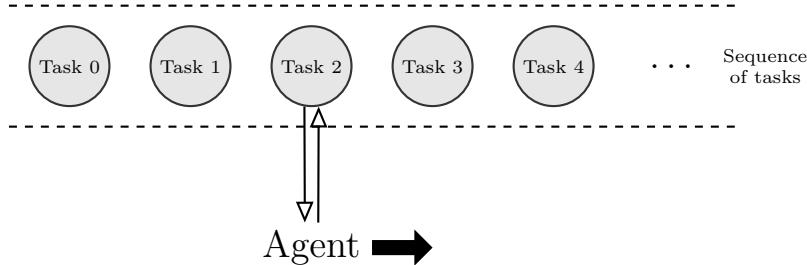


Figure 10.3: Illustration of the continual learning setting where an agent has to interact sequentially with related (but different) tasks.

10.2.3 Curriculum learning

A particular setting of continual learning is curriculum learning. Here, the goal is to explicitly design a sequence of source tasks for an agent to train on such that the final performance or learning speed is improved on a target task. The idea is to start by learning small and easy aspects of the target task and then to gradually increase the difficulty level (Bengio *et al.*, 2009; Narvekar *et al.*, 2016). For instance, Florensa *et al.* (2018) use generative adversarial training to automatically generate goals for a contextual policy such that they are always at the appropriate level of difficulty. As the difficulty and number of tasks increase, one

possibility to satisfy the bias-overfitting tradeoff is to consider network transformations through learning.

10.3 Learning without explicit reward function

In reinforcement learning, the reward function defines the goals to be achieved by the agent (for a given environment and a given discount factor). Due to the complexity of environments in practical applications, defining a reward function can turn out to be rather complicated. There are two other possibilities: (i) given demonstrations of the desired task, we can use imitation learning or extract a reward function using inverse reinforcement learning; (ii) a human may provide feedback on the agent's behavior in order to define the task.

10.3.1 Learning from demonstrations

In some circumstances, the agent is only provided with trajectories of an expert agent (also called the teacher), without rewards. Given an observed behavior, the goal is to have the agent perform similarly. Two approaches are possible:

- Imitation learning uses supervised learning to map states to actions from the observations of the expert's behavior (e.g., Giusti *et al.*, 2016). Among other applications, this approach has been used for self-driving cars to map raw pixels directly to steering commands thanks to a deep neural network (Bojarski *et al.*, 2016).
- Inverse reinforcement learning (IRL) determines a possible reward function given observations of optimal behavior. When the system dynamics is known (except the reward function), this is an appealing approach particularly when the reward function provides the most generalizable definition of the task (Ng, Russell, *et al.*, 2000; Abbeel and Ng, 2004). For example, let us consider a large MDP for which the expert always ends up transitioning to the same state. In that context, one may be able to easily infer, from only a few trajectories, what the probable goal of the task is (a reward function that explains the behavior of the teacher),

as opposed to directly learning the policy via imitation learning, which is much less efficient. Note that recent works bypass the requirement of the knowledge of the system dynamics (Boularias *et al.*, 2011; Kalakrishnan *et al.*, 2013; Finn *et al.*, 2016b) by using techniques based on the principle of maximum causal entropy (Ziebart, 2010).

A combination of the two approaches has also been investigated by Neu and Szepesvári, 2012; Ho and Ermon, 2016. In particular, Ho and Ermon, 2016 use adversarial methods to learn a discriminator (i.e., a reward function) such that the policy matches the distribution of demonstrative samples.

It is important to note that in many real-world applications, the teacher is not exactly in the same context as the agent. Thus, transfer learning may also be of crucial importance (Schulman *et al.*, 2016; Liu *et al.*, 2017).

Another setting requires the agent to learn directly from a sequence of observations without corresponding actions (and possibly in a slightly different context). This may be done in a meta-learning setting by providing positive reward to the agent when it performs as it is expected based on the demonstration of the teacher. The agent can then act based on new unseen trajectories of the teacher, with the objective that it can generalize sufficiently well to perform new tasks (Paine *et al.*, 2018).

10.3.2 Learning from direct feedback

Learning from feedback investigates how an agent can interactively learn behaviors from a human teacher who provides positive and negative feedback signals. In order to learn complex behavior, human trainer feedbacks has the potential to be more performant than a reward function defined a priori (MacGlashan *et al.*, 2017; Warnell *et al.*, 2017). This setting can be related to the idea of curriculum learning discussed in §10.2.3.

In the work of Hadfield-Menell *et al.*, 2016, the cooperative inverse reinforcement learning framework considers a two-player game between a human and a robot interacting with an environment with the purpose

of maximizing the human's reward function. In the work of Christiano *et al.*, 2017, it is shown how learning a separate reward model using supervised learning lets us significantly reduce the amount of feedback required from a human teacher. They also present the first practical applications of using human feedback in the context of deep RL to solve tasks with a high dimensional observation space.

10.4 Multi-agent systems

A multi-agent system is composed of multiple interacting agents within an environment (Littman, 1994).

Definition 10.2. A multi-agent POMDP with N agents is a tuple $(\mathcal{S}, \mathcal{A}_N, \dots, \mathcal{A}_N, T, R_1, \dots, R_N, \Omega, O_1, \dots, O_N, \gamma)$ where:

- \mathcal{S} is a finite set of states $\{1, \dots, N_{\mathcal{S}}\}$ (describing the possible configurations of all agents),
- $\mathcal{A} = \mathcal{A}_1 \times \dots \times \mathcal{A}_n$ is a finite set of actions $\{1, \dots, N_{\mathcal{A}}\}$,
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition function (set of conditional transition probabilities between states),
- $\forall i, R_i : \mathcal{S} \times \mathcal{A}_i \times \mathcal{S} \rightarrow \mathcal{R}$ is the reward function for agent i , where \mathcal{R} is a continuous set of possible rewards in a range $R_{\max} \in \mathbb{R}^+$ (e.g., $[0, R_{\max}]$ without loss of generality),
- Ω is a finite set of observations $\{1, \dots, N_{\Omega}\}$,
- $\forall i, O_i : \mathcal{S} \times \Omega \rightarrow [0, 1]$ is a set of conditional observation probabilities, and
- $\gamma \in [0, 1]$ is the discount factor.

For this type of system, many different settings can be considered.

- Collaborative versus non-collaborative setting. In a pure collaborative setting, agents have a shared reward measurement ($R_i = R_j, \forall i, j \in [1, \dots, N]$). In a mixed or non-collaborative (possibly adversarial) setting each agent obtains different rewards. In both

cases, each agent i aims to maximize a discounted sum of its rewards $\sum_{t=0}^H \gamma^t r_t^{(i)}$.

- Decentralized versus centralized setting. In a decentralized setting, each agent selects its own action conditioned only on its local information. When collaboration is beneficial, this decentralized setting can lead to the emergence of communication between agents in order to share information (e.g., Sukhbaatar *et al.*, 2016).

In a centralized setting, the RL algorithm has access to all observations $w^{(i)}$ and all rewards $r^{(i)}$. The problem can be reduced to a single-agent RL problem on the condition that a single objective can be defined (in a purely collaborative setting, the unique objective is straightforward). Note that even when a centralized approach can be considered (depending on the problem), an architecture that does not make use of the multi-agent structure usually leads to sub-optimal learning (e.g., Sunehag *et al.*, 2017).

In general, multi-agent systems are challenging because agents are independently updating their policies as learning progresses, and therefore the environment appears non-stationary to any particular agent. For training one particular agent, one approach is to select randomly the policies of all other agents from a pool of previously learned policies. This can stabilize training of the agent that is learning and prevent overfitting to the current policy of the other agents (Silver *et al.*, 2016b).

In addition, from the perspective of a given agent, the environment is usually strongly stochastic even with a known, fixed policy for all other agents. Indeed, any given agent does not know how the other agents will act and consequently, it doesn't know how its own actions contribute to the rewards it obtains. This can be partly explained due to partial observability, and partly due to the intrinsic stochasticity of the policies followed by other agents (e.g., when there is a high level of exploration). For these reasons, a high variance of the expected global return is observed, which makes learning challenging (particularly when used in conjunction with bootstrapping). In the context of the collaborative

setting, a common approach is to use an actor-critic architecture with a centralized critic during learning and decentralized actor (the agents can be deployed independently). These topics have already been investigated in works by Foerster *et al.*, 2017a; Sunehag *et al.*, 2017; Lowe *et al.*, 2017 as well as in the related work discussed in these papers. Other works have shown how it is possible to take into account a term that either considers the learning of the other agent (Foerster *et al.*, 2018) or an internal model that predicts the actions of other agents (Jaques *et al.*, 2018).

Deep RL agents are able to achieve human-level performance in 3D multi-player first-person video games such as Quake III Arena Capture the Flag (Jaderberg *et al.*, 2018). Thus, techniques from deep RL have a large potential for many real-world tasks that require multiple agents to cooperate in domains such as robotics, self-driving cars, etc.

11

Perspectives on deep reinforcement learning

In this section, we first mention some of the main successes of deep RL. Then, we describe some of the main challenges for tackling an even wider range of real-world problems. Finally, we discuss some parallels that can be found between deep RL and neuroscience.

11.1 Successes of deep reinforcement learning

Deep RL techniques have demonstrated their ability to tackle a wide range of problems that were previously unsolved. Some of the most renowned achievements are

- beating previous computer programs in the game of backgammon (Tesauro, 1995),
- attaining superhuman-level performance in playing Atari games from the pixels (Mnih *et al.*, 2015),
- mastering the game of Go (Silver *et al.*, 2016b), as well as
- beating professional poker players in the game of heads up no-limit Texas hold’em: Libratus (Brown and Sandholm, 2017) and Deepstack (Moravčík *et al.*, 2017).

These achievements in popular games are important because they show the potential of deep RL in a variety of complex and diverse tasks that require working with high-dimensional inputs. Deep RL has also shown lots of potential for real-world applications such as robotics (Kalashnikov *et al.*, 2018), self-driving cars (You *et al.*, 2017), finance (Deng *et al.*, 2017), smart grids (François-Lavet *et al.*, 2016), dialogue systems (Fazel-Zarandi *et al.*, 2017), etc. In fact, Deep RL systems are already in production environments. For example, Gauci *et al.* (2018) describe how Facebook uses Deep RL such as for pushing notifications and for faster video loading with smart prefetching.

RL is also applicable to fields where one could think that supervised learning alone is sufficient, such as sequence prediction (Ranzato *et al.*, 2015; Bahdanau *et al.*, 2016). Designing the right neural architecture for supervised learning tasks has also been cast as an RL problem (Zoph and Le, 2016). Note that those types of tasks can also be tackled with evolutionary strategies (Miikkulainen *et al.*, 2017; Real *et al.*, 2017).

Finally, it should be mentioned that deep RL has applications in classic and fundamental algorithmic problems in the field of computer science, such as the travelling salesman problem (Bello *et al.*, 2016). This is an NP-complete problem and the possibility to tackle it with deep RL shows the potential impact that it could have on several other NP-complete problems, on the condition that the structure of these problems can be exploited.

11.2 Challenges of applying reinforcement learning to real-world problems

The algorithms discussed in this introduction to deep RL can, in principle, be used to solve many different types of real-world problems. In practice, even in the case where the task is well defined (explicit reward function), there is one fundamental difficulty: it is often not possible to let an agent interact freely and sufficiently in the actual environment (or set of environments), due to either safety, cost or time constraints. We can distinguish two main cases in real-world applications:

1. The agent may not be able to interact with the true environment but only with an inaccurate simulation of it. This scenario occurs

for instance in robotics (Zhu *et al.*, 2016; Gu *et al.*, 2017a). When first learning in a simulation, the difference with the real-world domain is known as the *reality gap* (see e.g. Jakobi *et al.*, 1995).

2. The acquisition of new observations may not be possible anymore (e.g., the batch setting). This scenario happens for instance in medical trials, in tasks with dependence on weather conditions or in trading markets (e.g., energy markets and stock markets).

Note that a combination of the two scenarios is also possible in the case where the dynamics of the environment may be simulated but where there is a dependence on an exogenous time series that is only accessible via limited data (François-Lavet *et al.*, 2016).

In order to deal with these limitations, different elements are important:

- One can aim to develop a simulator that is as accurate as possible.
- One can design the learning algorithm so as to improve generalization and/or use transfer learning methods (see Chapter 7).

11.3 Relations between deep RL and neuroscience

One of the interesting aspects of deep RL is its relations to neuroscience. During the development of algorithms able to solve challenging sequential decision-making tasks, biological plausibility was not a requirement from an engineering standpoint. However, biological intelligence has been a key inspiration for many of the most successful algorithms. Indeed, even the ideas of reinforcement and deep learning have strong links with neuroscience and biological intelligence.

Reinforcement In general, RL has had a rich conceptual relationship to neuroscience. RL has used neuroscience as an inspiration and it has also been a tool to explain neuroscience phenomena (Niv, 2009). RL models have also been used as a tool in the related field of neuroeconomics (Camerer *et al.*, 2005), which uses models of human decision-making to inform economic analyses.

The idea of reinforcement (or at least the term) can be traced back to the work of Pavlov (1927) in the context of animal behavior. In the Pavlovian conditioning model, reinforcement is described as the strengthening/weakening effect of a behavior whenever that behavior is preceded by a specific stimulus. The Pavlovian conditioning model led to the development of the Rescorla-Wagner Theory (Rescorla, Wagner, *et al.*, 1972), which assumed that learning is driven by the error between predicted and received reward, among other prediction models. In computational RL, those concepts have been at the heart of many different algorithms, such as in the development of temporal-difference (TD) methods (Sutton, 1984; Schultz *et al.*, 1997; Russek *et al.*, 2017). These connections were further strengthened when it was found that the dopamine neurons in the brain act in a similar manner to TD-like updates to direct learning in the brain (Schultz *et al.*, 1997).

Driven by such connections, many aspects of reinforcement learning have also been investigated directly to explain certain phenomena in the brain. For instance, computational models have been an inspiration to explain cognitive phenomena such as exploration (Cohen *et al.*, 2007) and temporal discounting of rewards (Story *et al.*, 2014). In cognitive science, Kahneman (2011) has also described that there is a dichotomy between two modes of thoughts: a "System 1" that is fast and instinctive and a "System 2" that is slower and more logical. In deep reinforcement, a similar dichotomy can be observed when we consider the model-free and the model-based approaches. As another example, the idea of having a meaningful abstract representation in deep RL can also be related to how animals (including humans) think. Indeed, a conscious thought at a particular time instant can be seen as a low-dimensional combination of a few concepts in order to take decisions (Bengio, 2017).

There is a dense and rich literature about the connections between RL and neuroscience and, as such, the reader is referred to the work of Sutton and Barto (2017), Niv (2009), Lee *et al.* (2012), Holroyd and Coles (2002), Dayan and Niv (2008), Dayan and Daw (2008), Montague (2013), and Niv and Montague (2009) for an in-depth history of the development of reinforcement learning and its relations to neuroscience.

Deep learning Deep learning also finds its origin in models of neural processing in the brain of biological entities. However, subsequent developments are such that deep learning has become partly incompatible with current knowledge of neurobiology (Bengio *et al.*, 2015). There exists nonetheless many parallels. One such example is the convolutional structure used in deep learning that is inspired by the organization of the animal visual cortex (Fukushima and Miyake, 1982; LeCun *et al.*, 1998).

Much work is still needed to bridge the gap between machine learning and general intelligence of humans (or even animals). Looking back at all the achievements obtained by taking inspiration from neuroscience, it is natural to believe that further understanding of biological brains could play a vital role in building more powerful algorithms and conversely. In particular, we refer the reader to the survey by Hassabis *et al.*, 2017 where the bidirectional influence between deep RL and neuroscience is discussed.

12

Conclusion

Sequential decision-making remains an active field of research with many theoretical, methodological and experimental challenges still open. The important developments in the field of deep learning have contributed to many new avenues where RL methods and deep learning are combined. In particular, deep learning has brought important generalization capabilities, which opens new possibilities to work with large, high-dimensional state and/or action spaces. There is every reason to think that this development will continue in the coming years with more efficient algorithms and many new applications.

12.1 Future development of deep RL

In deep RL, we have emphasized in this manuscript that one of the central questions is the concept of generalization. To this end, the new developments in the field of deep RL will surely develop the current trends of taking explicit algorithms and making them differentiable so that they can be embedded in a specific form of neural network and can be trained end-to-end. This can bring algorithms with richer and smarter structures that would be better suited for reasoning on a more abstract level, which would allow to tackle an even wider range of

applications than they currently do today. Smart architectures could also be used for hierarchical learning where much progress is still needed in the domain of temporal abstraction.

We also expect to see deep RL algorithms going in the direction of meta-learning and lifelong learning where previous knowledge (e.g., in the form of pre-trained networks) can be embedded so as to increase performance and training time. Another key challenge is to improve current transfer learning abilities between simulations and real-world cases. This would allow learning complex decision-making problems in simulations (with the possibility to gather samples in a flexible way), and then use the learned skills in real-world environments, with applications in robotics, self-driving cars, etc.

Finally, we expect deep RL techniques to develop improved curiosity driven abilities to be able to better discover by themselves their environment.

12.2 Applications and societal impact of deep RL and artificial intelligence in general

In terms of applications, many areas are likely to be impacted by the possibilities brought by deep RL. It is always difficult to predict the timelines for the different developments, but the current interest in deep RL could be the beginning of profound transformations in information and communication technologies, with applications in clinical decision support, marketing, finance, resource management, autonomous driving, robotics, smart grids, and more.

Current developments in artificial intelligence (both for deep RL or in general for machine learning) follows the development of many tools brought by information and communications technologies. As with all new technologies, this comes with different potential opportunities and challenges for our society.

On the positive side, algorithms based on (deep) reinforcement learning promise great value to people and society. They have the potential to enhance the quality of life by automating tedious and exhausting tasks with robots (Levine *et al.*, 2016; Gandhi *et al.*, 2017; Pinto *et al.*, 2017). They may improve education by providing adaptive

content and keeping students engaged (Mandel *et al.*, 2014). They can improve public health with, for instance, intelligent clinical decision-making (Fonteneau *et al.*, 2008; Bennett and Hauser, 2013). They may provide robust solutions to some of the self-driving cars challenges (Bojarski *et al.*, 2016; You *et al.*, 2017). They also have the possibility to help managing ecological resources (Dietterich, 2009) or reducing greenhouse gas emissions by, e.g., optimizing traffic (Li *et al.*, 2016). They have applications in computer graphics, such as for character animation (Peng *et al.*, 2017b). They also have applications in finance (Deng *et al.*, 2017), smart grids (François-Lavet, 2017), etc.

However, we need to be careful that deep RL algorithms are safe, reliable and predictable (Amodei *et al.*, 2016; Bostrom, 2017). As a simple example, to capture what we want an agent to do in deep RL, we frequently end up, in practice, designing the reward function, somewhat arbitrarily. Often this works well, but sometimes it produces unexpected, and potentially catastrophic behaviors. For instance, to remove a certain invasive species from an environment, one may design an agent that obtains a reward every time it removes one of these organisms. However, it is likely that to obtain the maximum cumulative rewards, the agent will learn to let that invasive species develop and only then would eliminate many of the invasive organisms, which is of course not the intended behavior. All aspects related to safe exploration are also potential concerns in the hypothesis that deep RL algorithms are deployed in real-life settings.

In addition, as with all powerful tools, deep RL algorithms also bring societal and ethical challenges (Brundage *et al.*, 2018), raising the question of how they can be used for the benefit of all. Even though different interpretations can come into play when one discusses human sciences, we mention in this conclusion some of the potential issues that may need further investigation.

The ethical use of artificial intelligence is a broad concern. The specificity of RL as compared to supervised learning techniques is that it can naturally deal with sequences of interactions, which is ideal for chatbots, smart assistants, etc. As it is the case with most technologies, regulation should, at some point, ensure a positive impact of its usage.

In addition, machine learning and deep RL algorithms will likely yield to further automation and robotisation than it is currently possible. This is clearly a concern in the context of autonomous weapons, for instance (Walsh, 2017). Automation also influences the economy, the job market and our society as a whole. A key challenge for humanity is to make sure that the future technological developments in artificial intelligence do not create an ecological crisis (Harari, 2014) or deepen the inequalities in our society with potential social and economic instabilities (Piketty, 2013).

We are still at the very first steps of deep RL and artificial intelligence in general. The future is hard to predict; however, it is key that the potential issues related to the use of these algorithms are progressively taken into consideration in public policies. If that is the case, these new algorithms can have a positive impact on our society.

Appendices

A

Appendix

A.1 Deep RL frameworks

Here is a list of some well-known frameworks used for deep RL:

- DeeR (François-Lavet *et al.*, 2016) is focused on being (i) easily accessible and (ii) modular for researchers.
- Dopamine (Bellemare *et al.*, 2018) provides standard algorithms along with baselines for the ATARI games.
- ELF (Tian *et al.*, 2017) is a research platform for deep RL, aimed mainly to real-time strategy games.
- OpenAI baselines (Dhariwal *et al.*, 2017) is a set of popular deep RL algorithms, including DDPG, TRPO, PPO, ACKTR. The focus of this framework is to provide implementations of baselines.
- PyBrain (Schaal *et al.*, 2010) is a machine learning library with some RL support.
- rllab (Duan *et al.*, 2016a) provides a set of benchmarked implementations of deep RL algorithms.

Framework	Deep RL	Python interface	Automatic GPU support
DeeR	yes	yes	yes
Dopamine	yes	yes	yes
ELF	yes	no	yes
OpenAI baselines	yes	yes	yes
PyBrain	yes	yes	no
RL-Glue	no	yes	no
RLPy	no	yes	no
rllab	yes	yes	yes
TensorForce	yes	yes	yes

Table A.1: Summary of some characteristics for a few existing RL frameworks.

- TensorForce (Schaarschmidt *et al.*, 2017) is a framework for deep RL built around Tensorflow with several algorithm implementations. It aims at moving reinforcement computations into the Tensorflow graph for performance gains and efficiency. As such, it is heavily tied to the Tensorflow deep learning library. It provides many algorithm implementations including TRPO, DQN, PPO, and A3C.

Even though, they are not tailored specifically for deep RL, we can also cite the two following frameworks for reinforcement learning:

- RL-Glue (Tanner and White, 2009) provides a standard interface that allows to connect RL agents, environments, and experiment programs together.
- RLPy (Geramifard *et al.*, 2015) is a framework focused on value-based RL using linear function approximators with discrete actions.

Table A.1 provides a summary of some properties of the aforementioned libraries.

References

- Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.* 2016. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”. *arXiv preprint arXiv:1603.04467*.
- Abbeel, P. and A. Y. Ng. 2004. “Apprenticeship learning via inverse reinforcement learning”. In: *Proceedings of the twenty-first international conference on Machine learning*. ACM. 1.
- Amari, S. 1998. “Natural Gradient Works Efficiently in Learning”. *Neural Computation*. 10(2): 251–276.
- Amodei, D., C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. 2016. “Concrete problems in AI safety”. *arXiv preprint arXiv:1606.06565*.
- Anderson, T. W., T. W. Anderson, T. W. Anderson, T. W. Anderson, and E.-U. Mathématicien. 1958. *An introduction to multivariate statistical analysis*. Vol. 2. Wiley New York.
- Aytar, Y., T. Pfaff, D. Budden, T. L. Paine, Z. Wang, and N. de Freitas. 2018. “Playing hard exploration games by watching YouTube”. *arXiv preprint arXiv:1805.11592*.
- Bacon, P.-L., J. Harb, and D. Precup. 2016. “The option-critic architecture”. *arXiv preprint arXiv:1609.05140*.

- Bahdanau, D., P. Brakel, K. Xu, A. Goyal, R. Lowe, J. Pineau, A. Courville, and Y. Bengio. 2016. “An actor-critic algorithm for sequence prediction”. *arXiv preprint arXiv:1607.07086*.
- Baird, L. 1995. “Residual algorithms: Reinforcement learning with function approximation”. In: *ICML*. 30–37.
- Baker, M. 2016. “1,500 scientists lift the lid on reproducibility”. *Nature News*. 533(7604): 452.
- Bartlett, P. L. and S. Mendelson. 2002. “Rademacher and Gaussian complexities: Risk bounds and structural results”. *Journal of Machine Learning Research*. 3(Nov): 463–482.
- Barto, A. G., R. S. Sutton, and C. W. Anderson. 1983. “Neuronlike adaptive elements that can solve difficult learning control problems”. *IEEE transactions on systems, man, and cybernetics*. (5): 834–846.
- Beattie, C., J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, et al. 2016. “DeepMind Lab”. *arXiv preprint arXiv:1612.03801*.
- Bellemare, M. G., P. S. Castro, C. Gelada, K. Saurabh, and S. Moitra. 2018. “Dopamine”. <https://github.com/google/dopamine>.
- Bellemare, M. G., W. Dabney, and R. Munos. 2017. “A distributional perspective on reinforcement learning”. *arXiv preprint arXiv:1707.06887*.
- Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling. 2013. “The Arcade Learning Environment: An evaluation platform for general agents.” *Journal of Artificial Intelligence Research*. 47: 253–279.
- Bellemare, M. G., S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. 2016. “Unifying Count-Based Exploration and Intrinsic Motivation”. *arXiv preprint arXiv:1606.01868*.
- Bellman, R. 1957a. “A Markovian decision process”. *Journal of Mathematics and Mechanics*: 679–684.
- Bellman, R. 1957b. “Dynamic Programming”.
- Bellman, R. E. and S. E. Dreyfus. 1962. “Applied dynamic programming”.
- Bello, I., H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. 2016. “Neural Combinatorial Optimization with Reinforcement Learning”. *arXiv preprint arXiv:1611.09940*.

- Bengio, Y. 2017. “The Consciousness Prior”. *arXiv preprint arXiv:1709.08568*.
- Bengio, Y., D.-H. Lee, J. Bornschein, T. Mesnard, and Z. Lin. 2015. “Towards biologically plausible deep learning”. *arXiv preprint arXiv:1502.04156*.
- Bengio, Y., J. Louradour, R. Collobert, and J. Weston. 2009. “Curriculum learning”. In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 41–48.
- Bennett, C. C. and K. Hauser. 2013. “Artificial intelligence framework for simulating clinical decision-making: A Markov decision process approach”. *Artificial intelligence in medicine*. 57(1): 9–19.
- Bertsekas, D. P., D. P. Bertsekas, D. P. Bertsekas, and D. P. Bertsekas. 1995. *Dynamic programming and optimal control*. Vol. 1. No. 2. Athena scientific Belmont, MA.
- Bojarski, M., D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. 2016. “End to end learning for self-driving cars”. *arXiv preprint arXiv:1604.07316*.
- Bostrom, N. 2017. *Superintelligence*. Dunod.
- Bouckaert, R. R. 2003. “Choosing between two learning algorithms based on calibrated tests”. In: *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. 51–58.
- Bouckaert, R. R. and E. Frank. 2004. “Evaluating the replicability of significance tests for comparing learning algorithms”. In: *PAKDD*. Springer. 3–12.
- Boularias, A., J. Kober, and J. Peters. 2011. “Relative Entropy Inverse Reinforcement Learning.” In: *AISTATS*. 182–189.
- Boyan, J. A. and A. W. Moore. 1995. “Generalization in reinforcement learning: Safely approximating the value function”. In: *Advances in neural information processing systems*. 369–376.
- Brafman, R. I. and M. Tennenholtz. 2003. “R-max-a general polynomial time algorithm for near-optimal reinforcement learning”. *The Journal of Machine Learning Research*. 3: 213–231.

- Branavan, S., N. Kushman, T. Lei, and R. Barzilay. 2012. “Learning high-level planning from text”. In: *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*. Association for Computational Linguistics. 126–135.
- Braziunas, D. 2003. “POMDP solution methods”. *University of Toronto, Tech. Rep.*
- Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. 2016. “OpenAI Gym”.
- Brown, N. and T. Sandholm. 2017. “Libratus: The Superhuman AI for No-Limit Poker”. *International Joint Conference on Artificial Intelligence (IJCAI-17)*.
- Browne, C. B., E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. “A survey of monte carlo tree search methods”. *IEEE Transactions on Computational Intelligence and AI in games*. 4(1): 1–43.
- Brügmann, B. 1993. “Monte carlo go”. *Tech. rep.* Citeseer.
- Brundage, M., S. Avin, J. Clark, H. Toner, P. Eckersley, B. Garfinkel, A. Dafoe, P. Scharre, T. Zeitzoff, B. Filar, et al. 2018. “The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation”. *arXiv preprint arXiv:1802.07228*.
- Brys, T., A. Harutyunyan, P. Vrancx, M. E. Taylor, D. Kudenko, and A. Nowé. 2014. “Multi-objectivization of reinforcement learning problems by reward shaping”. In: *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE. 2315–2322.
- Bubeck, S., R. Munos, and G. Stoltz. 2011. “Pure exploration in finitely-armed and continuous-armed bandits”. *Theoretical Computer Science*. 412(19): 1832–1852.
- Burda, Y., H. Edwards, A. Storkey, and O. Klimov. 2018. “Exploration by Random Network Distillation”. *arXiv preprint arXiv:1810.12894*.
- Camerer, C., G. Loewenstein, and D. Prelec. 2005. “Neuroeconomics: How neuroscience can inform economics”. *Journal of economic Literature*. 43(1): 9–64.
- Campbell, M., A. J. Hoane, and F.-h. Hsu. 2002. “Deep blue”. *Artificial intelligence*. 134(1-2): 57–83.

- Casadevall, A. and F. C. Fang. 2010. “Reproducible science”.
- Castronovo, M., V. François-Lavet, R. Fonteneau, D. Ernst, and A. Couëtoux. 2017. “Approximate Bayes Optimal Policy Search using Neural Networks”. In: *9th International Conference on Agents and Artificial Intelligence (ICAART 2017)*.
- Chebotar, Y., A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox. 2018. “Closing the Sim-to-Real Loop: Adapting Simulation Randomization with Real World Experience”. *arXiv preprint arXiv:1810.05687*.
- Chen, T., I. Goodfellow, and J. Shlens. 2015. “Net2net: Accelerating learning via knowledge transfer”. *arXiv preprint arXiv:1511.05641*.
- Chen, X., C. Liu, and D. Song. 2017. “Learning Neural Programs To Parse Programs”. *arXiv preprint arXiv:1706.01284*.
- Chiappa, S., S. Racaniere, D. Wierstra, and S. Mohamed. 2017. “Recurrent Environment Simulators”. *arXiv preprint arXiv:1704.02254*.
- Christiano, P., J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei. 2017. “Deep reinforcement learning from human preferences”. *arXiv preprint arXiv:1706.03741*.
- Christopher, M. B. 2006. *Pattern recognition and machine learning*. Springer.
- Cohen, J. D., S. M. McClure, and J. Y. Angela. 2007. “Should I stay or should I go? How the human brain manages the trade-off between exploitation and exploration”. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*. 362(1481): 933–942.
- Cortes, C. and V. Vapnik. 1995. “Support-vector networks”. *Machine learning*. 20(3): 273–297.
- Coumans, E., Y. Bai, *et al.* 2016. “Bullet”. <http://pybullet.org/>.
- Da Silva, B., G. Konidaris, and A. Barto. 2012. “Learning parameterized skills”. *arXiv preprint arXiv:1206.6398*.
- Dabney, W., M. Rowland, M. G. Bellemare, and R. Munos. 2017. “Distributional Reinforcement Learning with Quantile Regression”. *arXiv preprint arXiv:1710.10044*.
- Dayan, P. and N. D. Daw. 2008. “Decision theory, reinforcement learning, and the brain”. *Cognitive, Affective, & Behavioral Neuroscience*. 8(4): 429–453.

- Dayan, P. and Y. Niv. 2008. “Reinforcement learning: the good, the bad and the ugly”. *Current opinion in neurobiology*. 18(2): 185–196.
- Dearden, R., N. Friedman, and D. Andre. 1999. “Model based Bayesian exploration”. In: *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 150–159.
- Dearden, R., N. Friedman, and S. Russell. 1998. “Bayesian Q-learning”.
- Deisenroth, M. and C. E. Rasmussen. 2011. “PILCO: A model-based and data-efficient approach to policy search”. In: *Proceedings of the 28th International Conference on machine learning (ICML-11)*. 465–472.
- Demšar, J. 2006. “Statistical comparisons of classifiers over multiple data sets”. *Journal of Machine learning research*. 7(Jan): 1–30.
- Deng, Y., F. Bao, Y. Kong, Z. Ren, and Q. Dai. 2017. “Deep direct reinforcement learning for financial signal representation and trading”. *IEEE transactions on neural networks and learning systems*. 28(3): 653–664.
- Dhariwal, P., C. Hesse, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. 2017. “OpenAI Baselines”.
- Dietterich, T. G. 1998. “Approximate statistical tests for comparing supervised classification learning algorithms”. *Neural computation*. 10(7): 1895–1923.
- Dietterich, T. G. 2009. “Machine learning and ecosystem informatics: challenges and opportunities”. In: *Asian Conference on Machine Learning*. Springer. 1–5.
- Dinculescu, M. and D. Precup. 2010. “Approximate predictive representations of partially observable systems”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 895–902.
- Dosovitskiy, A. and V. Koltun. 2016. “Learning to act by predicting the future”. *arXiv preprint arXiv:1611.01779*.
- Duan, Y., M. Andrychowicz, B. Stadie, J. Ho, J. Schneider, I. Sutskever, P. Abbeel, and W. Zaremba. 2017. “One-Shot Imitation Learning”. *arXiv preprint arXiv:1703.07326*.
- Duan, Y., X. Chen, R. Houthooft, J. Schulman, and P. Abbeel. 2016a. “Benchmarking deep reinforcement learning for continuous control”. In: *International Conference on Machine Learning*. 1329–1338.

- Duan, Y., J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel. 2016b. “RL $\hat{2}$: Fast Reinforcement Learning via Slow Reinforcement Learning”. *arXiv preprint arXiv:1611.02779*.
- Duchesne, L., E. Karangelos, and L. Wehenkel. 2017. “Machine learning of real-time power systems reliability management response”. *PowerTech Manchester 2017 Proceedings*.
- Džeroski, S., L. De Raedt, and K. Driessens. 2001. “Relational reinforcement learning”. *Machine learning*. 43(1-2): 7–52.
- Erhan, D., Y. Bengio, A. Courville, and P. Vincent. 2009. “Visualizing higher-layer features of a deep network”. *University of Montreal*. 1341(3): 1.
- Ernst, D., P. Geurts, and L. Wehenkel. 2005. “Tree-based batch mode reinforcement learning”. In: *Journal of Machine Learning Research*. 503–556.
- Farquhar, G., T. Rocktäschel, M. Igl, and S. Whiteson. 2017. “TreeQN and ATreeC: Differentiable Tree Planning for Deep Reinforcement Learning”. *arXiv preprint arXiv:1710.11417*.
- Fazel-Zarandi, M., S.-W. Li, J. Cao, J. Casale, P. Henderson, D. Whitney, and A. Geramifard. 2017. “Learning Robust Dialog Policies in Noisy Environments”. *arXiv preprint arXiv:1712.04034*.
- Finn, C., P. Abbeel, and S. Levine. 2017. “Model-agnostic meta-learning for fast adaptation of deep networks”. *arXiv preprint arXiv:1703.03400*.
- Finn, C., I. Goodfellow, and S. Levine. 2016a. “Unsupervised learning for physical interaction through video prediction”. In: *Advances In Neural Information Processing Systems*. 64–72.
- Finn, C., S. Levine, and P. Abbeel. 2016b. “Guided cost learning: Deep inverse optimal control via policy optimization”. In: *Proceedings of the 33rd International Conference on Machine Learning*. Vol. 48.
- Florensa, C., Y. Duan, and P. Abbeel. 2017. “Stochastic neural networks for hierarchical reinforcement learning”. *arXiv preprint arXiv:1704.03012*.
- Florensa, C., D. Held, X. Geng, and P. Abbeel. 2018. “Automatic goal generation for reinforcement learning agents”. In: *International Conference on Machine Learning*. 1514–1523.

- Foerster, J., R. Y. Chen, M. Al-Shedivat, S. Whiteson, P. Abbeel, and I. Mordatch. 2018. “Learning with opponent-learning awareness”. In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 122–130.
- Foerster, J., G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson. 2017a. “Counterfactual Multi-Agent Policy Gradients”. *arXiv preprint arXiv:1705.08926*.
- Foerster, J., N. Nardelli, G. Farquhar, P. Torr, P. Kohli, S. Whiteson, *et al.* 2017b. “Stabilising experience replay for deep multi-agent reinforcement learning”. *arXiv preprint arXiv:1702.08887*.
- Fonteneau, R., S. A. Murphy, L. Wehenkel, and D. Ernst. 2013. “Batch mode reinforcement learning based on the synthesis of artificial trajectories”. *Annals of operations research*. 208(1): 383–416.
- Fonteneau, R., L. Wehenkel, and D. Ernst. 2008. “Variable selection for dynamic treatment regimes: a reinforcement learning approach”.
- Fortunato, M., M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, *et al.* 2017. “Noisy networks for exploration”. *arXiv preprint arXiv:1706.10295*.
- Fox, R., A. Pakman, and N. Tishby. 2015. “Taming the noise in reinforcement learning via soft updates”. *arXiv preprint arXiv:1512.08562*.
- François-Lavet, V. 2017. “Contributions to deep reinforcement learning and its applications in smartgrids”. *PhD thesis*. University of Liege, Belgium.
- François-Lavet, V. *et al.* 2016. “DeeR”. <https://deer.readthedocs.io/>.
- François-Lavet, V., Y. Bengio, D. Precup, and J. Pineau. 2018. “Combined Reinforcement Learning via Abstract Representations”. *arXiv preprint arXiv:1809.04506*.
- François-Lavet, V., D. Ernst, and F. Raphael. 2017. “On overfitting and asymptotic bias in batch reinforcement learning with partial observability”. *arXiv preprint arXiv:1709.07796*.
- François-Lavet, V., R. Fonteneau, and D. Ernst. 2015. “How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies”. *arXiv preprint arXiv:1512.02011*.

- François-Lavet, V., D. Taralla, D. Ernst, and R. Fonteneau. 2016. “Deep Reinforcement Learning Solutions for Energy Microgrids Management”. In: *European Workshop on Reinforcement Learning*.
- Fukushima, K. and S. Miyake. 1982. “Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition”. In: *Competition and cooperation in neural nets*. Springer. 267–285.
- Gal, Y. and Z. Ghahramani. 2016. “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”. In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19–24, 2016*. 1050–1059.
- Gandhi, D., L. Pinto, and A. Gupta. 2017. “Learning to Fly by Crashing”. *arXiv preprint arXiv:1704.05588*.
- Garnelo, M., K. Arulkumaran, and M. Shanahan. 2016. “Towards Deep Symbolic Reinforcement Learning”. *arXiv preprint arXiv:1609.05518*.
- Gauci, J., E. Conti, Y. Liang, K. Virochhsiri, Y. He, Z. Kaden, V. Narayanan, and X. Ye. 2018. “Horizon: Facebook’s Open Source Applied Reinforcement Learning Platform”. *arXiv preprint arXiv:1811.00260*.
- Gelly, S., Y. Wang, R. Munos, and O. Teytaud. 2006. “Modification of UCT with patterns in Monte-Carlo Go”.
- Geman, S., E. Bienenstock, and R. Doursat. 1992. “Neural networks and the bias/variance dilemma”. *Neural computation*. 4(1): 1–58.
- Geramifard, A., C. Dann, R. H. Klein, W. Dabney, and J. P. How. 2015. “RLPy: A Value-Function-Based Reinforcement Learning Framework for Education and Research”. *Journal of Machine Learning Research*. 16: 1573–1578.
- Geurts, P., D. Ernst, and L. Wehenkel. 2006. “Extremely randomized trees”. *Machine learning*. 63(1): 3–42.
- Ghavamzadeh, M., S. Mannor, J. Pineau, A. Tamar, et al. 2015. “Bayesian reinforcement learning: A survey”. *Foundations and Trends® in Machine Learning*. 8(5-6): 359–483.

- Giusti, A., J. Guzzi, D. C. Cireşan, F.-L. He, J. P. Rodriguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. Di Caro, *et al.* 2016. “A machine learning approach to visual perception of forest trails for mobile robots”. *IEEE Robotics and Automation Letters*. 1(2): 661–667.
- Goodfellow, I., Y. Bengio, and A. Courville. 2016. *Deep learning*. MIT Press.
- Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. 2014. “Generative adversarial nets”. In: *Advances in neural information processing systems*. 2672–2680.
- Gordon, G. J. 1996. “Stable fitted reinforcement learning”. In: *Advances in neural information processing systems*. 1052–1058.
- Gordon, G. J. 1999. “Approximate solutions to Markov decision processes”. *Robotics Institute*: 228.
- Graves, A., G. Wayne, and I. Danihelka. 2014. “Neural turing machines”. *arXiv preprint arXiv:1410.5401*.
- Gregor, K., D. J. Rezende, and D. Wierstra. 2016. “Variational Intrinsic Control”. *arXiv preprint arXiv:1611.07507*.
- Gruslys, A., M. G. Azar, M. G. Bellemare, and R. Munos. 2017. “The Reactor: A Sample-Efficient Actor-Critic Architecture”. *arXiv preprint arXiv:1704.04651*.
- Gu, S., E. Holly, T. Lillicrap, and S. Levine. 2017a. “Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates”. In: *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE. 3389–3396.
- Gu, S., T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine. 2017b. “Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic”. In: *5th International Conference on Learning Representations (ICLR 2017)*.
- Gu, S., T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine. 2016a. “Q-prop: Sample-efficient policy gradient with an off-policy critic”. *arXiv preprint arXiv:1611.02247*.

- Gu, S., T. Lillicrap, Z. Ghahramani, R. E. Turner, B. Schölkopf, and S. Levine. 2017c. “Interpolated Policy Gradient: Merging On-Policy and Off-Policy Gradient Estimation for Deep Reinforcement Learning”. *arXiv preprint arXiv:1706.00387*.
- Gu, S., T. Lillicrap, I. Sutskever, and S. Levine. 2016b. “Continuous Deep Q-Learning with Model-based Acceleration”. *arXiv preprint arXiv:1603.00748*.
- Guo, Z. D. and E. Brunskill. 2017. “Sample efficient feature selection for factored mdps”. *arXiv preprint arXiv:1703.03454*.
- Haarnoja, T., H. Tang, P. Abbeel, and S. Levine. 2017. “Reinforcement learning with deep energy-based policies”. *arXiv preprint arXiv:1702.08165*.
- Haber, N., D. Mrowca, L. Fei-Fei, and D. L. Yamins. 2018. “Learning to Play with Intrinsically-Motivated Self-Aware Agents”. *arXiv preprint arXiv:1802.07442*.
- Hadfield-Menell, D., S. J. Russell, P. Abbeel, and A. Dragan. 2016. “Cooperative inverse reinforcement learning”. In: *Advances in neural information processing systems*. 3909–3917.
- Hafner, R. and M. Riedmiller. 2011. “Reinforcement learning in feedback control”. *Machine learning*. 84(1-2): 137–169.
- Halsey, L. G., D. Curran-Everett, S. L. Vowler, and G. B. Drummond. 2015. “The fickle P value generates irreproducible results”. *Nature methods*. 12(3): 179–185.
- Harari, Y. N. 2014. *Sapiens: A brief history of humankind*.
- Harutyunyan, A., M. G. Bellemare, T. Stepleton, and R. Munos. 2016. “Q (\lambda) with Off-Policy Corrections”. In: *International Conference on Algorithmic Learning Theory*. Springer. 305–320.
- Hassabis, D., D. Kumaran, C. Summerfield, and M. Botvinick. 2017. “Neuroscience-inspired artificial intelligence”. *Neuron*. 95(2): 245–258.
- Hasselt, H. V. 2010. “Double Q-learning”. In: *Advances in Neural Information Processing Systems*. 2613–2621.
- Hausknecht, M. and P. Stone. 2015. “Deep recurrent Q-learning for partially observable MDPs”. *arXiv preprint arXiv:1507.06527*.

- Hauskrecht, M., N. Meuleau, L. P. Kaelbling, T. Dean, and C. Boutilier. 1998. “Hierarchical solution of Markov decision processes using macro-actions”. In: *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 220–229.
- He, K., X. Zhang, S. Ren, and J. Sun. 2016. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
- Heess, N., G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa. 2015. “Learning continuous control policies by stochastic value gradients”. In: *Advances in Neural Information Processing Systems*. 2944–2952.
- Henderson, P., W.-D. Chang, F. Shkurti, J. Hansen, D. Meger, and G. Dudek. 2017a. “Benchmark Environments for Multitask Learning in Continuous Domains”. *ICML Lifelong Learning: A Reinforcement Learning Approach Workshop*.
- Henderson, P., R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. 2017b. “Deep Reinforcement Learning that Matters”. *arXiv preprint arXiv:1709.06560*.
- Hessel, M., J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. 2017. “Rainbow: Combining Improvements in Deep Reinforcement Learning”. *arXiv preprint arXiv:1710.02298*.
- Hessel, M., H. Soyer, L. Espeholt, W. Czarnecki, S. Schmitt, and H. van Hasselt. 2018. “Multi-task Deep Reinforcement Learning with PopArt”. *arXiv preprint arXiv:1809.04474*.
- Higgins, I., A. Pal, A. A. Rusu, L. Matthey, C. P. Burgess, A. Pritzel, M. Botvinick, C. Blundell, and A. Lerchner. 2017. “Darla: Improving zero-shot transfer in reinforcement learning”. *arXiv preprint arXiv:1707.08475*.
- Ho, J. and S. Ermon. 2016. “Generative adversarial imitation learning”. In: *Advances in Neural Information Processing Systems*. 4565–4573.
- Hochreiter, S. and J. Schmidhuber. 1997. “Long short-term memory”. *Neural computation*. 9(8): 1735–1780.
- Hochreiter, S., A. S. Younger, and P. R. Conwell. 2001. “Learning to learn using gradient descent”. In: *International Conference on Artificial Neural Networks*. Springer. 87–94.

- Holroyd, C. B. and M. G. Coles. 2002. “The neural basis of human error processing: reinforcement learning, dopamine, and the error-related negativity.” *Psychological review*. 109(4): 679.
- Houthooft, R., X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel. 2016. “Vime: Variational information maximizing exploration”. In: *Advances in Neural Information Processing Systems*. 1109–1117.
- Ioffe, S. and C. Szegedy. 2015. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. *arXiv preprint arXiv:1502.03167*.
- Islam, R., P. Henderson, M. Gomrokchi, and D. Precup. 2017. “Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control”. *ICML Reproducibility in Machine Learning Workshop*.
- Jaderberg, M., W. M. Czarnecki, I. Dunning, L. Marrs, G. Lever, A. G. Castaneda, C. Beattie, N. C. Rabinowitz, A. S. Morcos, A. Ruderman, *et al.* 2018. “Human-level performance in first-person multiplayer games with population-based deep reinforcement learning”. *arXiv preprint arXiv:1807.01281*.
- Jaderberg, M., V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu. 2016. “Reinforcement learning with unsupervised auxiliary tasks”. *arXiv preprint arXiv:1611.05397*.
- Jakobi, N., P. Husbands, and I. Harvey. 1995. “Noise and the reality gap: The use of simulation in evolutionary robotics”. In: *European Conference on Artificial Life*. Springer. 704–720.
- James, G. M. 2003. “Variance and bias for general loss functions”. *Machine Learning*. 51(2): 115–135.
- Jaques, N., A. Lazaridou, E. Hughes, C. Gulcehre, P. A. Ortega, D. Strouse, J. Z. Leibo, and N. de Freitas. 2018. “Intrinsic Social Motivation via Causal Influence in Multi-Agent RL”. *arXiv preprint arXiv:1810.08647*.
- Jaquette, S. C. *et al.* 1973. “Markov decision processes with a new optimality criterion: Discrete time”. *The Annals of Statistics*. 1(3): 496–505.

- Jiang, N., A. Kulesza, and S. Singh. 2015a. “Abstraction selection in model-based reinforcement learning”. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 179–188.
- Jiang, N., A. Kulesza, S. Singh, and R. Lewis. 2015b. “The Dependence of Effective Planning Horizon on Model Accuracy”. In: *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 1181–1189.
- Jiang, N. and L. Li. 2016. “Doubly robust off-policy value evaluation for reinforcement learning”. In: *Proceedings of The 33rd International Conference on Machine Learning*. 652–661.
- Johnson, J., B. Hariharan, L. van der Maaten, J. Hoffman, L. Fei-Fei, C. L. Zitnick, and R. Girshick. 2017. “Inferring and Executing Programs for Visual Reasoning”. *arXiv preprint arXiv:1705.03633*.
- Johnson, M., K. Hofmann, T. Hutton, and D. Bignell. 2016. “The Malmo Platform for Artificial Intelligence Experimentation.” In: *IJCAI*. 4246–4247.
- Juliani, A., V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange. 2018. “Unity: A General Platform for Intelligent Agents”. *arXiv preprint arXiv:1809.02627*.
- Kaelbling, L. P., M. L. Littman, and A. R. Cassandra. 1998. “Planning and acting in partially observable stochastic domains”. *Artificial intelligence*. 101(1): 99–134.
- Kahneman, D. 2011. *Thinking, fast and slow*. Macmillan.
- Kakade, S. 2001. “A Natural Policy Gradient”. In: *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*. 1531–1538.
- Kakade, S., M. Kearns, and J. Langford. 2003. “Exploration in metric state spaces”. In: *ICML*. Vol. 3. 306–312.
- Kalakrishnan, M., P. Pastor, L. Righetti, and S. Schaal. 2013. “Learning objective functions for manipulation”. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 1331–1336.

- Kalashnikov, D., A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, and S. Levine. 2018. “Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation”. *arXiv preprint arXiv:1806.10293*.
- Kalchbrenner, N., A. v. d. Oord, K. Simonyan, I. Danihelka, O. Vinyals, A. Graves, and K. Kavukcuoglu. 2016. “Video pixel networks”. *arXiv preprint arXiv:1610.00527*.
- Kansky, K., T. Silver, D. A. Mély, M. Eldawy, M. Lázaro-Gredilla, X. Lou, N. Dorfman, S. Sidor, S. Phoenix, and D. George. 2017. “Schema Networks: Zero-shot Transfer with a Generative Causal Model of Intuitive Physics”. *arXiv preprint arXiv:1706.04317*.
- Kaplan, R., C. Sauer, and A. Sosa. 2017. “Beating Atari with Natural Language Guided Reinforcement Learning”. *arXiv preprint arXiv:1704.05539*.
- Kearns, M. and S. Singh. 2002. “Near-optimal reinforcement learning in polynomial time”. *Machine Learning*. 49(2-3): 209–232.
- Kempka, M., M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski. 2016. “Vizdoom: A doom-based ai research platform for visual reinforcement learning”. In: *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*. IEEE. 1–8.
- Kirkpatrick, J., R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al. 2016. “Overcoming catastrophic forgetting in neural networks”. *arXiv preprint arXiv:1612.00796*.
- Klambauer, G., T. Unterthiner, A. Mayr, and S. Hochreiter. 2017. “Self-Normalizing Neural Networks”. *arXiv preprint arXiv:1706.02515*.
- Kolter, J. Z. and A. Y. Ng. 2009. “Near-Bayesian exploration in polynomial time”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM. 513–520.
- Konda, V. R. and J. N. Tsitsiklis. 2000. “Actor-critic algorithms”. In: *Advances in neural information processing systems*. 1008–1014.
- Krizhevsky, A., I. Sutskever, and G. E. Hinton. 2012. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 1097–1105.

- Kroon, M. and S. Whiteson. 2009. “Automatic feature selection for model-based reinforcement learning in factored MDPs”. In: *Machine Learning and Applications, 2009. ICMLA’09. International Conference on*. IEEE. 324–330.
- Kulkarni, T. D., K. Narasimhan, A. Saeedi, and J. Tenenbaum. 2016. “Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation”. In: *Advances in Neural Information Processing Systems*. 3675–3683.
- Lample, G. and D. S. Chaplot. 2017. “Playing FPS Games with Deep Reinforcement Learning.” In: *AAAI*. 2140–2146.
- LeCun, Y., Y. Bengio, et al. 1995. “Convolutional networks for images, speech, and time series”. *The handbook of brain theory and neural networks*. 3361(10): 1995.
- LeCun, Y., Y. Bengio, and G. Hinton. 2015. “Deep learning”. *Nature*. 521(7553): 436–444.
- LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. “Gradient-based learning applied to document recognition”. *Proceedings of the IEEE*. 86(11): 2278–2324.
- Lee, D., H. Seo, and M. W. Jung. 2012. “Neural basis of reinforcement learning and decision making”. *Annual review of neuroscience*. 35: 287–308.
- Leffler, B. R., M. L. Littman, and T. Edmunds. 2007. “Efficient reinforcement learning with relocatable action models”. In: *AAAI*. Vol. 7. 572–577.
- Levine, S., C. Finn, T. Darrell, and P. Abbeel. 2016. “End-to-end training of deep visuomotor policies”. *Journal of Machine Learning Research*. 17(39): 1–40.
- Levine, S. and V. Koltun. 2013. “Guided policy search”. In: *International Conference on Machine Learning*. 1–9.
- Li, L., Y. Lv, and F.-Y. Wang. 2016. “Traffic signal timing via deep reinforcement learning”. *IEEE/CAA Journal of Automatica Sinica*. 3(3): 247–254.
- Li, L., W. Chu, J. Langford, and X. Wang. 2011. “Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms”. In: *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM. 297–306.

- Li, X., L. Li, J. Gao, X. He, J. Chen, L. Deng, and J. He. 2015. “Recurrent reinforcement learning: a hybrid approach”. *arXiv preprint arXiv:1509.03044*.
- Liaw, A., M. Wiener, et al. 2002. “Classification and regression by randomForest”. *R news*. 2(3): 18–22.
- Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. 2015. “Continuous control with deep reinforcement learning”. *arXiv preprint arXiv:1509.02971*.
- Lin, L.-J. 1992. “Self-improving reactive agents based on reinforcement learning, planning and teaching”. *Machine learning*. 8(3-4): 293–321.
- Lipton, Z. C., J. Gao, L. Li, X. Li, F. Ahmed, and L. Deng. 2016. “Efficient exploration for dialogue policy learning with BBQ networks & replay buffer spiking”. *arXiv preprint arXiv:1608.05081*.
- Littman, M. L. 1994. “Markov games as a framework for multi-agent reinforcement learning”. In: *Proceedings of the eleventh international conference on machine learning*. Vol. 157. 157–163.
- Liu, Y., A. Gupta, P. Abbeel, and S. Levine. 2017. “Imitation from Observation: Learning to Imitate Behaviors from Raw Video via Context Translation”. *arXiv preprint arXiv:1707.03374*.
- Lowe, R., Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch. 2017. “Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments”. *arXiv preprint arXiv:1706.02275*.
- MacGlashan, J., M. K. Ho, R. Loftin, B. Peng, D. Roberts, M. E. Taylor, and M. L. Littman. 2017. “Interactive Learning from Policy-Dependent Human Feedback”. *arXiv preprint arXiv:1701.06049*.
- Machado, M. C., M. G. Bellemare, and M. Bowling. 2017a. “A Laplacian Framework for Option Discovery in Reinforcement Learning”. *arXiv preprint arXiv:1703.00956*.
- Machado, M. C., M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. 2017b. “Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents”. *arXiv preprint arXiv:1709.06009*.

- Mandel, T., Y.-E. Liu, S. Levine, E. Brunskill, and Z. Popovic. 2014. “Offline policy evaluation across representations with applications to educational games”. In: *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems. 1077–1084.
- Mankowitz, D. J., T. A. Mann, and S. Mannor. 2016. “Adaptive Skills Adaptive Partitions (ASAP)”. In: *Advances in Neural Information Processing Systems*. 1588–1596.
- Mathieu, M., C. Couprie, and Y. LeCun. 2015. “Deep multi-scale video prediction beyond mean square error”. *arXiv preprint arXiv:1511.05440*.
- Matiisen, T., A. Oliver, T. Cohen, and J. Schulman. 2017. “Teacher-Student Curriculum Learning”. *arXiv preprint arXiv:1707.00183*.
- McCallum, A. K. 1996. “Reinforcement learning with selective perception and hidden state”. *PhD thesis*. University of Rochester.
- McGovern, A., R. S. Sutton, and A. H. Fagg. 1997. “Roles of macro-actions in accelerating reinforcement learning”. In: *Grace Hopper celebration of women in computing*. Vol. 1317.
- Miikkulainen, R., J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, A. Navruzyan, N. Duffy, and B. Hodjat. 2017. “Evolving Deep Neural Networks”. *arXiv preprint arXiv:1703.00548*.
- Mirowski, P., R. Pascanu, F. Viola, H. Soyer, A. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, et al. 2016. “Learning to navigate in complex environments”. *arXiv preprint arXiv:1611.03673*.
- Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. 2016. “Asynchronous methods for deep reinforcement learning”. In: *International Conference on Machine Learning*.
- Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. 2015. “Human-level control through deep reinforcement learning”. *Nature*. 518(7540): 529–533.

- Mohamed, S. and D. J. Rezende. 2015. “Variational information maximisation for intrinsically motivated reinforcement learning”. In: *Advances in neural information processing systems*. 2125–2133.
- Montague, P. R. 2013. “Reinforcement Learning Models Then-and-Now: From Single Cells to Modern Neuroimaging”. In: *20 Years of Computational Neuroscience*. Springer. 271–277.
- Moore, A. W. 1990. “Efficient memory-based learning for robot control”.
- Morari, M. and J. H. Lee. 1999. “Model predictive control: past, present and future”. *Computers & Chemical Engineering*. 23(4-5): 667–682.
- Moravčík, M., M. Schmid, N. Burch, V. Lisy, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling. 2017. “DeepStack: Expert-level artificial intelligence in heads-up no-limit poker”. *Science*. 356(6337): 508–513.
- Mordatch, I., K. Lowrey, G. Andrew, Z. Popovic, and E. V. Todorov. 2015. “Interactive control of diverse complex characters with neural networks”. In: *Advances in Neural Information Processing Systems*. 3132–3140.
- Morimura, T., M. Sugiyama, H. Kashima, H. Hachiya, and T. Tanaka. 2010. “Nonparametric return distribution approximation for reinforcement learning”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 799–806.
- Munos, R. and A. Moore. 2002. “Variable resolution discretization in optimal control”. *Machine learning*. 49(2): 291–323.
- Munos, R., T. Stepleton, A. Harutyunyan, and M. Bellemare. 2016. “Safe and efficient off-policy reinforcement learning”. In: *Advances in Neural Information Processing Systems*. 1046–1054.
- Murphy, K. P. 2012. “Machine Learning: A Probabilistic Perspective.”
- Nagabandi, A., G. Kahn, R. S. Fearing, and S. Levine. 2017. “Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning”. *arXiv preprint arXiv:1708.02596*.
- Nagabandi, A., G. Kahn, R. S. Fearing, and S. Levine. 2018. “Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 7559–7566.

- Narvekar, S., J. Sinapov, M. Leonetti, and P. Stone. 2016. “Source task creation for curriculum learning”. In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 566–574.
- Neelakantan, A., Q. V. Le, and I. Sutskever. 2015. “Neural programmer: Inducing latent programs with gradient descent”. *arXiv preprint arXiv:1511.04834*.
- Neu, G. and C. Szepesvári. 2012. “Apprenticeship learning using inverse reinforcement learning and gradient methods”. *arXiv preprint arXiv:1206.5264*.
- Ng, A. Y., D. Harada, and S. Russell. 1999. “Policy invariance under reward transformations: Theory and application to reward shaping”. In: *ICML*. Vol. 99. 278–287.
- Ng, A. Y., S. J. Russell, et al. 2000. “Algorithms for inverse reinforcement learning.” In: *Icml*. 663–670.
- Nguyen, D. H. and B. Widrow. 1990. “Neural networks for self-learning control systems”. *IEEE Control systems magazine*. 10(3): 18–23.
- Niv, Y. 2009. “Reinforcement learning in the brain”. *Journal of Mathematical Psychology*. 53(3): 139–154.
- Niv, Y. and P. R. Montague. 2009. “Theoretical and empirical studies of learning”. In: *Neuroeconomics*. Elsevier. 331–351.
- Norris, J. R. 1998. *Markov chains*. No. 2. Cambridge university press.
- O’Donoghue, B., R. Munos, K. Kavukcuoglu, and V. Mnih. 2016. “PGQ: Combining policy gradient and Q-learning”. *arXiv preprint arXiv:1611.01626*.
- Oh, J., V. Chockalingam, S. Singh, and H. Lee. 2016. “Control of Memory, Active Perception, and Action in Minecraft”. *arXiv preprint arXiv:1605.09128*.
- Oh, J., X. Guo, H. Lee, R. L. Lewis, and S. Singh. 2015. “Action-conditional video prediction using deep networks in atari games”. In: *Advances in Neural Information Processing Systems*. 2863–2871.
- Oh, J., S. Singh, and H. Lee. 2017. “Value Prediction Network”. *arXiv preprint arXiv:1707.03497*.
- Olah, C., A. Mordvintsev, and L. Schubert. 2017. “Feature Visualization”. *Distill*. <https://distill.pub/2017/feature-visualization>.

- Ortner, R., O.-A. Maillard, and D. Ryabko. 2014. “Selecting near-optimal approximate state representations in reinforcement learning”. In: *International Conference on Algorithmic Learning Theory*. Springer. 140–154.
- Osband, I., C. Blundell, A. Pritzel, and B. Van Roy. 2016. “Deep Exploration via Bootstrapped DQN”. *arXiv preprint arXiv:1602.04621*.
- Ostrovski, G., M. G. Bellemare, A. v. d. Oord, and R. Munos. 2017. “Count-based exploration with neural density models”. *arXiv preprint arXiv:1703.01310*.
- Paine, T. L., S. G. Colmenarejo, Z. Wang, S. Reed, Y. Aytar, T. Pfaff, M. W. Hoffman, G. Barth-Maron, S. Cabi, D. Budden, et al. 2018. “One-Shot High-Fidelity Imitation: Training Large-Scale Deep Nets with RL”. *arXiv preprint arXiv:1810.05017*.
- Parisotto, E., J. L. Ba, and R. Salakhutdinov. 2015. “Actor-mimic: Deep multitask and transfer reinforcement learning”. *arXiv preprint arXiv:1511.06342*.
- Pascanu, R., Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. Reichert, T. Weber, D. Wierstra, and P. Battaglia. 2017. “Learning model-based planning from scratch”. *arXiv preprint arXiv:1707.06170*.
- Pathak, D., P. Agrawal, A. A. Efros, and T. Darrell. 2017. “Curiosity-driven exploration by self-supervised prediction”. In: *International Conference on Machine Learning (ICML)*. Vol. 2017.
- Pavlov, I. P. 1927. *Conditioned reflexes*. Oxford University Press.
- Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. 2011. “Scikit-learn: Machine learning in Python”. *Journal of Machine Learning Research*. 12(Oct): 2825–2830.
- Peng, J. and R. J. Williams. 1994. “Incremental multi-step Q-learning”. In: *Machine Learning Proceedings 1994*. Elsevier. 226–232.
- Peng, P., Q. Yuan, Y. Wen, Y. Yang, Z. Tang, H. Long, and J. Wang. 2017a. “Multiagent Bidirectionally-Coordinated Nets for Learning to Play StarCraft Combat Games”. *arXiv preprint arXiv:1703.10069*.

- Peng, X. B., G. Berseth, K. Yin, and M. van de Panne. 2017b. “DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning”. *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)*. 36(4).
- Perez-Liebana, D., S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson. 2016. “The 2014 general video game playing competition”. *IEEE Transactions on Computational Intelligence and AI in Games*. 8(3): 229–243.
- Petrik, M. and B. Scherrer. 2009. “Biasing approximate dynamic programming with a lower discount factor”. In: *Advances in neural information processing systems*. 1265–1272.
- Piketty, T. 2013. “Capital in the Twenty-First Century”.
- Pineau, J., G. Gordon, S. Thrun, *et al.* 2003. “Point-based value iteration: An anytime algorithm for POMDPs”. In: *IJCAI*. Vol. 3. 1025–1032.
- Pinto, L., M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel. 2017. “Asymmetric Actor Critic for Image-Based Robot Learning”. *arXiv preprint arXiv:1710.06542*.
- Plappert, M., R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz. 2017. “Parameter Space Noise for Exploration”. *arXiv preprint arXiv:1706.01905*.
- Precup, D. 2000. “Eligibility traces for off-policy policy evaluation”. *Computer Science Department Faculty Publication Series*: 80.
- Ranzato, M., S. Chopra, M. Auli, and W. Zaremba. 2015. “Sequence level training with recurrent neural networks”. *arXiv preprint arXiv:1511.06732*.
- Rasmussen, C. E. 2004. “Gaussian processes in machine learning”. In: *Advanced lectures on machine learning*. Springer. 63–71.
- Ravindran, B. and A. G. Barto. 2004. “An algebraic approach to abstraction in reinforcement learning”. *PhD thesis*. University of Massachusetts at Amherst.
- Real, E., S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. Le, and A. Kurakin. 2017. “Large-Scale Evolution of Image Classifiers”. *arXiv preprint arXiv:1703.01041*.
- Reed, S. and N. De Freitas. 2015. “Neural programmer-interpreters”. *arXiv preprint arXiv:1511.06279*.

- Rescorla, R. A., A. R. Wagner, *et al.* 1972. “A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement”. *Classical conditioning II: Current research and theory*. 2: 64–99.
- Riedmiller, M. 2005. “Neural fitted Q iteration—first experiences with a data efficient neural reinforcement learning method”. In: *Machine Learning: ECML 2005*. Springer. 317–328.
- Riedmiller, M., R. Hafner, T. Lampe, M. Neunert, J. Degrave, T. Van de Wiele, V. Mnih, N. Heess, and J. T. Springenberg. 2018. “Learning by Playing - Solving Sparse Reward Tasks from Scratch”. *arXiv preprint arXiv:1802.10567*.
- Rowland, M., M. G. Bellemare, W. Dabney, R. Munos, and Y. W. Teh. 2018. “An Analysis of Categorical Distributional Reinforcement Learning”. *arXiv preprint arXiv:1802.08163*.
- Ruder, S. 2017. “An overview of multi-task learning in deep neural networks”. *arXiv preprint arXiv:1706.05098*.
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1988. “Learning representations by back-propagating errors”. *Cognitive modeling*. 5(3): 1.
- Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.* 2015. “Imagenet large scale visual recognition challenge”. *International Journal of Computer Vision*. 115(3): 211–252.
- Russek, E. M., I. Momennejad, M. M. Botvinick, S. J. Gershman, and N. D. Daw. 2017. “Predictive representations can link model-based reinforcement learning to model-free mechanisms”. *bioRxiv*: 083857.
- Rusu, A. A., S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell. 2015. “Policy distillation”. *arXiv preprint arXiv:1511.06295*.
- Rusu, A. A., M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell. 2016. “Sim-to-real robot learning from pixels with progressive nets”. *arXiv preprint arXiv:1610.04286*.
- Sadeghi, F. and S. Levine. 2016. “CAD2RL: Real single-image flight without a single real image”. *arXiv preprint arXiv:1611.04201*.

- Salge, C., C. Glackin, and D. Polani. 2014. “Changing the environment based on empowerment as intrinsic motivation”. *Entropy*. 16(5): 2789–2819.
- Salimans, T., J. Ho, X. Chen, and I. Sutskever. 2017. “Evolution Strategies as a Scalable Alternative to Reinforcement Learning”. *arXiv preprint arXiv:1703.03864*.
- Samuel, A. L. 1959. “Some studies in machine learning using the game of checkers”. *IBM Journal of research and development*. 3(3): 210–229.
- Sandve, G. K., A. Nekrutenko, J. Taylor, and E. Hovig. 2013. “Ten simple rules for reproducible computational research”. *PLoS computational biology*. 9(10): e1003285.
- Santoro, A., D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap. 2017. “A simple neural network module for relational reasoning”. *arXiv preprint arXiv:1706.01427*.
- Savinov, N., A. Raichuk, R. Marinier, D. Vincent, M. Pollefeys, T. Lillicrap, and S. Gelly. 2018. “Episodic Curiosity through Reachability”. *arXiv preprint arXiv:1810.02274*.
- Schaarschmidt, M., A. Kuhnle, and K. Fricke. 2017. “TensorForce: A TensorFlow library for applied reinforcement learning”.
- Schaul, T., J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber. 2010. “PyBrain”. *The Journal of Machine Learning Research*. 11: 743–746.
- Schaul, T., D. Horgan, K. Gregor, and D. Silver. 2015a. “Universal value function approximators”. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 1312–1320.
- Schaul, T., J. Quan, I. Antonoglou, and D. Silver. 2015b. “Prioritized Experience Replay”. *arXiv preprint arXiv:1511.05952*.
- Schmidhuber, J. 2010. “Formal theory of creativity, fun, and intrinsic motivation (1990–2010)”. *IEEE Transactions on Autonomous Mental Development*. 2(3): 230–247.
- Schmidhuber, J. 2015. “Deep learning in neural networks: An overview”. *Neural Networks*. 61: 85–117.
- Schraudolph, N. N., P. Dayan, and T. J. Sejnowski. 1994. “Temporal difference learning of position evaluation in the game of Go”. In: *Advances in Neural Information Processing Systems*. 817–824.

- Schulman, J., P. Abbeel, and X. Chen. 2017a. “Equivalence Between Policy Gradients and Soft Q-Learning”. *arXiv preprint arXiv:1704.06440*.
- Schulman, J., J. Ho, C. Lee, and P. Abbeel. 2016. “Learning from demonstrations through the use of non-rigid registration”. In: *Robotics Research*. Springer. 339–354.
- Schulman, J., S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. 2015. “Trust Region Policy Optimization”. In: *ICML*. 1889–1897.
- Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. 2017b. “Proximal policy optimization algorithms”. *arXiv preprint arXiv:1707.06347*.
- Schultz, W., P. Dayan, and P. R. Montague. 1997. “A neural substrate of prediction and reward”. *Science*. 275(5306): 1593–1599.
- Shannon, C. 1950. “Programming a Computer for Playing Chess”. *Philosophical Magazine*. 41(314).
- Silver, D. L., Q. Yang, and L. Li. 2013. “Lifelong Machine Learning Systems: Beyond Learning Algorithms.” In: *AAAI Spring Symposium: Lifelong Machine Learning*. Vol. 13. 05.
- Silver, D., H. van Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto, *et al.* 2016a. “The predictron: End-to-end learning and planning”. *arXiv preprint arXiv:1612.08810*.
- Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.* 2016b. “Mastering the game of Go with deep neural networks and tree search”. *Nature*. 529(7587): 484–489.
- Silver, D., G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. 2014. “Deterministic Policy Gradient Algorithms”. In: *ICML*.
- Singh, S. P., T. S. Jaakkola, and M. I. Jordan. 1994. “Learning Without State-Estimation in Partially Observable Markovian Decision Processes.” In: *ICML*. 284–292.
- Singh, S. P. and R. S. Sutton. 1996. “Reinforcement learning with replacing eligibility traces”. *Machine learning*. 22(1-3): 123–158.
- Singh, S., T. Jaakkola, M. L. Littman, and C. Szepesvári. 2000. “Convergence results for single-step on-policy reinforcement-learning algorithms”. *Machine learning*. 38(3): 287–308.

- Sondik, E. J. 1978. “The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs”. *Operations research*. 26(2): 282–304.
- Srivastava, N., G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. 2014. “Dropout: a simple way to prevent neural networks from overfitting.” *Journal of Machine Learning Research*. 15(1): 1929–1958.
- Stadie, B. C., S. Levine, and P. Abbeel. 2015. “Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models”. *arXiv preprint arXiv:1507.00814*.
- Stone, P. and M. Veloso. 2000. “Layered learning”. *Machine Learning: ECML 2000*: 369–381.
- Story, G., I. Vlaev, B. Seymour, A. Darzi, and R. Dolan. 2014. “Does temporal discounting explain unhealthy behavior? A systematic review and reinforcement learning perspective”. *Frontiers in behavioral neuroscience*. 8: 76.
- Sukhbaatar, S., A. Szlam, and R. Fergus. 2016. “Learning multiagent communication with backpropagation”. In: *Advances in Neural Information Processing Systems*. 2244–2252.
- Sun, Y., F. Gomez, and J. Schmidhuber. 2011. “Planning to be surprised: Optimal bayesian exploration in dynamic environments”. In: *Artificial General Intelligence*. Springer. 41–51.
- Sunehag, P., G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, *et al.* 2017. “Value-Decomposition Networks For Cooperative Multi-Agent Learning”. *arXiv preprint arXiv:1706.05296*.
- Sutton, R. S. 1988. “Learning to predict by the methods of temporal differences”. *Machine learning*. 3(1): 9–44.
- Sutton, R. S. 1996. “Generalization in reinforcement learning: Successful examples using sparse coarse coding”. *Advances in neural information processing systems*: 1038–1044.
- Sutton, R. S. and A. G. Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. No. 1. MIT press Cambridge.
- Sutton, R. S. and A. G. Barto. 2017. *Reinforcement Learning: An Introduction (2nd Edition, in progress)*. MIT Press.

- Sutton, R. S., D. A. McAllester, S. P. Singh, and Y. Mansour. 2000. “Policy gradient methods for reinforcement learning with function approximation”. In: *Advances in neural information processing systems*. 1057–1063.
- Sutton, R. S., D. Precup, and S. Singh. 1999. “Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning”. *Artificial intelligence*. 112(1-2): 181–211.
- Sutton, R. S. 1984. “Temporal credit assignment in reinforcement learning”.
- Synnaeve, G., N. Nardelli, A. Auvolat, S. Chintala, T. Lacroix, Z. Lin, F. Richoux, and N. Usunier. 2016. “TorchCraft: a Library for Machine Learning Research on Real-Time Strategy Games”. *arXiv preprint arXiv:1611.00625*.
- Szegedy, C., S. Ioffe, V. Vanhoucke, and A. Alemi. 2016. “Inception-v4, inception-resnet and the impact of residual connections on learning”. *arXiv preprint arXiv:1602.07261*.
- Szegedy, C., S. Ioffe, V. Vanhoucke, and A. A. Alemi. 2017. “Inception-v4, inception-resnet and the impact of residual connections on learning.” In: *AAAI*. Vol. 4. 12.
- Tamar, A., S. Levine, P. Abbeel, Y. Wu, and G. Thomas. 2016. “Value iteration networks”. In: *Advances in Neural Information Processing Systems*. 2146–2154.
- Tan, J., T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke. 2018. “Sim-to-Real: Learning Agile Locomotion For Quadruped Robots”. *arXiv preprint arXiv:1804.10332*.
- Tanner, B. and A. White. 2009. “RL-Glue: Language-independent software for reinforcement-learning experiments”. *The Journal of Machine Learning Research*. 10: 2133–2136.
- Teh, Y. W., V. Bapst, W. M. Czarnecki, J. Quan, J. Kirkpatrick, R. Hadsell, N. Heess, and R. Pascanu. 2017. “Distral: Robust Multitask Reinforcement Learning”. *arXiv preprint arXiv:1707.04175*.
- Tesauro, G. 1995. “Temporal difference learning and TD-Gammon”. *Communications of the ACM*. 38(3): 58–68.
- Tessler, C., S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor. 2017. “A Deep Hierarchical Approach to Lifelong Learning in Minecraft.” In: *AAAI*. 1553–1561.

- Thomas, P. 2014. “Bias in natural actor-critic algorithms”. In: *International Conference on Machine Learning*. 441–448.
- Thomas, P. S. and E. Brunskill. 2016. “Data-efficient off-policy policy evaluation for reinforcement learning”. In: *International Conference on Machine Learning*.
- Thrun, S. B. 1992. “Efficient exploration in reinforcement learning”.
- Tian, Y., Q. Gong, W. Shang, Y. Wu, and C. L. Zitnick. 2017. “ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games”. *Advances in Neural Information Processing Systems (NIPS)*.
- Tieleman, H. 2012. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. *COURSERA: Neural Networks for Machine Learning*.
- Tobin, J., R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. 2017. “Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World”. *arXiv preprint arXiv:1703.06907*.
- Todorov, E., T. Erez, and Y. Tassa. 2012. “MuJoCo: A physics engine for model-based control”. In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE. 5026–5033.
- Tsitsiklis, J. N. and B. Van Roy. 1997. “An analysis of temporal-difference learning with function approximation”. *Automatic Control, IEEE Transactions on*. 42(5): 674–690.
- Turing, A. M. 1953. “Digital computers applied to games”. *Faster than thought*.
- Tzeng, E., C. Devin, J. Hoffman, C. Finn, P. Abbeel, S. Levine, K. Saenko, and T. Darrell. 2015. “Adapting deep visuomotor representations with weak pairwise constraints”. *arXiv preprint arXiv:1511.07111*.
- Ueno, S., M. Osawa, M. Imai, T. Kato, and H. Yamakawa. 2017. ““Re: ROS”: Prototyping of Reinforcement Learning Environment for Asynchronous Cognitive Architecture”. In: *First International Early Research Career Enhancement School on Biologically Inspired Cognitive Architectures*. Springer. 198–203.
- Van Hasselt, H., A. Guez, and D. Silver. 2016. “Deep Reinforcement Learning with Double Q-Learning.” In: *AAAI*. 2094–2100.

- Vapnik, V. N. 1998. “Statistical learning theory. Adaptive and learning systems for signal processing, communications, and control”.
- Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. 2017. “Attention Is All You Need”. *arXiv preprint arXiv:1706.03762*.
- Vezhnevets, A., V. Mnih, S. Osindero, A. Graves, O. Vinyals, J. Agapiou, et al. 2016. “Strategic attentive writer for learning macro-actions”. In: *Advances in Neural Information Processing Systems*. 3486–3494.
- Vinyals, O., T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, et al. 2017. “StarCraft II: A New Challenge for Reinforcement Learning”. *arXiv preprint arXiv:1708.04782*.
- Wahlström, N., T. B. Schön, and M. P. Deisenroth. 2015. “From pixels to torques: Policy learning with deep dynamical models”. *arXiv preprint arXiv:1502.02251*.
- Walsh, T. 2017. *It’s Alive!: Artificial Intelligence from the Logic Piano to Killer Robots*. La Trobe University Press.
- Wang, J. X., Z. Kurth-Nelson, D. Tirumala, H. Soyer, J. Z. Leibo, R. Munos, C. Blundell, D. Kumaran, and M. Botvinick. 2016a. “Learning to reinforcement learn”. *arXiv preprint arXiv:1611.05763*.
- Wang, Z., V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas. 2016b. “Sample efficient actor-critic with experience replay”. *arXiv preprint arXiv:1611.01224*.
- Wang, Z., N. de Freitas, and M. Lanctot. 2015. “Dueling network architectures for deep reinforcement learning”. *arXiv preprint arXiv:1511.06581*.
- Warnell, G., N. Waytowich, V. Lawhern, and P. Stone. 2017. “Deep TAMER: Interactive Agent Shaping in High-Dimensional State Spaces”. *arXiv preprint arXiv:1709.10163*.
- Watkins, C. J. and P. Dayan. 1992. “Q-learning”. *Machine learning*. 8(3-4): 279–292.
- Watkins, C. J. C. H. 1989. “Learning from delayed rewards”. *PhD thesis*. King’s College, Cambridge.

- Watter, M., J. Springenberg, J. Boedecker, and M. Riedmiller. 2015. “Embed to control: A locally linear latent dynamics model for control from raw images”. In: *Advances in neural information processing systems*. 2746–2754.
- Weber, T., S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, et al. 2017. “Imagination-Augmented Agents for Deep Reinforcement Learning”. *arXiv preprint arXiv:1707.06203*.
- Wender, S. and I. Watson. 2012. “Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft: Broodwar”. In: *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE. 402–408.
- Whiteson, S., B. Tanner, M. E. Taylor, and P. Stone. 2011. “Protecting against evaluation overfitting in empirical reinforcement learning”. In: *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on*. IEEE. 120–127.
- Williams, R. J. 1992. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. *Machine learning*. 8(3-4): 229–256.
- Wu, Y. and Y. Tian. 2016. “Training agent for first-person shooter game with actor-critic curriculum learning”.
- Xu, K., J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. 2015. “Show, attend and tell: Neural image caption generation with visual attention”. In: *International Conference on Machine Learning*. 2048–2057.
- You, Y., X. Pan, Z. Wang, and C. Lu. 2017. “Virtual to Real Reinforcement Learning for Autonomous Driving”. *arXiv preprint arXiv:1704.03952*.
- Zamora, I., N. G. Lopez, V. M. Vilches, and A. H. Cordero. 2016. “Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo”. *arXiv preprint arXiv:1608.05742*.
- Zhang, A., N. Ballas, and J. Pineau. 2018a. “A Dissection of Overfitting and Generalization in Continuous Reinforcement Learning”. *arXiv preprint arXiv:1806.07937*.
- Zhang, A., H. Satija, and J. Pineau. 2018b. “Decoupling Dynamics and Reward for Transfer Learning”. *arXiv preprint arXiv:1804.10689*.

- Zhang, C., O. Vinyals, R. Munos, and S. Bengio. 2018c. “A Study on Overfitting in Deep Reinforcement Learning”. *arXiv preprint arXiv:1804.06893*.
- Zhang, C., S. Bengio, M. Hardt, B. Recht, and O. Vinyals. 2016. “Understanding deep learning requires rethinking generalization”. *arXiv preprint arXiv:1611.03530*.
- Zhu, Y., R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi. 2016. “Target-driven visual navigation in indoor scenes using deep reinforcement learning”. *arXiv preprint arXiv:1609.05143*.
- Ziebart, B. D. 2010. *Modeling purposeful adaptive behavior with the principle of maximum causal entropy*. Carnegie Mellon University.
- Zoph, B. and Q. V. Le. 2016. “Neural architecture search with reinforcement learning”. *arXiv preprint arXiv:1611.01578*.

Cross-Domain Transfer for Reinforcement Learning

Matthew E. Taylor

Peter Stone

Department of Computer Sciences, The University of Texas at Austin

MTAYLOR@CS.UTEXAS.EDU

PSTONE@CS.UTEXAS.EDU

Abstract

A typical goal for transfer learning algorithms is to utilize knowledge gained in a source task to learn a target task faster. Recently introduced transfer methods in reinforcement learning settings have shown considerable promise, but they typically transfer between pairs of very similar tasks. This work introduces *Rule Transfer*, a transfer algorithm that first learns rules to summarize a source task policy and then leverages those rules to learn faster in a target task. This paper demonstrates that Rule Transfer can effectively speed up learning in Keepaway, a benchmark RL problem in the robot soccer domain, based on experience from source tasks in the gridworld domain. We empirically show, through the use of three distinct transfer metrics, that Rule Transfer is effective across these domains.

1. Introduction

Reinforcement learning (RL) methods excel at solving complex tasks with minimal feedback. *Transfer learning*, in an RL setting, typically attempts to decrease training time by learning a *source task* before learning the *target task*. While there have been a number of recent successes, most existing transfer methods focus on pairs of tasks that are closely related, such as different mazes where agents have the same sensors and actions available (Madden & Howley, 2004). Prior to this work, the most dissimilar source and target tasks we are aware of are pairs of tasks in a single domain with different reward structures, different actions, and/or different state variables (see, for instance, past transfer work in the robot soccer domain (Torrey et al., 2006)).

A more difficult challenge is to transfer between different domains, where we informally define a *domain* to be a setting for a group of semantically similar *tasks*. Such cross-domain transfer has been a long-term goal of transfer learning because it could allow transfer between significantly less similar tasks. While previous transfer work has focused on reducing training time by transferring from a simple to complex task in a single domain, a (potentially) more powerful way of simplifying a task is to formulate it

as an abstraction in a different domain. This work will focus on demonstrating that cross-domain transfer is feasible and effective, where the source task is selected from the relatively simple gridworld domain and the target task is the more complex RL benchmark task of 3 vs. 2 Keepaway in the robot soccer domain.

We first introduce *Rule Transfer*, a novel domain-independent RL transfer method. In addition to succeeding at cross-domain transfer and having low computational requirements in practice, this method allows production rules (henceforth *rules*) to transfer knowledge between agents which may have different internal representations. Thus an agent may train very quickly with a simple internal representation in the source task, but a more complex agent in the target task could still benefit from the transfer.

This paper evaluates three different rule utilization schemes for Rule Transfer in Keepaway. We then empirically show that cross-domain transfer can effectively improve the speed of learning if the relationships between the tasks is known. Lastly, learning such relationships is an important open problem and this work also takes a step towards learning the mapping, illustrating a process by which it can be derived from high-level qualitative knowledge.

2. Rule Transfer

The following steps summarize Rule Transfer:

1. **Learn a policy** ($\pi : S \mapsto A$) **in the source task**. After training has finished, or during the final training episodes, the agent records some number of interactions with the environment in the form of (S, A) pairs while following the learned policy.
2. **Learn a decision list** ($D_s : S \mapsto A$) **that summarizes the source policy**. After the data is collected, a rule learner is used to summarize the collected data to approximate the learned policy.
3. **Modify the decision list for use in the target task** ($\text{Translate } (D_s) \rightarrow D_t$). To allow the learned decision list to be applied in a target task that has different state variables and actions from the source task, the decision list must be translated before it can be used.
4. **Use D_t to learn a policy in the target task**. Section 2.2 will discuss how the transferred rules can be used in the target task to speed up learning.

The primary difference between this transfer method and previous work is that we leverage rules to provide an ab-

Appearing in *Proceedings of the 24th International Conference on Machine Learning*, Corvallis, OR, 2007. Copyright 2007 by the author(s)/owner(s).

stract representation of a source task policy that is usable in the target task. We choose rules because rule learning is fast and well understood, and they are human readable. By using rules as an intermediate representation, we decouple the particular learning techniques used in the two tasks. Other intermediate representations, such as neural networks, are possible in principle. As long as rules may be abstracted from the source agent’s behavior and leveraged by the target agent, agents in the two tasks may use different internal representations, as best suits their particular task.

2.1. Task-Specific Rule Translation

If the target task has different state variables or actions than the source task, an agent could not directly apply a learned decision list from the source task because the preconditions for the rules and/or actions recommended have changed. To define a relationship between two tasks, we define a pair of translation functions for actions and state variables, which we initially assume are provided. δ_A returns the target task action most similar to a source task action ($\delta_A(A_s) = A_t$) and δ_X returns the target task state variable most similar to a source state variable ($\delta_X(x_s) = x_t$). Together, these functions define how a pair of tasks are related.

`Translate()` procedurally modifies the source task decision list so that it can apply to a given target task by directly mapping state variables and actions between the tasks in the spirit of past transfer work (Soni & Singh, 2006; Torrey et al., 2006). In our experiments the source tasks’ state variables and actions all had mappings into the target task. If source task state variables or actions had no correspondence in the target task, such preconditions or rules would be removed from the translated decision list.

2.2. Rule Utilization

To make Rule Transfer effective, we treat the translated decision list as *advice* rather than as rules that must be followed. The agent benefits from the decision list initially, but refines its policy as it gathers more experience in the target task. This section introduces three advice utilization schemes. The first applies if the target task learner is using a value-function approximation method, like *temporal difference learning* (Sutton & Barto, 1998), but the second and third apply to other RL learning methods in principle.

Value Bonus uses the transferred decision list, D_t , to determine which target task action the decision list would recommend in the current state. The computed Q-value of this recommended action then receives a “bonus” so that it is increased by some constant. Actions recommended by D_t are initially more likely to be selected, but the bonus can be negated over time through learning.

Extra Action adds an action to the target task. When the target task agent selects this pseudo-action, the agent executes the action recommended by D_t . The learner treats this

pseudo-action as a normal action. To bias the learner towards this action, the agent is forced to execute the pseudo-action for a constant number of episodes at the beginning of training in the target task. Afterward, assuming pessimistic initialization, the pseudo-action will have higher Q-values than all other actions, which causes the agent to initially perform recommended actions, but over time learning can override this bias. For instance, in regions of the state space in which the advice is appropriate, the agent will learn to select the pseudo-action, while in other regions of the state space where the advice is non-optimal, the agent must learn to intelligently choose between all the actions.

Extra Variable adds an extra state variable to the target task’s state description. This variable takes on the value of the index for the action recommended by D_t . To assist the agent in learning the importance of this variable, we again initially force the agent to choose the action recommended by D_t . An agent quickly learns the importance of this state variable, but it can still learn to ignore the state variable when the advice is sub-optimal.

3. Task Descriptions

In this section we first describe 3 vs. 2 Keepaway, an RL benchmark task (Stone et al., 2006) in the domain of robot soccer. Next we introduce Ringworld and Knight Joust, novel tasks in the gridworld domain which are designed to exhibit similarities to Keepaway.

3.1. Keepaway

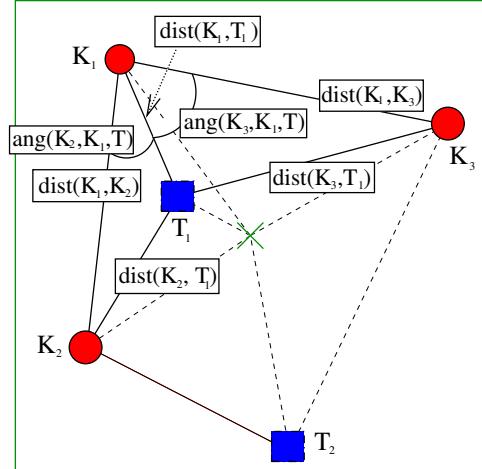


Figure 1. The 13 state variables used for learning 3 vs. 2 Keepaway, 7 of which are labeled with solid lines. There are 11 distances to players and the center of the field, as well as 2 angles along passing lanes.

Keepaway tasks in the RoboCup simulated Soccer domain are characterized by stochastic actions, noisy observations, and a continuous state space. In 3 vs. 2 Keepaway, a team of 3 *keepers* tries to possess a ball in a square area while 2 *takers* work to foil the keepers (see Figure 1). Over time the keepers attempt to learn to possess the ball longer, increas-

ing the average episode length. The players in our experiments are based on version 0.6 of the benchmark players distributed by UT Austin (Stone et al., 2006) and freely-available Soccer Server version 9.4.5.

In 3 vs. 2 Keepaway, three keepers are initially placed in three corners of a 25m \times 25m field and a ball is placed near one of the keepers. Two takers are placed in the fourth corner. When an episode starts, the three keepers attempt to keep control of the ball by passing among themselves and moving to open positions. The keepers receive a reward of +1 for every time step that the ball remains in play. The episode finishes when a taker gains control of the ball or the ball is kicked out of bounds. The episode is then reset with a random keeper placed near the ball.

Keepers can choose from 3 macro-actions when possessing the ball: $A = \{hold, Pass_1, Pass_2\}$. Keepers which do not possess the ball follow a hand-coded policy which attempts to capture an open ball or moves to get open for a pass. Takers follow a fixed strategy and do not learn. The agent’s state is defined by 13 variables, as is shown by line segments and angles in Figure 1. These variables describe relevant distance and angles of the keepers $K_1 - K_n$, the takers $T_1 - T_m$, and the center of the playing region, C . Keepers and takers are ordered by increasing distance from the ball. Learning in this multi-agent domain is complicated by a continuous state space and (simulated) noise in agent sensors and actuators.

Keepers learn using Sarsa (Rummery & Niranjan, 1994; Singh & Sutton, 1996) for estimating the action-value function. Because Keepaway has a continuous state space, some kind of function approximation is necessary. We utilized a radial basis function approximator (Sutton & Barto, 1998), as was done previously in this domain (Stone et al., 2006).

3.2. Ringworld

Ringworld is a novel task that is situated on a grid¹ with 0.01 meter tiles. There is no noise in agents’ perceptions and the player receives a reward of +1 for every time step it is not captured by the opponent. The opponent always moves towards the player. When the episode starts, the player is randomly assigned two possible “Run Targets,” which lie on a static ring. At every timestep, the player may either stay in its current location, or choose to “run” to one of the two targets. If the player runs, it moves at twice the speed of the opponent to the run target. If the opponent does not intercept the player, as determined by the transition function, two new random run targets on the ring are chosen for the player and the episode continues. As the opponent approaches the player, either when the player is standing still or while running, the chance of capture increases. Thus

¹An agent sees an average of only 8065 distinct states over the course of a 25,000 episode learning trial.

the only stochasticity in the environment is the randomness associated with the probability of capture. The player learns to maximize average episode length by using Sarsa with a Q-value table. $A = \{Stay, RunNear, RunFar\}$ and the state is represented by 5 distances and 2 angles (see Figure 2).

This gridworld task was constructed so that it would have similarities to 3 vs. 2 Keepaway. For instance, the 7 state variables were chosen to be similar to the state variables in Keepaway. The width of the ring (9.5 m) was selected so that the distance between the distance between the player and the target is similar, on average, to the

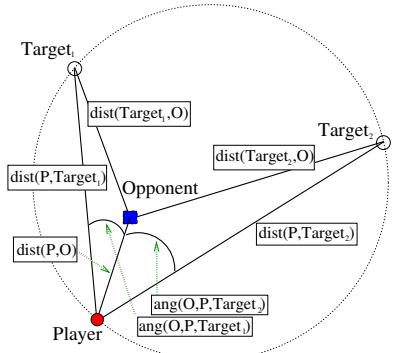


Figure 2. Ringworld: The player may stay still or run to 1 of 2 possible targets. The episode ends when the opponent captures the player. The underlying 0.1m grid is not shown.

distance between keepers. The transition function that determines if the episode ends (which takes as input the state variable $dist(P, O)$: the distance between the player and opponent) was based on the observed likelihood that a Keepaway episode ends (i.e. the probability that an episode ends given $dist(K_1, T_1)$). While it would be impossible to recreate many of the dynamics associated with a complex, stochastic, and continuous task, we designed Ringworld to capture some of Keepaway’s characteristics.

3.3. Knight Joust

Knight Joust is also situated in the gridworld domain but was designed to be less similar to Keepaway than Ringworld, and more simple than Ringworld. In this task the player begins on one end of a 25m \times 25m board, the opponent begins on the other, and the players alternate moves. The

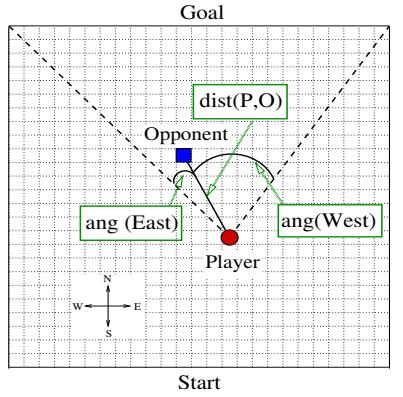


Figure 3. Knight Joust: The player attempts to reach the goal end of a 25 \times 25 grid-world while the opponent attempts to touch the player.

player’s goal is to reach the opposite end of the board without being touched by the opponent (see Figure 3); the episode ends if the player reaches the goal line or the

opponent is on the same square as the player. The state space is discretized into 1m squares² and again there is no noise in the perception. The player's state variables are composed of the distance from the player to the opponent, and two angles which describe how much of the goal line is viewable by the player.

```

if opponent is E of player then
    Move W with probability 0.9
else if opponent is W of player then
    Move E with probability 0.9
if opponent is N of player then
    Move S with probability 1.0
else if opponent is S of player then
    Move N with probability 0.8

```

Figure 4. Knight Joust opponent policy

$A = \{Forward, Jump_{West}, Jump_{East}\}$. The opponent may move in any of 8 directions and follows a fixed stochastic policy³, shown in Figure 4.

The player receives a reward of +20 every time it takes the forward action, a +20 upon reaching the goal line, and 0 otherwise. The player uses Sarsa with a tabular representation to learn in this task. While this task is quite dissimilar from Keepaway, note that there are some similarities, such as favoring larger distances between player and opponent.

4. Utilizing Rules Effectively

To determine reasonable settings for the different rule utilization methods outlined in Section 2.2, we analyze Rule Transfer by using 3 vs. 2 Keepaway as the source *and* target task. We first train in 3 vs. 2 for five simulator hours (roughly 1,300 episodes). Next, JRip, an implementation of RIPPER (Cohen, 1995) included in Weka (Witten & Frank, 2005), learns a decision list summarizing the source task policy⁴. Lastly, we utilize the decision list in a new instance of Keepaway.

We will compare the different rule utilization methods to learning without transfer via three metrics:

1. Initial Performance: Measure the average hold time at time = 0.
2. Asymptotic Performance: Measure the average hold time after learning has plateaued. We measure this after 40 simulator hours because initial informal ex-

²A learner in Knight Joust sees an average of only 601 distinct states over a 50,000 episode learning trial

³If the opponent could move in any direction with probability 1.0, the player would never succeed in reaching the goal line. The player can only take a limited number of jumps before hitting the edge of the board and must hope that the opponent "stumbles" so that the player can pass it.

⁴RIPPER is a simple propositional rule learner that can learn a decision list. If additional representational power were needed, an ILP rule learner like Aleph (Srinivasan, 2001) could be used, but we found the additional complexity unnecessary.

periments showed that learning without transfer has plateaued by this time.

3. Accumulated Reward: Approximate the area under the curve by summing the average reward accumulated by a particular trial at every hour: $\sum_{t=0}^{40} (\text{average reward at time } t)$.

Figure 5 shows the performance averaged over 10 learning curves for learning without transfer with a 1000 episode sliding window. One learning curve shows the performance when always utilizing the rules ("Always Use Rules"). Additionally, the three rule utilization methods are shown.

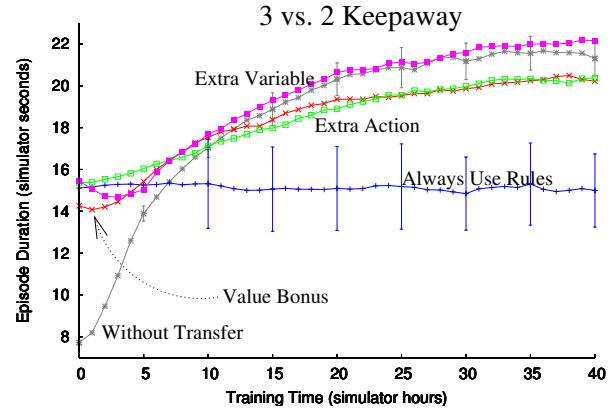


Figure 5. This graph shows the average of 10 independent learning trials for learning without transfer and for using rules after learning for five simulator hours (not shown): without further learning, with a Value Bonus of +10, with Extra Action after 100 episodes, and with Extra Variable after 100 episodes. Learning without transfer and using rules without learning have standard error bars in 5 hour increments.

Table 1 details the results, which show that while the transfer metrics are affected by the relevant rule utilization parameters, each rule utilization method has a wide range of effective parameters.

5. Cross-Domain Transfer Results

In the previous section we determined appropriate advice utilization settings via three transfer metrics. In this section we apply those same settings while using Ringworld and Knight Joust as source tasks and Keepaway as the target. This section demonstrates that transfer from Ringworld is able to significantly improve all three transfer metrics, that Rule Transfer settings are not particularly brittle, and that transfer from Knight Joust is able to significantly improve one of the transfer metrics.

5.1. Ringworld to 3 vs. 2 Keepaway

In this section we first detail how to perform Rule Transfer between Ringworld and Keepaway. We compare the results from using the three different advice utilization schemes and show that Extra Action is superior. Lastly, we perform a set of experiments to show that Rule Transfer, while it has many parameters, is not particularly sensitive to these parameters' settings.

Cross-Domain Transfer for Reinforcement Learning

Keepaway to Keepaway			
Initial Performance	Asymptotic Performance	Accumulated Reward	
Without Transfer			
7.0 ± 0.7	19.4 ± 2.0	688.4 ± 68.7	
Added Constant			
1	12.3 ± 1.7	17.1 ± 1.7	630.1 ± 61.3
5	12.7 ± 1.9	18.2 ± 1.8	666.0 ± 66.4
10	13.0 ± 1.8	18.4 ± 2.2	686.4 ± 77.5
20	12.6 ± 1.7	16.6 ± 1.7	611.9 ± 65.3
50	12.8 ± 1.9	13.9 ± 1.9	534.2 ± 73.8
Initial Episodes			
0	7.6 ± 1.7	19.0 ± 2.1	648.2 ± 72.1
50	13.8 ± 2.1	18.3 ± 2.0	676.1 ± 75.4
100	14.0 ± 2.4	18.5 ± 2.0	688.2 ± 75.7
250	13.9 ± 2.3	18.4 ± 1.9	675.3 ± 69.0
500	13.7 ± 2.2	18.1 ± 1.9	678.8 ± 72.2
1000	13.4 ± 2.0	17.9 ± 2.1	648.2 ± 72.1
Initial Episodes			
0	7.0 ± 0.7	20.0 ± 2.0	691.4 ± 68.8
50	13.6 ± 2.0	19.9 ± 2.0	715.9 ± 70.2
100	14.0 ± 2.3	20.1 ± 2.1	726.0 ± 72.4
250	13.7 ± 2.1	19.9 ± 2.1	717.6 ± 74.6
500	13.6 ± 2.2	20.2 ± 2.0	729.2 ± 73.8
1000	13.7 ± 2.4	17.4 ± 6.3	637.4 ± 207.6

Table 1. Results compare three different rule utilization schemes where each row represents 10 independent tests. The three rule metrics from Section 4 are shown in columns 2-4. The first column contains the constant added to the recommended action in Value Bonus, or the number of episodes the learner in initially forced to select the recommended action for Extra Action and Extra Variable.

Agents learn for 25,000 episodes in Ringworld and then record 20,000 (S, A) pairs, which takes less than 1,000 episodes. After JRip learns a decision list, and the rules are transformed via `Translate()` and the cross-domain mappings (Table 2), the decision list is utilized to learn Keepaway.

Table 3 shows one of the main results of this paper; all three rule utilization methods can significantly increase all three transfer metrics. Furthermore, the asymptotic performance

Cross-Domain Mappings for Ringworld to Keepaway

Ringworld	Keepaway
	δ_A
Stay	Hold Ball
Run_{Near}	$Pass_1$: Pass to K_1
Run_{Far}	$Pass_2$: Pass to K_3
	δ_X
$dist(P, O)$	$dist(K_1, T_1)$
$dist(P, Target_1)$	$dist(K_1, K_2)$
$dist(Target_1, O)$	$Min(dist(K_2, T_1), dist(K_2, T_2))$
$ang(O, P, Target_1)$	$Min(ang(K_2, K_1, T_1), ang(K_2, K_1, T_2))$
$dist(P, Target_2)$	$dist(K_1, K_3)$
$dist(Target_2, O)$	$Min(dist(K_3, T_1), dist(K_3, T_2))$
$ang(O, P, Target_2)$	$Min(ang(K_3, K_1, T_1), ang(K_3, K_1, T_2))$

Table 2. This table describes the cross-domain mapping used by `Translate()` to modify a decision list learned in Ringworld so that it can apply to Keepaway.

Ringworld to Keepaway			
Initial Performance	Asymptotic Performance	Accumulated Reward	
Without Transfer			
7.8 ± 0.1	21.6 ± 0.8	756.7 ± 21.8	
Added Constant			
5	11.1 ± 1.4	19.8 ± 0.6	722.3 ± 24.3
10	11.5 ± 1.7	22.2 ± 0.8	813.7 ± 23.6
Initial Episodes			
100	11.9 ± 1.8	23.0 ± 0.5	842.0 ± 26.9
250	11.8 ± 1.9	23.0 ± 0.8	827.4 ± 33.0
Initial Episodes			
100	11.8 ± 1.9	21.9 ± 0.9	784.8 ± 27.0
250	11.7 ± 1.8	22.4 ± 0.8	793.5 ± 22.2

Table 3. A comparison of three rule utilization schemes to learning Keepaway without transfer. Each row is the average of 20 independent trials and shows the standard error (note that the top line row uses the same settings as learning without transfer in Table 1 but with more trials). Numbers in **bold** are statistically better than learning without transfer at the 95% level, as determined via a Student's t-test.

IF (($dist(K_1, T_1) \leq 4$) AND
 $(Min(dist(K_3, T_1), dist(K_3, T_2)) \geq 12.8)$ AND
 $(ang(K_3, K_1, T) \geq 36)$) THEN Pass to K_3

Figure 6. An example transformed rule from Ringworld that would be difficult for a human to generate from domain knowledge alone.

is not adversely effected by Rule Transfer for the best parameter settings. These results show that the Extra Action rule utilization method is slightly superior to the other two methods and confirm that cross-domain transfer can be effective at increasing the speed of learning in Keepaway. Figure 7 shows learning in Keepaway without transfer and when using Extra Action Rule Transfer from Ringworld.

An example of the type of knowledge transferred from Ringworld to Keepaway, consider the transformed rule in Figure 6 from one trial. This rule demonstrates that the agent has learned that it should pass if a taker is close, there isn't a taker very close to the target teammate, and the passing angle to the teammate is open.

To further determine the robustness of transfer from Ringworld to Keepaway, we perform a series of additional studies, shown in Table 4, to determine the sensitivity of Rule Transfer to various parameter settings. First we try learning for different amounts of time in Ringworld. At 20,000 episodes the learning in Ringworld has not plateaued and it is not surprising that the initial performance in Keepaway is therefore slightly reduced. Different ring diameters make the task less similar to Keepaway, but all diameters do successfully improve one or more transfer metrics relative to learning without transfer.

We also performed experiments that examined the robustness of rule learning for transfer. In the first experiment we recorded different amounts of Ringworld data; less

Cross-Domain Transfer for Reinforcement Learning

Ringworld Sensitivity Analysis

Param	Initial Performance	Asymptotic Performance	Accumulated Reward
Episodes of Ringworld Training before Recording Data			
20,000	10.1 ± 1.7	21.8 ± 1.3	762.5 ± 44.1
25,000	11.9 ± 1.8	23.0 ± 0.5	842.0 ± 26.9
30,000	12.0 ± 1.7	20.7 ± 5.0	793.9 ± 47.8
Ringworld's Ring Diameter (m)			
7.5	14.8 ± 2.4	20.0 ± 1.5	748.2 ± 53.6
8.5	13.5 ± 1.5	21.1 ± 1.2	776.8 ± 45.2
9.5	11.9 ± 1.8	23.0 ± 0.5	842.0 ± 26.9
10.5	9.4 ± 1.0	21.5 ± 1.3	757.7 ± 42.4
11.5	8.2 ± 1.3	20.1 ± 1.6	705.0 ± 41.6
Amount of recorded Ringworld Data			
5,000	12.2 ± 1.2	20.6 ± 4.9	765.1 ± 59.4
20,000	11.9 ± 1.8	23.0 ± 0.5	842.0 ± 26.9
40,000	11.2 ± 1.3	21.4 ± 1.3	776.4 ± 46.6
JRip Settings			
N=2, O=2	13.7 ± 1.7	20.9 ± 1.3	767.3 ± 44.3
N=100, O=2	10.7 ± 1.5	21.6 ± 1.2	784.3 ± 49.9
N=100, O=10	11.9 ± 1.8	23.0 ± 0.5	842.0 ± 26.9
N=2, O=10	14.0 ± 1.8	20.9 ± 1.3	763.3 ± 44.7

Table 4. This table shows Ringworld transfer with Extra Action rule usage after forcing the action advised by D_t for 100 episodes. The settings used previously (in Table 3) are shown in bold for comparison, each row is the average over 20 independent trials, and the standard error is shown.

data would force more generalization while more data may cause overfitting. The last sensitivity analysis tried varying the parameters to JRip, again showing that the performance of the 4 metrics is not particularly sensitive to the rule learning settings as they all outperform learning without transfer. N is the minimum number of instances a rule must cover (JRip default = 2) and O is the number of optimization runs to increase generality (JRip default = 2). Thus, while there are a number of parameters to set in Rule Transfer, the parameters proved easy to set in practice and were not critical to the method’s success.

5.2. Knight Joust to 3 vs. 2 Keepaway

In this section we present the results for transferring from Knight Joust to Keepaway using the cross-domain mappings in Table 5. Briefly, the intuition is that the forward action is similar to the hold ball action because the player should take it whenever possible. Note that the we have made West in the Knight Joust correspond to K_2 and East correspond to K_3 , but either is reasonable, as long as the state variables and actions are consistent. When it is “too dangerous,” the player instead jumps to the side, similar to passing the ball. We first train the Knight Joust players for 50,000 episodes, as initial experiments showed that learners generally stopped learning after roughly this many episodes. The advice is utilized by Extra Action Rule Transfer in Keepaway (informal experiments showed that Value Bonus and Extra Variable under-performed Extra Action, as in Ringworld) and other parameters are unchanged from the previous section. The results from these experiments are presented in Table 6.

The Knight Joust task is less similar to Keepaway than

Cross-Domain Mappings for Knight Joust to Keepaway

Knight Joust	Keepaway
δ_A	
Forward	Hold Ball
$Jump_{West}$	Pass to closest keeper
$Jump_{East}$	Pass to furthest keeper
δ_X	
$dist(P, O)$	$dist(K_1, T_1)$
ang(West)	$\text{Min}(\text{ang}(K_2, K_1, T_1),$ $\text{ang}(K_2, K_1, T_2))$
ang(East)	$\text{Min}(\text{ang}(K_3, K_1, T_1),$ $\text{ang}(K_3, K_1, T_2))$

Table 5. This table describes the cross-domain mapping used by `Translate()` to modify a decision list learned in the Knight Joust so that it can apply to Keepaway.

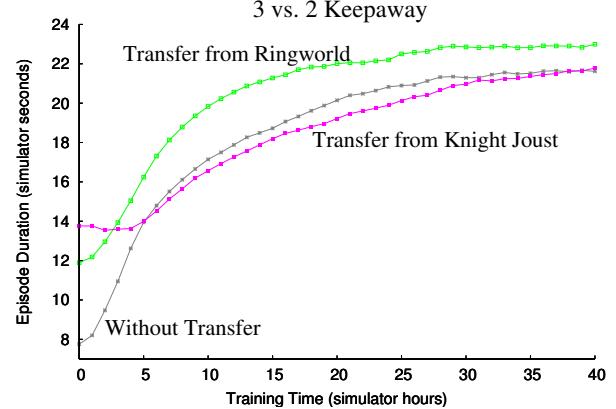


Figure 7. Learning curves in 3 vs. 2 Keepaway, averaged over 20 trials, showing learning without transfer, learning with Extra Action from Ringworld after 100 episodes of following the rule-suggested actions, and learning with Extra Action from Knight Joust after 100 episodes of following the rule-suggested actions.

Ringworld. There are many fewer state variables, a less similar transition function, and a very different reward structure. However, information from Knight Joust can significantly improve the initial performance of Keepaway players because very basic information, such as that it is desirable to maximize the distance to the opponent, will initially cause the players to perform better than acting randomly. The other two transfer metrics are not improved by transfer, however. We hypothesize that this is because the transferred knowledge, while allowing the agents to perform better than acting randomly, does not bias the learner towards an optimal policy and thus the rules are less helpful

Knight Joust into Keepaway

Param	Initial Performance	Asymptotic Performance	Accumulated Reward
Without Transfer			
	7.8 ± 0.1	21.6 ± 0.8	756.7 ± 21.8
Extra Action			
100	13.8 ± 1.1	21.8 ± 1.2	758.5 ± 29.3
250	13.5 ± 0.9	21.6 ± 0.9	747.9 ± 25.3

Table 6. Transferring from Knight Joust to Keepaway significantly improves the initial performance, but the other two metrics are not improved. All results are averaged over 20 independent trials and the standard error is shown. Numbers in **bold** are statistically different from learning without transfer at the 95% level, as determined via a Student’s t-test.

after learning in the target task. In informal experiments, after 40 hours of training in Keepaway, agents that transferred advice from Ringworld were following the advice for 90% of the actions, while agents that transferred advice from Knight Joust were following the advice for only 85% of the actions, indicating that the Ringworld advice was more useful in Keepaway.

6. Learning a Rule Translation Function

Thus far, this work has relied on hand-coded cross-domain mappings δ_A and δ_X , as has much of the past transfer learning research. Learning cross-domain mappings in their entirety is an open and very challenging research problem, and is beyond the scope of this paper. In this section, we sketch a method by which a cross-domain mapping from Ringworld can be partially learned starting from some knowledge about the qualitative characteristics of the source domain. This method increases the degree of automation of Rule Transfer by removing the requirement for some given knowledge, though at the expense of requiring different, arguably more intuitive, information. Although this section focuses on using Ringworld as a source task, a similar process could be followed for Knight Joust.

An analysis of Ringworld results in observations A-C, and we make assumption D for feasibility:

- A An opponent approaches the player, as shown by a *distance to opponent* state variable, eventually ending the episode when the player fails to move.
- B As the player moves, the opponent will continue moving toward the player.
- C The action *RunNear* takes longer to complete than the action *RunFar*.
- D The learner has a sense of “identity” and can distinguish between other objects in the world. We accomplish this by assigning unique identifiers (UIDs) to all objects in a task. The agent initially knows only its own UID and is able to determine the UIDs that are used to compute each state variable.

Given these observations, we construct the following procedure under the assumption that this process could map Ringworld to multiple target tasks, and that constructing such a procedure is simpler than creating a mapping for a given target (or impossible if the target is unknown beforehand). Players in the target (3 vs. 2 Keepaway) first record $(S, A, time)$ tuples (for 60 episodes or roughly 7 simulator minutes). They then determine the mapping by the following procedure:

1. **Identify the state variable(s) that are near zero when the episode ends.** Observation A tells us that the Ringworld state variable $dist(P, O)$ should map to this state variable. We find that, on average, $dist(K_1, T_1) = 0.48 \pm 0.2$ and $dist(K_1, T_2) = 0.94 \pm 0.4$ when the episode ends after *Hold* (all other state variables are at least an order of magnitude larger, on average).

2. **Identify the action that causes the target task variable(s) from step 1 to most consistently decrease.** Also by observation A, we find that *Stay* should be mapped to *Hold* in Keepaway because both $dist(K_1, T_1)$ and $dist(K_1, T_2)$ decrease during this action on average with a small variance, while this is not true for the actions *Pass₁* and *Pass₂*.
3. **Identify the state variables that correspond to $dist(P, Target_1)$ and $dist(P, Target_2)$ and the actions that correspond to *RunNear* and *RunFar*.**
 - (a) Step 1 identified the UIDs of the two opponents. Using assumption D, we search all Keepaway state variables and consider only those that include the player’s UID but not the opponents’ UIDs: $dist(T_1, C)$, $dist(T_2, C)$, $dist(K_2, T)$, and $\text{Min}(dist(K_3, T_1), dist(K_3, T_2))$.
 - (b) Next, consider the state before an action (either *Pass₁* or *Pass₂*) is executed. Before the action, some state variable will be consistently greater than the value of $dist(P, O)$ after the action (Observation B). For Keepaway, $dist(K_2, T)$ is greater than $dist(P, O)$ after *Pass₁* (the distance decreases by 6.8m on average) and $\text{Min}(dist(K_3, T_1), dist(K_3, T_2))$ is greater than $dist(P, O)$ after taking the action *Pass₂* (an average decrease of 10.8m).
 - (c) Observation C allows us to sort the two pass actions by how long they take: the action *Pass₁* takes an average of 7.4 time steps while *Pass₂* takes an average of 9.5 time steps. This allows us to decide that *RunNear* should map to *Pass₁* and *RunFar* should map to *Pass₂*. Sorting out the pass actions also allows us to map $dist(P, Target_1)$ to $dist(K_2, T)$ and $dist(P, Target_2)$ to $\text{Min}(dist(K_3, T_1), dist(K_3, T_2))$.
4. **Identify $dist(P, Target_1)$ and $dist(P, Target_2)$.** Step 3 identified the UIDs of *Target₁* and *Target₂* and the UID of the player is known (assumption D). In Keepaway, the UIDs of *Target₁* and the player identify $dist(P, Target_1)$ as $dist(K_1, K_2)$ and the UIDs of *Target₂* and the player identify $dist(P, Target_2)$ as $dist(K_1, K_3)$.
5. **Identify $ang(O, P, Target_1)$ and $ang(O, P, Target_2)$.** Similar to Step 4, using the known UIDs of the player, the enemies, and the run targets, we identify $ang(O, P, Target_1)$ as $ang(T, K_1, K_2)$ and $ang(O, P, Target_2)$ as $ang(T, K_1, K_3)$.

Thus, using source task observations, we are able to map the source task onto Keepaway, using recorded data to construct a cross-domain mapping very similar to the hand-coded mapping from Table 2. The only difference is that

the learned cross-domain mapping maps $dist(P, O)$ onto both $dist(K_1, T_1)$ and $dist(K_1, T_2)$, but since the takers' policy in 3 vs. 2 Keepaway causes them to stay very close together, in practice these two mappings are equivalent.

We emphasize that our choice of observations describing the source task are as important as the method for using them. If, for example, one of the observations were that $dist(P, Target_1)$ did not ever change when taking the *Stay* action, $dist(P, Target_1)$ would not map to any state variable in Keepaway because all state variables change over time, regardless of action, due to noise in the sensors. However, in some cases, such as this one, we would be able to detect that the source task observations were violated. In these cases, it would be possible to consider a different set of observations for the source task, consider a different source task for transfer, or simply learn Keepaway without transfer. Thus we believe this method provides a reasonable solution for solving the learned mapping problem when the source task is well understood but the target is unknown.

7. Related Work

While a number of methods can successfully transfer between pairs of tasks in the same domain, the main novelty of this work is to show that *inter-domain* transfer is not only feasible, but beneficial. Additionally, we extend previous work on using advice to improve performance by comparing multiple rule utilization strategies. Most similar to our Value Bonus method is work by Kuhlmann et al. (2004) which gives a bonus to actions which are suggested by hand-coded rules. Other work (Madden & Howley, 2004) has shown that learned rules can be used to initialize Q-values when visiting a novel state, but such a method is not directly applicable in continuous domains. Learned advice has also been successfully used as soft constraints to help initialize a target task's function approximator off-line before learning (Torrey et al., 2006).

Other work has focused on learning task relationships. For instance, some work has used homomorphisms (Soni & Singh, 2006) to generate and empirically test a number of possible relationships. If a human can define both tasks can be defined in terms of *qualitative dynamic Bayes networks*, Liu and Stone (2006) showed that a graph matching method can automatically find task similarities.

8. Conclusion and Future Work

In this work we have introduced Rule Transfer along with three different advice utilization methods. Using three transfer metrics, Rule Transfer significantly improved learning in robot soccer after first learning in a gridworld task. In addition to hand-coding an cross-domain mapping function, we give evidence such a mapping may be learned from observed task data.

The Ringworld task was constructed directly from data

gathered in the target task while the Knight Joust task was chosen as intuitively related to Keepaway. In future work, rather than constructing such tasks by hand, we would like to automatically construct such source tasks.

The cross-domain transfer experiments in this paper begin to demonstrate the flexibility of rule transfer; agents were able to transfer knowledge successfully irrespective of the underlying function approximator's representation. In the future we would like to further exploit this flexibility by transferring between agents with more dissimilar internal representations and learning methods. Finally, if both the source and target task had a finite number of states, the policies could be transferred directly. We plan to test this and determine the effect of using rules as an intermediary when they are not required for transfer.

Acknowledgments

We would like to thank Raymond Mooney, Cynthia Matuszek, Lilyana Mihalkova, Nick Jong, and the anonymous reviewers for helpful comments and suggestions. This research was supported in part by DARPA grant HR0011-04-1-0035, NSF CAREER award IIS-0237699, and NSF award EIA-0303609.

References

- Cohen, W. W. (1995). Fast effective rule induction. *International Conf. on Machine Learning* (pp. 115–123).
- Kuhlmann, G., Stone, P., Mooney, R., & Shavlik, J. (2004). Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer. *The AAAI-2004 Workshop on Supervisory Control of Learning and Adaptive Systems*.
- Liu, Y., & Stone, P. (2006). Value-function-based transfer for reinforcement learning using structure mapping. *Proc. of the 21st National Conference on Artificial Intelligence*.
- Madden, M. G., & Howley, T. (2004). Transfer of experience between reinforcement learning environments with progressive difficulty. *Artif. Intell. Rev.*, 21, 375–398.
- Rummery, G. A., & Niranjan, M. (1994). *On-line Q-learning using connectionist systems* Technical Report CUED/F-INFENG-RT 116. Engineering Dept., Cambridge University.
- Singh, S. P., & Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22, 123–158.
- Soni, V., & Singh, S. (2006). Using homomorphisms to transfer options across continuous reinforcement learning domains. *Proc. of the 21st National Conference on Artificial Intelligence*.
- Srinivasan, A. (2001). The aleph manual.
- Stone, P., Kuhlmann, G., Taylor, M. E., & Liu, Y. (2006). Keepaway soccer: From machine learning testbed to benchmark. In I. Noda, A. Jacoff, A. Bredenfeld and Y. Takahashi (Eds.), *RoboCup-2005: Robot soccer world cup IX*, vol. 4020, 93–105. Berlin: Springer Verlag.
- Sutton, R. S., & Barto, A. G. (1998). *Introduction to reinforcement learning*. MIT Press.
- Torrey, L., Shavlik, J. W., Walker, T., & Maclin, R. (2006). Skill acquisition via transfer learning and advice taking. *ECML* (pp. 425–436). Springer.
- Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Grumpy & Pinocchio: Answering Human-Agent Negotiation Questions through Realistic Agent Design

Johnathan Mell

University of Southern California
12015 Waterfront Drive
Los Angeles, CA, USA
mell@ict.usc.edu

Jonathan Gratch

USC Institute for Creative Technologies
12015 Waterfront Drive
Los Angeles, CA, USA
gratch@ict.usc.edu

ABSTRACT

We present the Interactive Arbitration Guide Online (IAGO) platform, a tool for designing human-aware agents for use in negotiation. Current state-of-the-art research platforms are ideally suited for agent-agent interaction. While helpful, these often fail to address the reality of human negotiation, which involves irrational actors, natural language, and deception. To illustrate the strengths of the IAGO platform, the authors describe four agents which are designed to showcase the key design features of the system. We go on to show how these agents might be used to answer core questions in human-centered computing, by reproducing classical human-human negotiation results in a 2x2 human-agent study. The study presents results largely in line with expectations of human-human negotiation outcomes, and helps to demonstrate the validity and usefulness of the IAGO platform.

General Terms

Experimentation; Human Factors.

Keywords

Virtual Humans; Human-Agent Competition; Negotiation; IAGO

1. INTRODUCTION

Negotiation is the focus of a great deal of research, both within the traditional business and conflict resolution literatures, and (more recently) in artificial intelligence. Negotiation—in the computational sense—tends to be researched in one of two broad paths. Agent-agent negotiation focuses on distributed problem solving and computational efficiency, as perfectly rational agents can quickly exchange thousands of offers in order to solve large problems. Human-agent negotiation offers separate challenges, as all agent designs must be subject to empirical evaluation and testing in the field. Human-agent negotiation also tends to be much slower than agent-agent negotiation, and may involve additional channels of communication—emotional exchange, free chat, and preference statements in addition to offer exchanges. All of these features are necessary for a human-agent system that attempts to simulate the often free-wheeling style of interaction that characterizes human-human negotiation.

Appears in: *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017), S. Das, E. Durfee, K. Larson, M. Winikoff (eds.), May 8–12, 2017, São Paulo, Brazil.*

Copyright © 2017, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

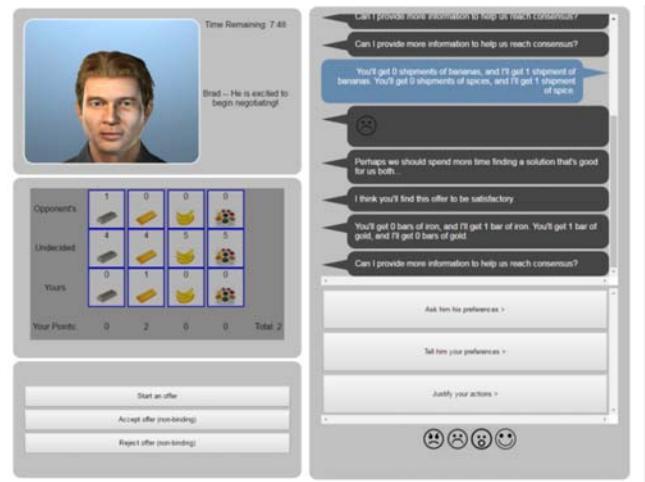


Figure 1: Agent Running on IAGO

This new class of human-aware agents has drawn recent interest, since they can be used as mediators or conflict resolvers, or can be used as pedagogical tools to teach negotiation skills [3]. The latter is of particular use, since teaching negotiation skills is often arduous and expensive¹, relegating it to those that have the time or funds to attend business training courses or employ the use of a personal trainer. Obviously, agents have several benefits as teachers, as they can be both perfectly patient and readily available. But while it is true that current agents cannot substitute perfectly for a human, this work aims to build on previous attempts to bring human negotiation techniques into the virtual world [5], and show that indeed, these techniques can be just as effective with an AI partner as with a flesh and blood human.

The current paper evaluates the effectiveness of the Interactive Arbitration Guide Online (IAGO) platform for designing agents that interact primarily with humans. We describe IAGO from a high level, and discuss the principles by which agents may be designed for it. To that end, we illustrate the usefulness of the IAGO platform by stepping through the construction of four different computational agents that function on its framework. These agents are able to negotiate with humans through the most notable channels of human negotiation—namely, offer exchange, emotional expression, and natural language/preference sharing. These latter two channels are novel for a human-agent platform, and we demonstrate the full use of these channels in an empirical context.

¹ As an example, at the time of this document's preparation, Wharton Business School offered a 5-day negotiation seminar at the low cost of \$11,000. See executiveeducation.wharton.upenn.edu.

IAGO, and the agents that run on it, employ strategies for generating and responding to offers, expressing and reacting to emotion, and revealing critical information about preference in a multi-issue bargaining task negotiation scenario. By constructing a sample study that pairs these agents against human participants recruited from Amazon’s Mechanical Turk (MTurk) service, we can generate results in a human-agent context that are comparable to human-human results. These results show the implications for future agent design using the IAGO framework, and the experimental benefits of conducting human-agent interactions in an online context, since the results are similar to what would be suggested by the human-human negotiation literature. By showing IAGO-designed agents performing an a human-agent study, we show that they should be able to perform at a similar or higher level to humans in negotiation games, based on real-world data collected from human-computer negotiation sessions.

2. BACKGROUND

Negotiation, whether it be between two humans, a human and an agent, groups of agents, or ever-more esoteric combinations, is a research topic that spans myriad scientific domains. The human-agent case in particular is a relatively new direction, and requires tools to promote its investigation. Platforms must be developed upon which agents that interact with humans can be designed, and real-world data must be collected and reviewed regarding the interaction of humans with these new agents.

One classical option for investigating human-agent negotiation takes the form of the multi-issue bargaining task, which is considered a de facto standard problem for research into social cognition and interpersonal skill training [26]. In the multi-issue bargaining task, two participants work to determine how to split varying issues, each with hidden values to each side. The task may involve distinct phases, where first information about preferences is exchanged, and then a series of offers are made. The task is also often characterized by time pressure, which is often modeled as a decaying utility function. Even with a small number of issues, the task can quickly become a challenge for agents to simulate, especially those which aim to act as partners for humans in such a negotiation in real time, and numerous works attempt to address the multi-issue bargaining task [8],[9],[14],[23]. While this makes the multi-issue bargaining task a difficult challenge computationally, adding a human actor complicates issues even further, since humans often behave “irrationally” in game theoretic contexts.

Many negotiation research foci attempt to simplify the problem by making protocols that strongly limit what information can be exchanged. They often model information exchange as a costly endeavor by which every instance of interchange is modeled by a set “price” that reduces endgame utility values, (more commonly) refuse to allow information exchange at all, instead preferring to model opponent preferences using stochastic processes [2]. Other attempts require offers to alternate from one side or another, or specify that only full offers, wherein no items are left undecided, can be exchanged. While these solutions allow for progress in limited human-agent contexts, and certainly have their benefits in agent-agent negotiation, they hardly resemble the freeform nature of actual human-human negotiation.

Therefore, our work is motivated by an attempt to design agents that can practically negotiate with humans. Agents, like humans, should make use of similar channels of communication, such as emotional exchange, preference utterances, and partial offer exchange. These agents should use human techniques, like the exchange of informal favors [17], or the use of anger in negotia-

tion to secure value [6]. Ideally, agents should be able to build trust over time with repeated negotiations, and should recognize past betrayals and alliances. These features are key to solving age-old negotiation challenges, such as what Kelley calls the “dilemma of trust” and the “dilemma of honesty” [13][27] However, there exists no platform upon which these challenges may be readily explored (to the authors’ knowledge) in the human-agent interaction context.

To wit, the dilemma of honesty refers to the idea that true information about oneself, whether that be preferences in a negotiation, or how much one loses if the negotiation falls through, is very valuable to keep secret. Even without considering the possibility of lying about said information, which may lead to long term harms to trust, there is still much to be said about when and how much information should be shared. Therefore, any platform which attempts to address this dilemma should have a robust method for exchanging preferences and other valuable pieces of information. Ideally, this should resemble human-human negotiation as much as is possible. This includes providing multiple natural language ways to express the same logical fact: e.g., “I like the apples better than the oranges”, versus the equivalent “I like oranges less than apples”, or the similar but slightly more informative “I like apples best”, which IAGO attempts to address.

The dilemma of trust is equally important, as it requires agents be able to judge the truth of statements they receive. To understand the dilemma of trust as it applies to negotiation, a good understanding of how and when humans lie is required. Any platform that would attempt to address this thorny issue should be able to provide a detailed history of past statements and questions, as well as bargaining history and other details. While a worthy subject, it is not the thrust of the sample agents in this research, which take information at face-value [15][18].

Once the agents have been designed based on these empirical observations, they must also be tested in the field against actual humans. While humans often treat agents differently than their human counterparts or even human-controlled avatars (agents are often subject to outgroup effects) [4],[10], virtual agents that exhibit human-like features such as emotion or natural language are often treated in a near-human way. To that end, a platform for designing agents and hosting negotiations between them and humans must needs have the ability to manipulate channels of communication used by humans.

Previous efforts to allow for effective human-agent negotiation include the multi-issue bargaining task game, Colored Trails [11],[20], its web-based cousin WebCT [17], as well more natural-language focused approaches such as NegoChat [22]. However, these platforms tend to focus on a single channel of communication, such as the exchange of formal offers or natural language messages. None of them include an emotional channel wherein deliberate information about a player’s emotional state can be exchanged. For these reasons, we present the IAGO platform, which has multiple channels of communication and is designed specifically to be deployed for human-agent interaction over the web. Using this framework, it is hoped that agents can be designed that will answer the questions of human behavior and interaction with agents in a negotiation context.

3. SYSTEM DESIGN

3.1 IAGO Platform

To describe the design behind our agents, it is important to understand the basic guiding principles behind IAGO, the online plat-

form on which they run. IAGO boasts a number of design principles that make it suitable for human-agent negotiation. These design principles are:

1. It must support current web-standards and require little to no installation of complex support software on a user's machine.
2. It must deploy a well-defined API that allows both agent designers and negotiation game designers to easily create and specify behaviors for the purposes of competition/research.
3. It must support currently unexamined aspects of human-human negotiation in a human-agent context. Specifically, this must include partial offers, visual representation of emotional signals, and relative preference elicitation/revelation. [16]

The design of IAGO is such that it can be used by a human participant through a web browser. Actions taken by the user, such as crafting an offer to send to the agent, or commenting on the quality of previous deals, are sent via an HTML5 GUI through a WebSocket and onto the agent code, which is hosted as a Java Web-Servlet on any Tomcat 7 or newer server. This structure allows any participant to simply be given a URL to a running IAGO instance, and requires no installation on any client machine. Furthermore, as an added benefit, the agent designer wishing to build IAGO instances can do so in a cross-platform manner, requiring only a single .jar file and a Tomcat installation to begin work.

The second and third design principles are encapsulated by the Event system used in IAGO. While extensive description of each of the functions available in the API is impossible herein, IAGO can generally be described as allowing for rule-based agent design in reaction to a set of distinct events:

1. SEND_MESSAGE
2. SEND_OFFER
3. SEND_EXPRESSION
4. TIME
5. OFFER_IN_PROGRESS
6. FORMAL_ACCEPT

From there the agent designer makes decisions on how to react to the event. For example, upon receiving a SEND_EXPRESSION event with content indicating that the player was expressing sadness, the agent could decide to adopt a shocked expression itself, and then create a new counter-offer a few seconds later.

While agents are able to manifest the emotion channel through the SEND_EXPRESSION event, they are similarly able to interact using offers and natural language messages using the SEND_OFFER and SEND_MESSAGE events, respectively. It is important to highlight a particular class of message utterances subsumed under the SEND_MESSAGE Event. These utterances take the form of comparing the point values of one or two items. Example utterances for this game included "I like the bars of iron more than the shipments of bananas." or "Do you like the shipments of spices best?" Preference utterances could use any of 5 relational operators: greater than, less than, equally, best, or least. Furthermore, utterances could be either queries or statements, allowing for a total of $2 * 5 = 10$ types of preference utterances. These preference utterances are often considered to be "valuable" information, as they reveal some information about the point values of the opponent, and are an important part of designing the information exchange policies of an agent.

Agents have full control over the timing of their actions through use of the TIME event—for example, agent designers may schedule events to occur only after a specified number of seconds have passed. Whereas an agent-agent system would be limited only by the bandwidth and latency of communication between the two

partners, IAGO agent designers must be aware of the physical and mental limitations of their human partners. Humans are not capable of processing dozens of offers per second, and tend to read data from multiple channels simultaneously. It may prove more effective to program an agent to smile for a few seconds, then wait before sending an offer and a comforting message. Indeed, IAGO negotiations can be characterized by the usage of their idle periods nearly as much as by the Eventful sections.

The final two Events as listed above bear brief mention. OFFER_IN_PROGRESS is used as a cue that the human or agent player is considering sending an offer but has not done so yet. An agent designer can use this to avoid overwhelming a human player, or (conversely) to interrupt them in advance of receiving an expected poor offer. Visually, the human views the agent version of this event as a flashing ellipsis in the chat menu, like many instant messaging programs. Secondly, the FORMAL_ACCEPT Event is used to finalize the distribution of the task items and end the negotiation. More notably, there is no "casual accept" event, since IAGO is designed to mimic human negotiations, where previously agreed-upon terms may often be retracted or modified with no formal penalty.

3.2 Agent Design

Agents designed for IAGO implement several policies to categorize their response to different events. Ideally, these policies should work together to determine the full behavior of an agent throughout the entire negotiation. Often, there is substantial overlap, as even an event as simple as sending an offer may involve natural language, offer evaluation, and emotional reaction in a single response. As such, these division are recommended, but not enforced, when deciding to create agents.

3.2.1 Offer Exchange

BehaviorPolicies determine the type of offers that agents will accept and craft to send to their human partner. Although "acceptances" and "rejections" of offers are allowed by either party, the IAGO framework does not enforce these in any way. Agent developers may choose to adhere to previous agreements within a negotiation if they choose, but only the final, fully-distributed full offer is locked in (accepting this "formal offer" ends the game). BehaviorPolicies are perhaps the most comprehensive policies supported by IAGO since they tend to define both incoming and outgoing offers.

3.2.2 Information Exchange

MessagePolicies determine the language agents use. This can be in reaction to the set of pre-selected chat utterances or any other event. Commonly, both the BehaviorPolicy and the MessagePolicy are invoked when the player sends an offer, as the agent must decide if it wants to accept, reject, or ignore the offer, as well as what it should say (e.g. "Yes, that offer sounds good to me!").

3.2.3 Emotion Exchange

Finally, the ExpressionPolicy determines what emotions are shown by the agent. Emotions are sent in two ways. First, the portrait of the agent will change—for example, to display "happy", the agent will show a smiling version of its avatar. Second, an emoticon is sent through the chat that expresses the selected emotion. It is important to distinguish that "emotions" are not literally sent, but rather "expressions of emotions". There is no automatic detection of user emotions, nor is the agent designer under any compunction to show emotions that realistically correspond to the simulated mental state of the agent (or to show emo-

tions at all, for that matter). However, this channel does allow deliberate expressions of emotions to be sent, and this information is often valuable to either party in a negotiation.

3.3 Game Customization & Setup

Each IAGO game environment is configurable with a number of options. These options range from the essential, such as the type of game being played (multi-issue bargaining, ultimatum game, etc.), to the more esoteric, such as whether or not the on-screen timer is visible to the user. Further options allow the number of issues to be customized (normally between 1 and 5), the levels of each issue to be set, point payouts to be settled for each player, and visual representations of the items to be loaded and displayed. Finally, the pre-set natural language messages that the user can express are also set during the game customization phase.

The gamespace we used for our agents as a demonstration was configured to be a 4-issue multi-issue bargaining task, with each issue having 6 levels (5 items). Each item was assigned a point value between 1 and 4, inclusive. All point accruals were linear, meaning that gaining 1 of the 4-point item was worth 4 points, 2 was worth 8-points, etc. The items were given images and descriptions that cast the game as a “Resource Exchange Game”. These items were “bars of gold”, “bars of iron”, “shipments of bananas”, and “shipments of spices”. The two players took on the role of negotiators determining how to split the items between them.

The human player was assigned 4 points for each shipment of spices he or she acquired, 3 points for each shipment of bananas, 2 points for each bar of gold, and 1 point for each bar of iron. The agent player was assigned 4 points for each bar of iron, 3 points for each bar of gold, 2 points for each shipment of bananas, and 1 point for each shipment of spice. In this way, the game was set up to have “integrative potential” – if each player got their top items, the maximum joint value earned would be 70, whereas if each player only got their least important items, the maximum joint value earned would only be 30. These values are summarized in Table 1.

The game was set to a timed length of 10 minutes. Participants were warned at the 1-minute-remaining mark of their remaining time. If time expired with no agreement being reached, then participants were awarded their Best Alternative To Negotiated Agreement (BATNA). Both the human and the agent had a BATNA of 4 points, of which they were made aware before the game (players only knew their own BATNA, not their opponent’s).

Table 1. Item Payoffs

	Agent Player	Human Player
Shipments of Spices	1	4
Shipments of Bananas	2	3
Bars of Gold	3	2
Bars of Iron	4	1

4. AGENT DESIGN

4.1 Shared Agent Policies

To successfully design experiments wherein agents negotiate with humans, it is important that any experimental manipulations be well-understood and contained. Unfortunately, the nature of negotiation makes this particularly challenging, as agent functioning is depending highly upon the actions of the user. Fortunately, by constructing Policies and sharing them between agents, a clear experimental design can be achieved. Customization of offer, information, and emotion exchange allows for a very wide space of agents to be designed. In this paper, we fix much of the behavior and define variability in two of those dimensions (information exchange and emotional language) to illustrate how to examine different techniques commonly examined in the human-human literature. Thus, we designed four agents, which share several aspects of their Policies in common, and which are intended to showcase several aspects of the IAGO platform. These agents, named “Pinocchio”, “Grumpy”, “Rumble”, and “Merlin”, were designed specifically to experimentally test differences in tone (“nice” vs. “nasty” agents) and preference revelation strategy (“strategic” vs. “free” agents), but outside of these differences function identically. The agents are summarized in Table 2, and their differences are detailed in sections 4.2 and 4.3. The shared elements of the agents’ policies are described below, in sections 4.1.1 – 4.1.3.

4.1.1 Behavior Policies

The agents designed are intended to make and accept offers that are both largely fair and consistent between agents. The parts of the BehaviorPolicy that defined how offers are proposed was identical between all example agents created. Offers often differed during each negotiation, since the offers are dictated by player choice, the amount of information revealed by the player, and other factors. However, all policies were identical across agents.

Agents will propose offers to the human player in one of two scenarios. First, if the player proposes an offer the agent wishes to reject, the agent will reject it and then, after a short waiting period, craft a counter-offer. Secondly, the agent will oblige in crafting an offer if the player asks it to do so in chat, using the “Why don’t you make an offer” utterance. Agents create offers using the Minimax Preference Algorithm (see below) to determine the human player’s preference ordering. Then, they attempt to make offers that progressively allocate one item from the agent’s and human’s top choices. In our example, if the agent supposes through the Minimax Algorithm that the human prefers gold, it will attempt to make a deal that gives the human one additional gold while the agent gets one additional iron (the agent’s preferred item). If the agent believes the human wants the same item it does, it will attempt to split the remaining balance fairly.

Table 2. Agent Matrix

	Strategic	Free
Nice	Merlin	Pinocchio
Nasty	Rumble	Grumpy

When receiving an offer, agents check if the offer is both “locally fair” as well as “globally fair”. Local fairness refers to the offers itself being fair, while global fairness refers to the current state of the board (taking into account all offers so far) being fair. Again, the agent determines human preference the Minimax Preference Algorithm. Then, it determines if the currently proposed offer would boost the human more than it would boost the agent. The agent must have >0 positive benefit, and there is a window equal to the number of issues wherein the agent would consider the offer “locally fair”. In our example of a 4-issue game, the agent would consider an offer that increased its points by 7 and the human’s points by 10 to be “locally fair” as $10 - 7 < 4$. To determine global fairness, the agent follows the same procedure but instead looks at the entire offer board as it stands based on prior acceptances. If the offer is considered fair on both counts, it is accepted. Otherwise, it is rejected, although agents do have unique dialogue for if it is considered locally fair but not globally fair.

4.1.2 Message Policies

The agents all attempt to gain information about their partner’s preferences in the form of relational utterances. This can take the form of occasionally asking direct questions about preferences, or reconfirming information already gathered. For example, all agents respond to one user utterance by reiterating: “Your favorite item is ___, right?” assuming the favorite item has been determined by this point (at which point the blank would be filled in by a description of the item).

One core principle of all the four agents is that they never lie, and further, always assume that their partner is telling them the truth. Although the value and ethical complexities of lying in negotiation are well established [1],[12],[25], these first designs are more straightforward in their approach to information. If, at any point, the agents determine that the information given to it by the player is somehow contradictory (for example, if a player claims both that an item is their most valuable, but also that it is valued less than another item), the agent will reconcile its history of statements and point out the discrepancy to the player. All agents use the Consistency Algorithm to do this (see below), although they differ in the tone of the messages associated with it.

4.1.3 Expression Policies

The expression policies have little in common between the nice and nasty agents. However, they do share the same basic timing. When the agent receives a negative or aggressive statement from the player, such as “Your offer sucks!” they will respond with

some sort of emotional response. Similarly, the agents respond to positive statements. Finally, the agents also respond based on the trend of the offers received from the human player; if the offers have been getting better, the agents react one way, but if the offers have been getting worse, the agents react differently.

4.1.4 Consistency Algorithm

The algorithm used to check for consistency in preference statements is fairly straightforward. Whenever a new preference statement is uttered by the human player, all agents log that statement in an ordered queue. Then the agent attempts to reconcile per the following procedure:

1. Start with the list of all possible permutations of value orderings. In a 3-issue game, for example, this list would be [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], and [3, 2, 1].
2. For each preference statement, eliminate contradictory orderings.
3. If there are no orderings left, see if dropping the oldest preference in the queue would create orderings.
4. Continue until the end of the queue is reached.
 - a. As soon as one is found, notify the player which preference statement was dropped.
 - b. End the iteration.
5. If only removing the most recent preference statement would rectify the orderings, then drop the entirety of all preference history and notify the player.

4.1.5 Minimax Preference Algorithm

This algorithm makes use of the results of the Consistency Algorithm above. After running the Consistency Algorithm, the agent checks the remaining valid orderings. For example, if the potential human orderings are (1 being the top choice, 4 being the last choice):

$$A: \{4, 3, 2, 1\}, B: \{3, 2, 4, 1\}, C: \{4, 1, 3, 2\}$$

It will determine which one is worth the most points to itself and assume that to be the true ordering until corrected. For example, if the agent prefers 1 best, it will most likely pick ordering A or B due to 1 being worth the least to the player. The agent will assume this is the true human ordering until a new preference statement is revealed, at which point the algorithm must be rerun. In this way, the agents behave “optimistically”, in that they assume, given equally likely unknown distributions, the correct distribution is the one that will end up favoring them the best.

Table 3. Nice vs. Nasty Language (Non-comprehensive)

Event	Nice Language	Nasty Language
Agent rejects offer	I’m sorry, but I don’t think that offer is fair to me.	That’s not fair.
User says “It is important that we are both happy with an agreement.”	I agree! What is your favorite item?	I suppose, if you want to be all ‘flowers and sunshine’ about it. What item do you want the most?
User says “Why don’t you make an offer?”	Sure! Let’s see how this sounds...	Thought you’d never ask...
User says “This is the very best offer possible.”	Ok, I understand. I do wish we could come up with something that is a more even split though.	Oh really? That’s pretty sad. I think you could do better.
User sends an “angry” emoticon	I’m sorry, have I done something to upset you?	What’s wrong?
User does nothing for several seconds	Can I provide more information for us to reach consensus?	Are you even still there?

4.2 Agent Conversational Tone

(Nice vs. Nasty)

In creating the agents, determining how they will respond through text to all of the potential events that can occur is of utmost importance. As there is no restriction on the “script” of the agents, authors of agents are given wide latitude in deciding the proper wording. This approach allows the benefits of expert input (from writers, for example), while still allowing agents to respond to various classes of events without enumerating a ballooning number of potential scenarios. Of course, automatic approaches to dialogue writing are possible as well.

For these agents, the differences in tone between the agents are myriad, but are restricted to the language that the agents use throughout the game, and thus, their respective MessagePolicies. The key points of distinction are summarized here. “Nice” agents include the “Merlin” agent and the “Pinocchio” agent, while the “Nasty” agents are represented by “Grumpy” and “Rumble”. Although the textual differences between the nice and nasty agents are broad, it was attempted that no informational content differs between them. For example, the nice agents will reject an offer with language like “I’m sorry, but I don’t think that offer is fair to me,” while the nasty agents will say “That’s not fair.” A sampling of the differences in language is found in Table 3.

Nice and nasty agents also differed in their ExpressionPolicy. When nice agents received poor offers from their opponent, they expressed sadness, whereas the nasty agents expressed anger.

When good offers were received, nice agents smiled, while nasty agents expressed no emotion. Additionally, many of the utterances that the player could say would result in an emotional expression from the agent. Following the same pattern, nice agents smiled or showed sadness, while nasty agents showed nothing or anger, respectively.

4.3 Information Revelation Strategy

(Strategic vs. Free)

The “Rumble” and “Merlin” agents follow a strategic information revelation strategy, while the “Pinocchio” and “Grumpy” agents follow a free revelation strategy. When preference queries are made by the human player, the strategic agents will refuse to reveal information about their preferences. This design follows a general principle of human negotiation—since information about a player’s preferences can give the opponent an advantage over them by allowing them to mislead you. Instead, the strategic agents follow a “tit-for-tat” strategy, where they will reveal information that mirrors the information they receive. This debt is always paid back immediately, so if a human player reveals their top item, the strategic agents will (truthfully) reveal theirs as well.

Table 4. Agent Policies

	BehaviorPolicy	ExpressionPolicy	MessagePolicy
Pinocchio	NiceBehaviorPolicy	NiceExpressionPolicy	NiceFreeMessagePolicy
Grumpy	NastyBehaviorPolicy	NastyExpressionPolicy	NastyFreeMessagePolicy
Rumble	NastyBehaviorPolicy	NastyExpressionPolicy	NastyStrategicMessagePolicy
Merlin	NiceBehaviorPolicy	NiceExpressionPolicy	NiceStrategicMessagePolicy

The free agents are designed under the assumption that revealing the information too early is not a large advantage, but can generate rapport or goodwill that will allow the negotiation to proceed more smoothly. They will also follow the tit-for-tat strategy that the strategic agents follow, but will additionally simply respond to direct questions (e.g., “Do you like bars of iron best?”).

4.4 Agent Summary

In summation, the agents use a collective total of 8 differing Policies across 4 agents. There is a NiceExpressionPolicy and a NastyExpressionPolicy, as well as a NiceBehaviorPolicy and a NastyBehaviorPolicy. There are 4 MessagePolicies, as the agent behavior for the experimental design overlaps. Thus, we have a NiceStrategicMessagePolicy, NastyStrategicMessagePolicy, NiceFreeMessagePolicy, and finally NastyFreeMessagePolicy. While the overlapping nature of the messaging makes 4 policies necessary, it is important to note that the policies are merely guidelines to encapsulate thematic regions of the program. In other 2x2 experiments designed to run on IAGO, such divisions may not be necessary. The agents themselves simply load the proper policies in order to distinguish themselves in the experimental game. The agents choose policies according to Table 4.

5. EXPERIMENTAL DESIGN

By designing the four agents per the policies described in Table 2, a 2x2 matrix design of a potential study follows directly. We design an experiment that takes two factors traditionally left unexamined by agent-agent negotiation research (namely, use of emotional language and information exchange) and examine them to see if they yield similar results to human-human studies. Since IAGO supports these factors intrinsically, and IAGO agents are intended to be prototyped and tested quickly, we can use these agents to run a rapid human-agent study using Amazon’s Mechanical Turk (MTurk) service. By utilizing these agents as partners for human players, we can demonstrate that IAGO is functional as a platform by replicating behaviors found in human-human negotiation.

To that end, we will answer the following research questions:

Q1: In human negotiation, information about preferences is considered valuable. If the agents and platform perform like a human would, we would expect that the strategy for revealing preferences will have some effect on human behavior. Does strategically revealing preferences encourage players to reveal information about their own preferences? Is this effect mitigated by the emotional language used?

Q2: Strategy in revealing preferences may have some effect on messages and preferences exchanged, per Q1. But how will it affect the joint value discovered by the human and the agent? In human negotiations, if both parties understand each others’ true preferences, they are more likely to “grow the pie” and find additional joint value in integrative situations. However, if the strategy is seen as a refusal to compromise, or simply an aggressive move, then the opposite may occur, and joint value might be lost.

Q3: Next, we can examine the effect of use of aggressive, nasty language and emotions on a negotiation. Previous literature indicates that aggressive behavior will often cause the opponent to concede. So therefore, we might suppose that nasty agents will have a greater lead in points over the player than nice agents. What effects, if any, does emotional expression have on who “wins” the negotiation?

We recruited one hundred and ninety-six participants using MTurk as subjects for our sample study. All participants marked an online consent form detailing the study, which contained a 5-minute demographic survey portion and an approximately 15-minute interactive game session with one of the 4 virtual IAGO agents. The participants were all over 18 years old, and were residents of the U.S. and native English speakers. Non-U.S. participants were excluded to minimize cultural effects. Participants were kept anonymous through their MTurk-assigned unique ID number.

After recruitment, all subjects were presented with a survey that recorded basic demographic information and a few standard behavioral measures. Participants then read through a visual tutorial and were asked several attention/verification questions to ensure they understood the game. Subjects that successfully answered the questions were then randomly assigned to one of the 4 agents.

Each agent introduced itself as a male Artificial Intelligence named "Brad". A computer-generated image (see Figure 1) of a male face was part of the setup. Each participant was then free to interact with the agent by sending and receiving offers, messages, preferences, and emotional expressions. At the termination of 10 minutes, or once both players had formally accepted the fully allocated offer of the other player, the game terminated. Participants were paid near the MTurk market rate and were further incentivized by the promise of lottery tickets for a series of \$10 bonus awards. Each participant was awarded lottery tickets equal to the number of points he or she scored in the game.

All events that took place in the IAGO framework were recorded for analysis, including the final score, whether the game ended on a timeout, the number of messages passed, and how many times (if any) the participant lied (by expressing preferences about the issues that were untrue), and other measures.

6. RESULTS AND DISCUSSION

Our first result shows that pairs that involved strategic agents, which were more "cagey" about revealing information about their preferences, ended up having a significantly lower total amount of points than pairs with the free agents. We performed a univariate analysis of variance (ANOVA) to examine the effect of strategic vs. free agents on joint value at the end of the negotiation. There was a significant negative effect associated with the strategic agents, $F(1, 194) = 5.887, p = .016, d = 0.352$.² See Figure 2.

This unified loss of value may be attributable to the increased amount of negotiations that timed out and forced both players to take their BATNA, thus reducing the total value to a mere 8 points. Indeed, this was the case, as verified by a χ^2 analysis which revealed significance: $\chi^2 [1, N = 196], = 5.737, p = .014$.³

We can further analyze this loss of value by performing ANOVAs on both the human and agent points to see if the effect on the total value is driven by only one of them. The results of this ANOVA reveal that both values are significant, indicating that the hit in value is shouldered by both the human and the agent. F statistics for the human and agent are provided, respectively: $F(1, 194) = 6.638, p = .011, d = 0.375$, and then $F(1, 194) = 4.688, p = .032, d = 0.314$.

² F (between-groups DoF, within-groups DoF) = F statistic, p = significance, d = Cohen's d

³ χ^2 [degrees of freedom, sample size] = Pearson's χ^2 , p = significance

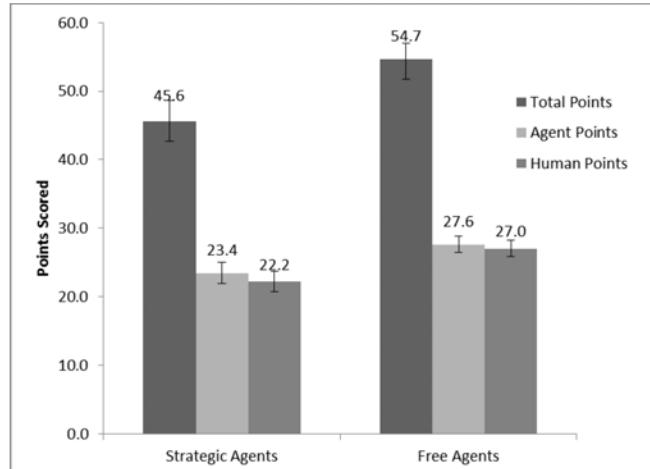


Figure 2: Strategic Agents "Shrink the pie"

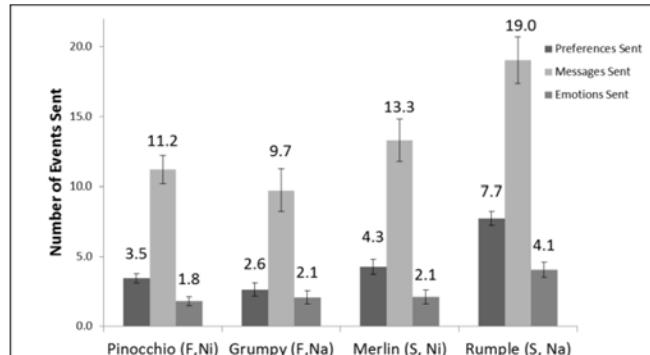


Figure 3: People Interact More with Strategic, Nasty Agents

From these results, it may be easy to take a dismal view on the usefulness of strategies that are not immediately forthright with information. These results are germane to our second research question (Q2), as they indicate that strategic agents are losing joint value. Indeed, this seems in line with negotiation literature that promotes relationship building and compromise, in order to avoid losing the integrative potential of a given situation. However, if these pairs are not reaching agreement, it begs the question of what they are doing during the negotiation, and whether or not discourse increased during these negotiations (per Q1).

We can analyze the behavior of the human player during negotiations with strategic agents vis-à-vis free agents. Figure 3 demonstrates several significant effects. First, participants sent substantially more messages to the strategic, nasty agent than they did to its other counterparts.

This effect is verified by again performing a univariate ANOVA comparing revelation strategy type to the amount of user messages: $F(1, 194) = 15.361, p < .001, d = 0.517$. There is an interaction with the language the agent used (nice vs. nasty). Rumple, the strategic, nasty agent, received the most messages, while Grumpy, the free, nice agent, received the fewest messages: $F(1, 194) = 6.619, p = .014, d = 0.358$.

Preferences, which are themselves a subset of messages, were also significantly higher, and an ANOVA reveals: $F(1, 194) = 12.534, p = .001, d = 0.434$. Language again had an interaction effect, $F(1, 194) = 6.523, p = .011, d = 0.370$. It should be expected that a strategic agent that requires the player to send preference data in order to receive it would have an effect here, and that effect

clearly does drive the effect of user messages as well. However, due to the interaction, nasty language increases the amount of preferences sent for the strategic agent, but decreases it for the free agent. Yet, if we look at user messages *not counting preferences*, the effect remains significant: $F(1, 194) = 4.949, p = .027, d = 0.303$, but there is no significant interaction effect. We can therefore conclude that the strategy increases the amount of discourse that the human player sends, even while it degrades joint value. This result addresses our first research question (Q1), by showing that even the smallest change in agent behavior can have far-reaching effects on human garrulousness.

This loquacious effect is not limited to messages alone. Players are also significantly more likely to send expressive emotions to strategic agents over free ones. ANOVA: $F(1, 194) = 6.026, p = .015, d = -0.342$. Here, however, we can find another main effect with the language the agent. Humans also emote more with the nasty agents over the nice agents. See Figure 3 for a visual guide, with ANOVA results of $F(1, 194) = 5.760, p = .017, d = 0.332$.

However, while these results discuss implications for shared value and external events such as messages, they say little about the effect on actually winning the negotiation by having more points than one's opponent. One of the core results of human-human negotiation is that aggressive behavior (such as that of the nasty agents) can often intimidate opponents into giving up value [6],[7],[26]. In the human-agent context, we examine this effect by comparing condition to the winner (which player has more points). The χ^2 analysis for strategic agents proves not to be significant: $\chi^2 [1, N = 196], = 1.242, p = \text{ns}$, but when examining nice vs. nasty agents, we can see an inverse relationship between niceness and winning. $\chi^2 [1, N = 196], = 5.879, p = .011$. When agents are nice, they end up losing. This falls nicely in line with our initial ideas in Q3 that aggressive behavior should help gain ground. Further, we can state that strategic information revelation does not hurt the agent's chances, a result in line with Thompson [24].

We can examine the result a different way by looking at the player lead in points. In human negotiation, we expect that aggressive (nasty) tactics would allow the player that employs them to claim value, independent of the size of the pie. Figure 4 demonstrates that indeed, there is a main effect of niceness. A univariate ANOVA test demonstrates significance, $F(1, 194)=5.780, p=0.017, d=0.416$. Further, "Pinocchio", the nice, free agent is the only agent that has an average negative lead against the human. All other agents beat the human in points, in the average case, although this trend is not significant.

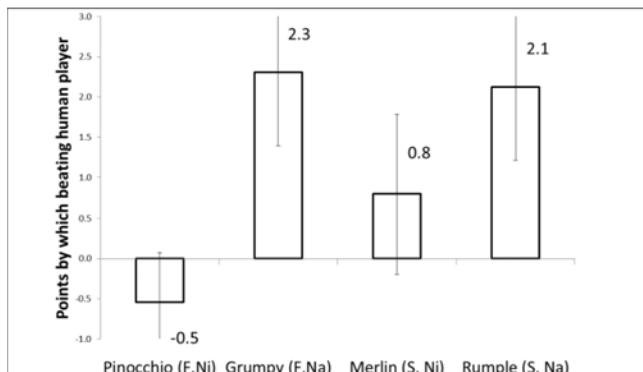


Figure 4: Nasty Agents Win against Humans

7. CONCLUSIONS

The first goal of any research that aims to create virtual agents that interact with humans should be that they serve their intended purpose adequately. With IAGO, we showcased some of the strengths of the platform by creating agents that were able to engage in a robust negotiation with a human in one of negotiations most classic prototypical problems: the multi-issue bargaining task. In that goal, our success can be measured by the agent's success: 30 times the agents collectively outright beat their human partner, compared to 26 times they lost. Often, they performed on the same level, tying for points.

Of course, looking in aggregate could be trivial, if the changes that were made to each agent did not have any effect on the human partner. But each interaction with our agent did have significant effects of the behavior their partners. Humans that negotiated with our strategic Rumple and Merlin agents sent more messages, more preferences, and more emotional expressions, a strong answer to Q1. Indeed, the Rumple agent encouraged the liveliest response, its combination of strategy and nasty dialogue serving perhaps to frustrate its opponents into discussion.

We are able to reproduce some classical human-human results within this human-agent context, showing that aggressive language and emotional displays can serve to help agents win against their human counterparts (Q3). We also showed that strategic behavior may reduce joint value, as we attempted to answer Q2. And indeed, the Rumple agent was able to grow joint value and subsequently claim the majority of it (Figure 4).

This work is encouraging in that it demonstrates the strength of both the IAGO platform for designing agents, as well as the ease by which experimental protocols may be designed and run. Human-human results, which often differ markedly from agent-agent results, are able to be reproduced using IAGO. Further, this work shows the deep importance of leveraging channels not often used in traditional computational negotiation to bring about desired results. These channels are not yet fully understood, and additional work such as the experiment conducted in this work should be conducted to further tease apart the factors that led to the most successful agents. The actions of a fully-realized human-aware agent, from emotional displays to natural language, are critical to the continued development of the field, and will yield agents that challenge their counterparts, and eventually teach the negotiation skills so necessary to life.

8. ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers of this work for their many insights. This work was supported by the National Science Foundation under grant BCS-1419621 and the U.S. Army. Any opinion, content or information presented does not necessarily reflect the position or the policy of the United States Government, and no official endorsement should be inferred.

9. REFERENCES

- [1] Aquino, K., & Becker, T. E. (2005). Lying in negotiations: How individual and situational factors influence the use of neutralization strategies. *Journal of Organizational Behavior*, 26(6), 661-679.
- [2] Baarslag, T., & Hindriks, K. V. (2013, May). Accepting optimally in automated negotiation with incomplete information. In Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems (pp. 715-722). International Foundation for Autonomous Agents and Multiagent Systems.
- [3] Broekens, J., Harbers, M., Brinkman, W.-P., Jonker, C. M., Van den Bosch, K., & Meyer, J.-J. (2012). "Virtual reality negotiation training increases negotiation knowledge and skill". 12th International Conference on Intelligent Virtual Agents. Santa Cruz, CA
- [4] Blascovich, J. (2002). Social influence within immersive virtual environments. In *The social life of avatars* (pp. 127-145). Springer London.
- [5] Core, M., Traum, D., Lane, H. C., Swartout, W., Gratch, J., Van Lent, M., & Marsella, S. (2006). "Teaching negotiation skills through practice and reflection with virtual humans". *Simulation*, 82(11), 685-701.
- [6] de Melo, C. M., Carnevale, P., & Gratch, J. (2011, May). "The effect of expression of anger and happiness in computer agents on negotiations with humans "In The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 3 (pp. 937-944). International Foundation for Autonomous Agents and Multiagent Systems.
- [7] de Melo, C., Gratch, J., & Carnevale, P. (2014). Humans vs. Computers: Impact of Emotion Expressions on People's Decision Making.
- [8] Faratin, P., Sierra, C., & Jennings, N. R. (2002). Using similarity criteria to make issue trade-offs in automated negotiations. *artificial Intelligence*, 142(2), 205-237.
- [9] Fatima, S. S., Wooldridge, M., & Jennings, N. R. (2007, May). Approximate and online multi-issue negotiation. In Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems (p. 156). ACM.
- [10] Fox, J., Ahn, S. J., Janssen, J. H., Yeykelis, L., Segovia, K. Y., & Bailenson, J. N. (2015). Avatars versus agents: a meta-analysis quantifying the effect of agency on social influence. *Human-Computer Interaction*, 30(5), 401-432.
- [11] Gal, Y. A., Grosz, B. J., Kraus, S., Pfeffer, A., & Shieber, S. (2005, July). Colored trails: a formalism for investigating decision-making in strategic environments. In *Proceedings of the 2005 IJCAI workshop on reasoning, representation, and learning in computer games* (pp. 25-30).
- [12] Gratch, J., Nazari, Z., & Johnson, E. (2016, May). The Misrepresentation Game: How to win at negotiation while seeming like a nice guy. In Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems (pp. 728-737). International Foundation for Autonomous Agents and Multiagent Systems.
- [13] Kelley, H. H. (1966). A classroom study of the dilemmas in interpersonal negotiations. *Strategic interaction and conflict*, 49, 73.
- [14] Kraus, S. (2001). Strategic negotiation in multiagent environments. MIT press.
- [15] Lucas, G., Stratou, G., Lieblich, S., & Gratch, J. (2016, October). Trust me: multimodal signals of trustworthiness. In Proceedings of the 18th ACM International Conference on Multimodal Interaction (pp. 5-12). ACM.
- [16] Mell, J., & Gratch, J. (2016, May). IAGO: Interactive Arbitration Guide Online. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems* (pp. 1510-1512). International Foundation for Autonomous Agents and Multiagent Systems.
- [17] Mell, J., Lucas, G., & Gratch, J. (2015). An Effective Conversation Tactic for Creating Value over Repeated Negotiations. In Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May, 4-8, 2015, Istanbul, Turkey.
- [18] Olekalns, M., & Smith, P. L. (2009). Mutually dependent: Power, trust, affect and the use of deception in negotiation. *Journal of Business Ethics*, 85(3), 347-365.
- [19] Patton, B. (2005). Negotiation. *The Handbook of Dispute Resolution*, Jossey-Bass, San Francisco, 279-303.
- [20] Peled, N., Gal, Y. A. K., & Kraus, S. (2011, May). A study of computational and human strategies in revelation games. In The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1 (pp. 345-352).
- [21] Raiffa, H. (1982). *The art and science of negotiation*. Harvard University Press.
- [22] Rosenfeld, A., Zuckerman, I., Segal-Halevi, E., Drein, O., & Kraus, S. (2014, May). NegoChat: a chat-based negotiation agent. In Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems (pp. 525-532). International Foundation for Autonomous Agents and Multiagent Systems.
- [23] Robu, V., Somefun, D. J. A., & La Poutré, J. A. (2005, July). Modeling complex multi-issue negotiations using utility graphs. In Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems (pp. 280-287). ACM.
- [24] Thompson, L. L. (1991). Information exchange in negotiation. *Journal of Experimental Social Psychology*, 27(2), 161-179.
- [25] White, J. J. (1980). Machiavelli and the bar: Ethical limitations on lying in negotiation. *Law & Social Inquiry*, 5(4), 926-938.
- [26] Van Kleef, G. A., De Dreu, C. K., & Manstead, A. S. (2004). "The interpersonal effects of emotions in negotiations: a motivated information processing approach". *Journal of personality and social psychology*, 87(4), 510.
- [27] Yang, Y., Falcão, H., Delicado, N., & Ortony, A. (2014). Reducing Mistrust in Agent-Human Negotiations. *IEEE Intelligent Systems*, 29(2), 36-43.

Randomized Kinodynamic Planning

Steven M. LaValle

Dept. of Computer Science
Iowa State University
Ames, IA 50011 USA
lavalle@cs.iastate.edu

James J. Kuffner, Jr.

Dept. of Computer Science
Stanford University
Stanford, CA 94305 USA
kuffner@stanford.edu

Abstract This paper presents a state-space perspective on the kinodynamic planning problem, and introduces a randomized path planning technique that computes collision-free kinodynamic trajectories for high degree-of-freedom problems. By using a state space formulation, the kinodynamic planning problem is treated as a $2n$ -dimensional nonholonomic planning problem, derived from an n -dimensional configuration space. The state space serves the same role as the configuration space for basic path planning; however, standard randomized path planning techniques do not directly apply to planning trajectories in the state space. We have developed a randomized planning approach that is particularly tailored to kinodynamic problems in state spaces, although it also applies to standard nonholonomic and holonomic planning problems. The basis for this approach is the construction of a tree that attempts to rapidly and uniformly explore the state space, offering benefits that are similar to those obtained by successful randomized planning methods, but applies to a much broader class of problems. Some preliminary results are discussed for an implementation that determines kinodynamic trajectories for hovercrafts and satellites in cluttered environments, resulting in state spaces of up to twelve dimensions.

1 Introduction

There is a strong need for a simple, efficient planning technique that determines control inputs to drive a robot from an initial configuration and velocity to a goal configuration and velocity while obeying physically-based dynamic constraints and avoiding obstacles in the robot's environment. Although many interesting approaches exist to specific kinodynamic problems, they fall short of being able to solve many complicated, high degree-of-freedom problems. Randomized techniques have led to efficient, incomplete planners for basic path planning (holonomic and purely kinematic); however, there appears to be no equivalent technique for the broader kinodynamic planning problem (or even nonholonomic planning in the configuration space). We try to account for some of the reasons for this, and argue the need for a simple, general-purpose kinodynamic planner. We present a heuristic, randomized approach to kinodynamic planning that quickly explores the state space, and scales well for problems with high degrees-of-freedom and complicated system dynamics.

The common model in motion planning research has been to decouple the general robotics problem by solv-

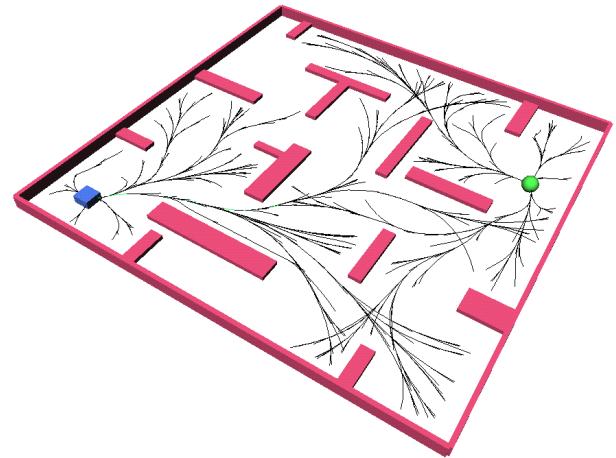


Figure 1. We consider planning problems with dynamic constraints induced by physical laws. The above image shows the state exploration trees computed for a rigid rectangular object (left). The goal location is represented by a sphere (upper right).

ing basic path planning, and then finding a trajectory and controller that satisfies the dynamics and follows the path [3, 16, 24]. The vast majority of basic path planning algorithms consider only *kinematics*, while ignoring the system *dynamics* entirely. Motion planning that takes into account dynamic constraints as well as kinematic constraints is known as *kinodynamic planning* [10]. In this paper, we consider kinodynamic planning as a generalization of holonomic and nonholonomic planning in configuration spaces, by replacing popular configuration-space notions by their *state space* (or phase space) counterparts. A point in the state space includes both configuration parameters and velocity parameters (i.e., it is the tangent bundle of the configuration space).

It may be the case that the result of a purely kinematic planner will be *unexecutable* by the robot in the environment due to limits on the actuator forces and torques. Imprecision in control, which is always present in real-world robotic systems, may require explicitly modeling system dynamics to guarantee collision-free trajectories. Robots with significant dynamics are those in which natural physical laws, along with limits on the avail-

able controls, impose severe constraints on the allowable velocities at each configuration. Examples of such systems include helicopters, airplanes, certain-classes of wheeled vehicles, submarines, unanchored space robots, and legged robots with fewer than four legs. In general, it is preferable to look for solutions to these kinds of systems that naturally flow from the physical models, as opposed to viewing dynamics as an obstacle.

These concerns provide the general basis for kinodynamic planning research. Algebraic approaches solve for the trajectory exactly, though the only known solutions are for point masses with velocity and acceleration bounds in one-dimension[19] and two-dimensions[5]. Provably approximately-optimal kinodynamic trajectories are computed in [10] by performing a search of the state space by systematically applying control inputs. Other papers have extended or modified this technique [9, 8, 12, 21]. In [11], an incremental, variational approach is presented to perform state-space search. An approach to kinodynamic planning based on Hamiltonian mechanics is presented in [7]. Sensor-based motion strategies that account for robot inertia have been devised that maintain an emergency stopping path [25].

The computational complexity of kinodynamic planning problems depends upon the assumptions made for a particular instance of the problem. However, kinodynamic planning in general is believed to be at least as hard as the *generalized mover’s problem*, which has been proven to be PSPACE-hard [22]. Hard bounds have also been established for time-optimal trajectories. Finding an exact time-optimal trajectory for a point mass with bounded acceleration and velocity moving amidst polyhedral obstacles in 3D has been proven to be NP-hard [10]. The need for simple, efficient algorithms for kinodynamic planning, along with the discouraging lower-bound complexity results, have motivated us to explore the development of randomized techniques for kinodynamic planning. This parallels the reasoning that led to the success of randomized planning techniques for basic path planning.

2 A State Space Formulation

We formulate the kinodynamic planning problem as path planning in an $2n$ -dimensional state space that has first-order nonholonomic constraints. We would like the state space to have the same utility as a representational tool as the configuration space for a purely-kinematic problem. Let \mathcal{C} denote the configuration space (\mathcal{C} -space) that arises from a rigid or articulated body that moves in a 2D or 3D world. Let \mathcal{X} denote the state space, in which a state, $x \in \mathcal{X}$, is defined as $x = (q, \dot{q})$, for $q \in \mathcal{C}$.

Constraints When planning in \mathcal{C} , nonholonomic constraints often arise from the presence of one or more rolling contacts between rigid bodies, or from the set of controls that it is possible to apply to a system. When planning in \mathcal{X} , nonholonomic constraints also arise from conservation laws (e.g. angular momentum conservation). Using Lagrangian mechanics, the dynamics can be represented by a set of equations of the form $h_i(\ddot{q}, \dot{q}, q) = 0$. Using the state space representation, this

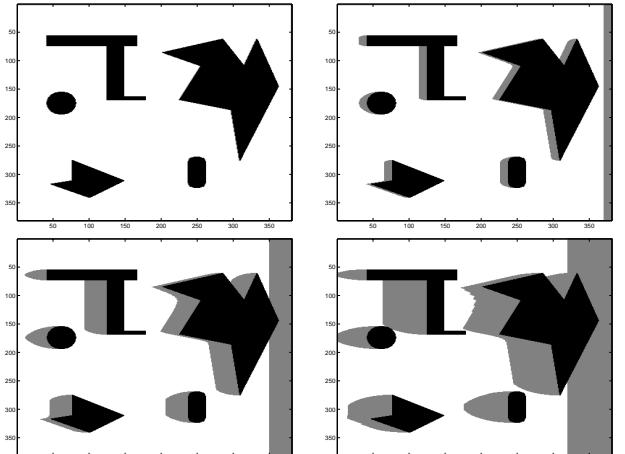


Figure 2. Slices of \mathcal{X} for a point mass robot in 2D with increasingly higher initial speeds. White areas represent \mathcal{X}_{free} ; black areas are \mathcal{X}_{obst} ; gray areas approximate \mathcal{X}_{ric} .

can be simply written as a set of m implicit equations of the form $G_i(x, \dot{x}) = 0$, for $i = 1, \dots, m$ and $m < 2n$. It is well known that under appropriate conditions the Implicit Function Theorem allows the constraints to be written in the form of a control system

$$\dot{x} = f(x, u) \quad (1)$$

in which $u \in U$, and U represents a set of allowable controls or inputs. We assume that f is a smooth function of (x, u) . Equation 1 effectively yields a convenient parameterization of the allowable state transitions via the controls in U .

Obstacles in \mathcal{X} Assume that the world in which the robot lives contains static obstacles. There are interesting differences between finding collision-free paths in \mathcal{C} versus the state space, \mathcal{X} . When planning in \mathcal{C} , it is useful to characterize the set \mathcal{C}_{obst} of configurations at which the robot is in collision with an obstacle (or itself) [16]. The path planning problem involves finding a continuous path that maps into $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obst}$. For planning in \mathcal{X} , this could lead to a straightforward definition of \mathcal{X}_{obst} by declaring $x \in \mathcal{X}_{obst}$ if and only if $q \in \mathcal{C}_{obst}$ for $x = (q, \dot{q})$. However, another interesting possibility exists: the *region of inevitable collision*. Let \mathcal{X}_{ric} denote the set of states in which the robot is either in collision or, because of its velocity, it cannot do anything to avoid collision (there exist no controls that will prevent it). Note that $\mathcal{X}_{obst} \subseteq \mathcal{X}_{ric}$. Thus, it might be preferable to define $\mathcal{X}_{free} = \mathcal{X} \setminus \mathcal{X}_{ric}$, as opposed to $\mathcal{X} \setminus \mathcal{X}_{obst}$.

Figure 2 illustrates conservative approximations of \mathcal{X}_{ric} for a point mass robot. The robot is assumed to have L^2 -bounded acceleration, and an initial velocity pointing along the positive x axis. As expected intuitively, if the speed increases, \mathcal{X}_{ric} grows.

Solution Trajectory The kinodynamic planning problem is to find a trajectory from an initial state x_{init}

to a goal state x_{goal} . A trajectory is defined as a time-parameterized continuous path $\tau : [0, T] \rightarrow \mathcal{X}_{free}$ that satisfies the nonholonomic constraints. By integrating Equation 1 from an initial state and control, we obtain a trajectory that inherently satisfies the nonholonomic constraints. Our task is then reduced to finding a control function $u : [0, T] \rightarrow U$, representing time-varying input that when applied, moves the system from x_{init} to x_{goal} while avoiding obstacles. It might also be appropriate to select a path that optimizes some criterion, such as the time to reach x_{goal} .

3 Issues in Randomized Kinodynamic Planning

One of the key differences between \mathcal{X} and \mathcal{C} is a factor of two in dimension. The curse of dimensionality has already contributed to the success and popularity of randomized planning methods for \mathcal{C} -space; therefore, it seems that there would be an even greater need to develop randomized algorithms for kinodynamic planning. One reason that might account for the lack of practical, efficient planners for problems in \mathcal{X} -space is that attention is usually focused on the decoupled planning problem. Another plausible reason is that kinodynamic planning is considerably harder, aside from the fact that the dimension is larger. We briefly indicate some reasons for this difficulty.

Existing Randomized Techniques It would certainly be useful if ideas can be borrowed or adapted from existing randomized path planning techniques that have been successful for planning in \mathcal{C} -space. For the purpose of discussion, we choose two different techniques that have been successful in recent years: randomized potential fields (e.g., [2, 6]) and randomized roadmaps (e.g., [1, 15]). In the randomized potential field approach, a heuristic function is defined on the configuration space that attempts to steer the robot toward the goal through gradient descent. If the search becomes trapped in a local minimum, random walks are used to help escape. In the randomized roadmap approach, a graph is constructed in the configuration space by generating random configurations and attempting to connect pairs of nearby configurations with a local planner. Once the graph has been constructed, the planning problem becomes one of searching a graph for a path between two nodes.

Why is Kinodynamic Planning Harder? Consider applying either of the previously mentioned randomized techniques to the problem of finding a path in \mathcal{X}_{free} that also satisfies (1), instead of finding a holonomic path in \mathcal{C}_{free} . The potential field method appears nicely suited to the problem because a discrete-time control can repeatedly selected that reduces the potential. The primary problem is that dynamical systems usually have drift, which could easily cause the robot to overshoot the goal, leading to oscillations. Without a cleverly-constructed potential function (which actually becomes a difficult nonlinear control problem), the method cannot be expected to work well. Imagine how often the system

will be pulled into X_{ric} . The problem of designing a good heuristic function becomes extremely complicated for the case of kinodynamic planning.

The randomized roadmap technique might also appear amenable to kinodynamic planning. The primary requirement is the ability to design a local planner that will connect pairs of configurations (or states in our case) that are generated at random. Indeed, this method was successfully applied to a nonholonomic planning problem in [27]. One result that greatly facilitated this extension of the technique to nonholonomic planning was the existence of Reeds-Shepp curves [20] for car-like robots. This result directly enables the connection of two configurations with the optimal-length path. For more complicated problems, such as kinematic planning for a tractor-trailer, a reasonable roadmap planner can be developed using steering results [4, 17, 23, 26]. These results enable a system to be driven from one configuration to another, and generally apply to driftless systems that are nilpotentizable (a condition on the underlying Lie algebra). In general, however, the connection problem can again be as difficult as designing a nonlinear controller. The randomized roadmap technique might require the connections of thousands of states to find a solution, and if each connection is akin to a nonlinear control problem, it seems impractical for systems that do not allow steering.

4 Rapidly-Exploring Random Trees

The observations of Section 3 motivated us to develop a randomized planning technique that nicely extends to kinodynamic planning (it also applies to the simpler problems of nonholonomic planning in \mathcal{C} and basic path planning in \mathcal{C}). Our intention has been to develop a method that easily “drives forward” like potential field methods, and also quickly and uniformly explores the space like randomized roadmap methods. This led us to develop Rapidly-Exploring Random Trees (RRTs).

To motivate and illustrate the concepts, first consider the simple case of planning for a point robot in a two-dimensional configuration space. To prepare for the extension to kinodynamic planning, suppose that the motion of the robot is governed by a control law, $x_{k+1} = f(x_k, u_k)$, which is considered as a discrete-time approximation to (1). For this simple problem, suppose that U represents a direction in S^1 toward which the robot can be moved a fixed, small distance in time Δt . Consider Figure 3, in which the robot starts at (50,50) in an environment that ranges from (0,0) to (100,100), and the robot can move 2 units in one application of the discrete-time control law. The first scheme can be considered as a Naive Random Tree, which is incrementally constructed by randomly picking an existing vertex, x_k from the tree, a control $u_k \in U$ at random, and adding an edge of length 2 from x_k to $f(x_k, u_k)$. Although it appears somewhat random, this tree has a very strong bias towards places it has already explored. To overcome this bias, we propose to construct a Rapidly-Exploring Random Tree as follows. Insert the initial state as a vertex. Repeatedly select a point at random in $[0, 100] \times [0, 100]$, and find the nearest-neighbor, x_k , in the tree. Choose the control $u_k \in U$ that pulls the ver-

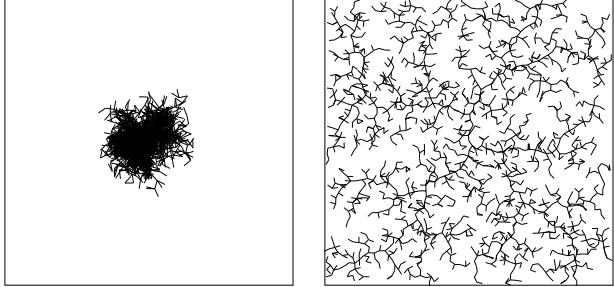


Figure 3. A Naive Random Tree vs. a Rapidly-Exploring Random Tree. Each tree has 2000 vertices.

tex toward the random point. Insert the new edge and vertex for $x_{k+1} = f(x_k, u_k)$. This technique generates a tree that rapidly and uniformly explores the state space. An argument for this can be made by considering the Voronoi regions of the vertices. Random sampling tends to extend vertices that have larger Voronoi regions, and are therefore have too much unexplored space in their vicinity. By incrementally reducing the size of larger Voronoi regions, the graph spreads in a uniform manner.

Rapidly Exploring the State Space When moving from the problem shown in Figure 3 to exploring \mathcal{X} for a kinodynamic planning problem, several complications immediately occur: i) the dimension is typically much higher; ii) the tree must stay within \mathcal{X}_{free} ; iii) drift and other dynamic constraints can yield undesired motions and biases; iv) there is no natural metric on \mathcal{X} for selecting “nearest” neighbors. For the first complication, approximate nearest neighbor techniques [14] can be employed to help improve performance; it is not crucial to have the absolute closest neighbor. The second complication can make it harder to wander through narrow passageways, much like in the case of randomized roadmaps. The third complication can be partly overcome by choosing an action that brings the velocity components of x as close as possible toward the random sample. The fourth complication might lead to the selection of one metric over another for particular kinodynamic planning problems, if one would like to optimize performance. In theory, there exists a perfect metric (or pseudo-metric due to asymmetry) that could overcome all of these complications if it were easily computable. This is the optimal cost (for any criterion, such as time, energy, etc.) to get from one state to another. Unfortunately, computing the ideal metric as hard as solving the original planning problem. In general, we try to overcome these additional complications while introducing as few heuristics as possible. This enables the planner to be applied with minor adaptation to a broad class of problems.

A Randomized Planning Algorithm We present an algorithm that grows two RRTs, one rooted at the start state x_{init} , and the other rooted at x_{goal} . The algorithm searches for states that are “common” to both trees. Two states, x and x' , are considered to be common if $\rho(x, x') < \epsilon$ some some metric ρ and small $\epsilon > 0$. Our basic algorithm stops at the first solution trajec-

```

GROW_RANDOM_TREES()
1   InsertState( $\mathcal{T}_{init}$ , nil,  $x_{init}$ );
2   InsertState( $\mathcal{T}_{goal}$ , nil,  $x_{goal}$ );
3   while continuePlan do
4        $x_{rand} \leftarrow \text{RandomState}();$ 
5        $x_i \leftarrow \text{GEN\_STATE}(\mathcal{T}_{init}, x_{rand}, FWD);$ 
6       if  $x_i \neq \text{nil}$  and  $\text{NearbyState}(\mathcal{T}_{goal}, x_i)$  then
7           record candidate solution  $\tau$  connecting
                 $\mathcal{T}_{init}$  and  $\mathcal{T}_{goal}$  through  $x_i$ ;
8        $x_g \leftarrow \text{GEN\_STATE}(\mathcal{T}_{goal}, x_{rand}, BACK);$ 
9       if  $x_g \neq \text{nil}$  and  $\text{NearbyState}(\mathcal{T}_{init}, x_g)$  then
10      record candidate solution  $\tau$  connecting
                 $\mathcal{T}_{init}$  and  $\mathcal{T}_{goal}$  through  $x_g$ ;

```

Figure 4. The algorithm incrementally grows two RRTs, from the start and the goal, until meeting at a common state.

tory found, but one could continue to grow the trees and maintain a growing collection of solution trajectories. The “best” solution found so far can be chosen according to a cost functional based on some criteria (such as execution time or energy expended).

Here we present the algorithm in detail. The pseudocode for GROW_RANDOM_TREES() is shown in Figure 4. To begin, we define two RRTs, \mathcal{T}_{init} and \mathcal{T}_{goal} , each initialized to contain a single node representing x_{init} and x_{goal} , respectively. We pick a random state x_{rand} and generate new nodes in both trees via the function GEN_STATE(). First, the nearest neighbor of x_{rand} in \mathcal{T}_{init} is selected. From this state, all possible controls are applied to the system for a fixed time interval Δt , yielding successor states derived from $(\mathbf{u}, \Delta t) - \text{bangmotions}$. The successor states are generated by integrating (1) over Δt . A successor state x_{k+1} that satisfies velocity bounds, is collision-free, and minimizes $\rho(x_{k+1}, x_{rand})$ is inserted into the tree and returned. It is then tested to see if it lies within an ϵ -neighborhood of any of the states generated so far in \mathcal{T}_{goal} . If so, we have found a common state that joins the two trees, and we record the candidate solution trajectory τ that joins \mathcal{T}_{init} and \mathcal{T}_{goal} and passes through the node. If not, we invoke GEN_STATE() again on \mathcal{T}_{goal} with the same random state x_{rand} . Successor states are generated exactly as before, except for one minor change. Since we are searching backwards from the goal, we integrate the state transition equation backwards in time. The returned successor state is tested to see if it lies within an ϵ -neighborhood of any of the currently explored states in \mathcal{T}_{init} . If so, the candidate solution trajectory is recorded. The algorithm terminates on the first successful solution found.

Our initial experiments attempted to grow a single RRT from x_{init} to connect with the goal x_{goal} . These experiments worked well for state spaces of low dimension. However, growing dual trees improves efficiency for state spaces of high dimension, at the expense of having to connect a pair of nodes between the two trees. In higher dimensions it becomes more difficult to randomly wander upon a state that is close enough to the goal state for each of the state space variables. It might also be possible to improve performance by biasing the

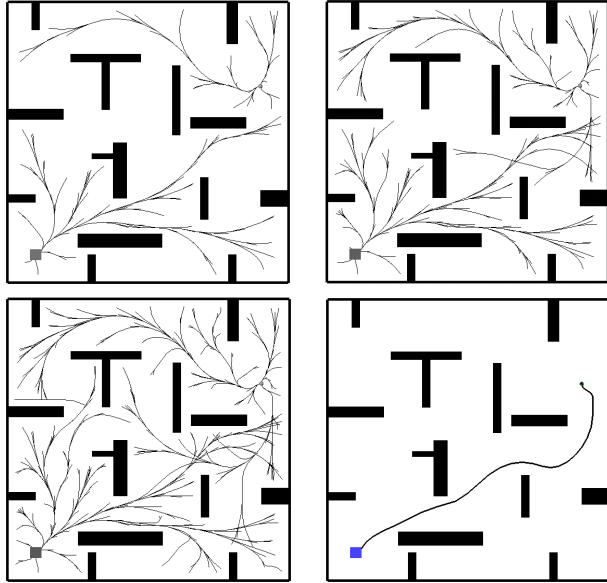


Figure 5. Various stages of state exploration during planning. The top two images show the RRTs after 500 and 1000 nodes, respectively. The bottom two images show the final trees and the computed solution trajectory after 1582 nodes.

sampling toward goal states; this is currently under investigation as an alternative to using dual trees. We generally want to avoid defining some complicated artificial bias because we might be faced with a challenge that is similar to defining a good artificial potential function for the potential field approach to path planning (this task should be even harder for kinodynamic planning).

5 Hovercrafts and Satellites

Basic experiments involving RRTs in the state space were conducted on simple kinodynamic systems. The algorithm was implemented in C++ on a 200MHz SGI Indigo2 with 128MB of memory. The systems considered involve both non-rotating and rotating rigid objects in 2D and 3D with velocity and acceleration bounds obeying L_2 norms. The dynamic models were derived from Newtonian mechanics of rigid bodies in non-gravity environments. All experiments utilized a simple metric on \mathcal{X} based on a weighted Euclidean distance for position coordinates and their derivatives, along with a weighted metric on unit quaternions for rotational coordinates and their derivatives.

Planar Translating Body ($\dim \mathcal{X} = 4$) The first experiment considered a rigid object with a set of translational controls that restrict its motion to a plane. A total of 4 controls were used, consisting of a set of two pairs of opposing forces acting through the center of mass of the body. Figure 5 shows snapshots during various stages of the computation. Anywhere between 500 and 2500 nodes are explored on average before a solution trajectory is found, with total computation time ranging between 5 and 15 seconds.

Planar Body with Rotation ($\dim \mathcal{X} = 6$) We extend the previous experiments to consider systems with

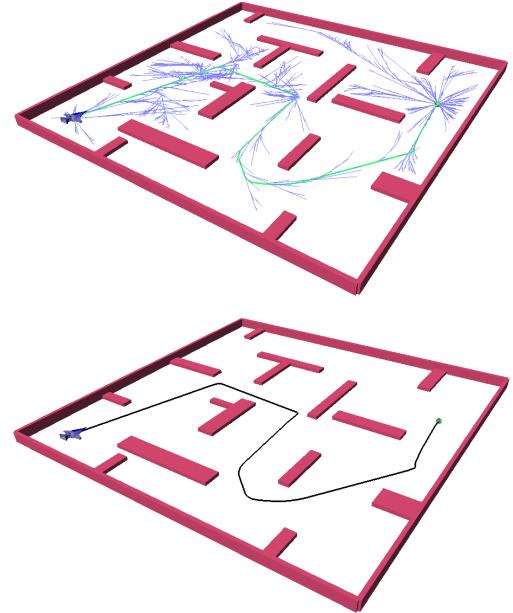


Figure 6. RRTs of 13,600 nodes and solution trajectory for the planar body with unilateral thrusters that allow it to rotate freely but translate only in the forward direction.

rotation. First, we consider the case of a rigid object with two unilateral thrusters each producing a torque of opposite sign, such as the one described in [18]. Each thruster provides a line of force fixed in the body frame that restricts its motion to a plane. This model is similar to that of a hovercraft, navigating with drift. The state space of this system has 6 degrees of freedom, but only 3 controls: translate forward, rotate clockwise, and rotate counter-clockwise are provided. Figure 6 shows the RRT after 13,600 nodes. The total computation time for this example was 4.2 minutes.

Translating 3D Body ($\dim \mathcal{X} = 6$) We consider the case of a free-floating rigid object, such as an unanchored satellite in space. The object is assumed to be equipped with thruster controls to be used for translating in a non-gravity environment. The satellite has three opposing pairs of thrusters along each of its principal axes forming a set of six controls spanning a 6-dimensional state space. The task is to thrust through a sequence of two narrow passages amidst a collection of obstacles. Figure 7 shows the RRTs generated during the planning process, and Figure 8 shows the candidate solution found after a total of 16,300 nodes were explored. The total computation time for this case was 4.1 minutes.

3D Body with Rotation ($\dim \mathcal{X} = 12$) Finally, we consider the case of a fully-orientable satellite model with limited translation. The satellite is assumed to have momentum wheels that enable it to orient itself along any axis, and a single pair of opposing thruster controls that allow it to translate along the primary axis of the cylinder. This model has a 12-dimensional state space. The task of the satellite, modeled as a rigid cylindrical object, is to perform a collision-free docking maneuver into the cargo bay of the space shuttle model amidst a

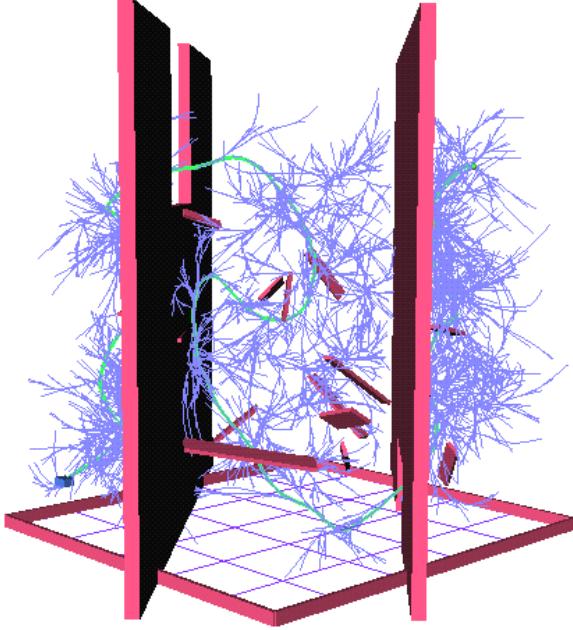


Figure 7. The RRTs computed for the task of navigating a sequence of narrow passages for the 3D translation case.

cloud of obstacles. Figure 9 illustrates all trajectories explored during the planning process, and Figure 10 shows the candidate solution found after 23,800 states were explored. The total computation time was 8.4 minutes.

6 Discussion

We believe randomized kinodynamic planning techniques will prove useful in a wide array of applications that includes robotics, virtual prototyping, and computer graphics. We presented a state-space perspective on the kinodynamic planning problem that is modeled after the configuration-space perspective on basic path planning. We then presented an efficient, randomized planning technique that is particularly suited to the difficulties that arise in kinodynamic planning. We implemented this technique and generated experiments for hovercraft problems of up to 12 degrees-of-freedom. We still consider this work to be in a preliminary stage, and we are experimenting with different RRT-based variants of the algorithm. Many more experiments will have to be performed, on a wide array of dynamical systems, to assess the full generality and adaptability of the approach. The planning technique appears to generate good paths; however, we make no claims that the paths are optimal or near optimal (this assumption is common for path planning algorithms in \mathcal{C}). One idea for further investigation might be to construct RRTs to find initial trajectories, and then employ a variational technique to optimize the trajectories (see, for example, [28]). We are currently exploring the use of RRTs for other problems, such as kinodynamic planning for automobiles).

Acknowledgments Steve LaValle thanks Bruce Donald for his advice on the kinodynamic planning problem. He

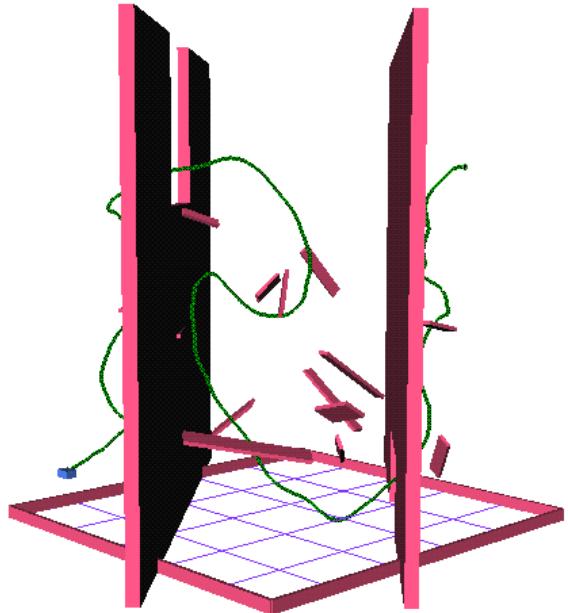


Figure 8. Solution trajectory for navigating through a sequence of narrow passages for the 3D translation case. The initial state is at the lower left; the goal is at the upper right.

also thanks Brian George for his helpful suggestions. Steve LaValle is supported in part by an NSF CAREER award. James Kuffner thanks Nobuaki Akatsu and Bill Behrman for providing him with sound technical advice during the initial stages of this research. James Kuffner is supported in part by an NSF Graduate Fellowship in Engineering.

References

- [1] N. M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *IEEE Int. Conf. Robot. & Autom.*, pages 113–120, 1996.
- [2] J. Barraquand and J.-C. Latombe. Robot motion planning: A distributed representation approach. *Int. J. Robot. Res.*, 10(6):628–649, December 1991.
- [3] J. Bobrow, S. Dubowsky, and J. Gibson. Time-optimal control of robotic manipulators. *Int. Journal of Robotics Research*, 4(3), 1985.
- [4] L.G. Bushnell, D.M. Tilbury, and S.S. Sastry. Steering three-input nonholonomic systems: The fire-truck example. *Int. Journal of Robotics Research*, 14(3), 1995.
- [5] J. Canny, A. Rege, and J. Reif. An exact algorithm for kinodynamic planning in the plane. *Discrete and Computational Geometry*, 6:461–484, 1991.
- [6] D. Chalou, D. Boley, M. Gini, and V. Kumar. A parallel formulation of informed randomized search for robot motion planning problems. In *IEEE Int. Conf. Robot. & Autom.*, pages 709–714, 1995.
- [7] C. Connolly, R. Grupen, and K. Souccar. A hamiltonian framework for kinodynamic planning. In *Proc. of the IEEE International Conf. on Robotics and Automation (ICRA'95)*, Nagoya, Japan, 1995.
- [8] B. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning for cartesian robots and open chain manipulators. *Algorithmica*, 14(6):480–530, 1995.

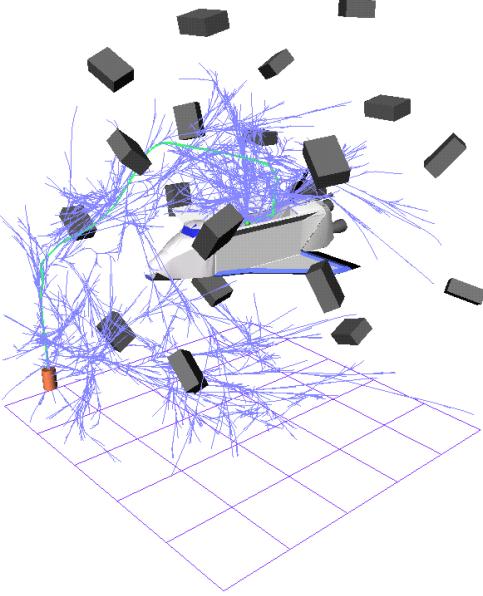


Figure 9. The RRTs constructed during planning for the fully-orientable satellite model with limited translation. A total of 23,800 states were explored before a successful candidate solution trajectory was found.

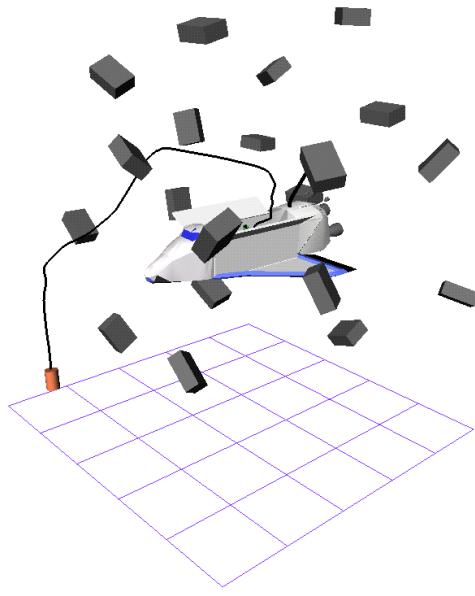


Figure 10. The docking maneuver computed for the fully-orientable satellite model. The satellite's initial state is in the lower left corner, and the goal state is in the interior of the cargo bay of the shuttle.

- [9] B. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning: Robots with decoupled dynamics bounds. *Algorithmica*, 14(6):443–479, 1995.
- [10] B. Donald, P. Xavier, J. Canny, and J. Reif. Kinodynamic motion planning. *Journal of the ACM*, 40(5):1048–1066, November 1993.
- [11] P. Ferbach. A method of progressive constraints for non-holonomic motion planning. In *Proc. of the IEEE International Conf. on Robotics and Automation (ICRA'96)*, pages 1637–1642, Minneapolis, MN, April 1996.
- [12] G. Heinzinger, P. Jacobs, J. Canny, and B. Paden. Time-optimal trajectories for a robotic manipulator: A provably good approximation algorithm. In *Proc. of IEEE Int. Conf. on Robotics and Automation*, pages 150–155, Cincinnati, OH, 1990.
- [13] D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *Int. J. Comput. Geom. & Appl.* To appear.
- [14] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, 1998.
- [15] L. Kavraki, P. Švestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration space. *Int. Transactions on Robotics and Automation*, 12(4):566–580, 1996.
- [16] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.
- [17] A. D. Lewis and R. M. Murray. Configuration controllability of simple mechanical control systems. *SIAM Journal on Control and Optimization*, 35(3):766–790, 1997.
- [18] K.M. Lynch. Controllability of a planar body with unilateral thrusters. *IEEE Trans. on Automatic Control*. To appear.
- [19] C. O'Dunlaing. Motion planning with inertial constraints. *Algorithmica*, 2(4):431–475, 1987.
- [20] J.A. Reeds and R.A. Schepp. Optimal paths for a car that goes both forward and backward. *Pacific Journal of Mathematics*, 145(2), 1990.
- [21] J. Reif and H. Wang. Non-uniform discretization approximations for kinodynamic motion planning. In J.-P. Laumond and M. Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 97–112. A K Peters, Wellesley, MA, 1997.
- [22] J. H. Reif. Complexity of the mover's problem and generalizations. In *Proc. 20th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 421–427, 1979.
- [23] S. Sekhavat, F. Lamiraux, J.-P. Laumond, G. Bauzil, and A. Ferrand. Motion planning and control for hilare pulling a trailer: experimental issues. In *Proc. of IEEE Int. Conf. on Robotics and Automation*, April 1997.
- [24] Z. Shiller and S. Dubowsky. On computing time-optimal motions of robotic manipulators in the presence of obstacles. *IEEE Trans. on Robotics and Automation*, 7(7), December 1991.
- [25] A. Shkel and V. Lumelsky. The jogger's problem : Control of dynamics in real-time motion planning. *Automatica, A Journal of the Int. Federation of Automatic Control*, 33(7):1219–1233, July 1997.
- [26] H. K. Struemper. *Motion Control for Nonholonomic Systems on Matrix Lie Groups*. PhD thesis, University of Maryland, College Park, MD, 1997.
- [27] P. Švestka and M.H. Overmars. Coordinated motion planning for multiple car-like robots using probabilistic roadmaps. In *Proc. of IEEE Int. Conf. on Robotics and Automation*, 1995.
- [28] M. Zefran, J. Desai, and V. Kumar. Continuous motion plans for robotic systems with changing dynamic behavior. In *Proc. of IEEE Int. Conf. on Robotics and Automation*, 1996.

Real-Time Adaptive A*

Sven Koenig

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781

skoenig@usc.edu

Maxim Likhachev

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3891

maxim+@cs.cmu.edu

ABSTRACT

Characters in real-time computer games need to move smoothly and thus need to search in real time. In this paper, we describe a simple but powerful way of speeding up repeated A* searches with the same goal states, namely by updating the heuristics between A* searches. We then use this technique to develop a novel real-time heuristic search method, called Real-Time Adaptive A*, which is able to choose its local search spaces in a fine-grained way. It updates the values of all states in its local search spaces and can do so very quickly. Our experimental results for characters in real-time computer games that need to move to given goal coordinates in unknown terrain demonstrate that this property allows Real-Time Adaptive A* to follow trajectories of smaller cost for given time limits per search episode than a recently proposed real-time heuristic search method [5] that is more difficult to implement.

Categories and Subject Descriptors

I.2 [Artificial Intelligence]: Problem Solving, Control Methods, and Search—*Graph and Tree Search Strategies*

General Terms

Algorithms

Keywords

A*; Agent Planning; D* Lite; Games; Heuristic Search; Incremental Search; Perception, Action and Planning in Agents; Planning with the Freespace Assumption; Real-Time Decision Making

1. INTRODUCTION

Agents need to use different search methods than the off-line search methods often studied in artificial intelligence. Characters in real-time computer games, for example, need to move smoothly and thus need to search in real time. Real-time heuristic search methods find only the beginning of a trajectory from the current state of an agent to a goal state [9, 4]. They restrict the search to a small part of the state space that can be reached from the current state with a small number of action executions (local search space). The agent determines the local search space, searches it, decides

how to move within it, and executes one or more actions along the resulting trajectory. The agent then repeats this process until it reaches a goal state. Real-time heuristic search methods thus do not plan all the way to a goal state which often results in smaller total search times but larger trajectory costs. Most importantly, real-time heuristic search methods can satisfy hard real-time requirements in large state spaces since the sizes of their local search spaces (= their lookaheads) are independent of the sizes of the state spaces and can thus remain small. To focus the search and prevent cycling, they associate heuristics with the states and update them between searches, which accounts for a large chunk of the search time per search episode. In this paper, we describe Real-Time Adaptive A*, a novel real-time heuristic search method. It is a contract anytime method [12] that is able to choose its local search spaces in a very fine-grained way, updates the heuristics of all states in the local search space and is able to do so very quickly. Our experimental results for goal-directed navigation tasks in unknown terrain demonstrate that Real-Time Adaptive A* follows trajectories of smaller cost for given time limits per search episode than a recently proposed real-time heuristic search method [5] because it updates the heuristics more quickly, which allows it to use larger local search spaces and overcompensate for its slightly less informed heuristics. At the same time, Real-Time Adaptive A* is easier to implement.

2. MAIN IDEA

Our main idea is simple but powerful. Assume that one has to perform several A* searches with consistent heuristics in the same state space and with the same goal states but possibly different start states. Adaptive A* makes the heuristics more informed after each A* search in order to speed up future A* searches. We now explain the main idea behind Adaptive A*.

A* [3] is an algorithm for finding cost-minimal paths in state spaces (graphs). For every state s , the user supplies a heuristic $h[s]$ that estimates the goal distance of the state (= the cost of a cost-minimal path from the state to a goal state). The heuristics need to be consistent [10]. For every state s encountered during the search, A* maintains two values: the smallest cost $g[s]$ of any discovered path from the start state s_{curr} to state s (which is initially infinity), and an estimate $f[s] := g[s] + h[s]$ of the distance from the start state s_{curr} via state s to a goal state. A* then operates as follows: It maintains a priority queue, called open list, which initially contains only the start state s_{curr} . A* removes a state s with a smallest f-value from the priority queue. If state s is a goal state, it terminates. Otherwise, it expands the state, meaning that it updates the g-value of each successor state of the state and then inserts those successor states into the open list whose g-value decreased. It then repeats the process. After termination, the g-value of every expanded state

s is equal to the distance from the start state s_{curr} to state s .

We now explain how one can make the heuristics more informed after each A* search in order to speed up future A* searches. Assume that s is a state that was expanded during such an A* search. We can obtain an admissible (= non-overestimating) estimate of its goal distance $gd[s]$ as follows: The distance from the start state s_{curr} to any goal state via state s is equal to the distance from the start state s_{curr} to state s plus the goal distance $gd[s]$ of state s . It clearly can not be smaller than the goal distance $gd[s_{\text{curr}}]$ of the start state s_{curr} . Thus, the goal distance $gd[s]$ of state s is no smaller than the goal distance $gd[s_{\text{curr}}]$ of the start state s_{curr} (= the f-value $f[\bar{s}]$ of the goal state \bar{s} that was about to be expanded when the A* search terminates) minus the distance from the start state s_{curr} to state s (= the g-value $g[s]$ of state s when the A* search terminates).

$$\begin{aligned} g[s] + gd[s] &\geq gd[s_{\text{curr}}] \\ gd[s] &\geq gd[s_{\text{curr}}] - g[s] \\ gd[s] &\geq f[\bar{s}] - g[s] \end{aligned}$$

Consequently, $f[\bar{s}] - g[s]$ provides an admissible estimate of the goal distance $gd[s]$ of state s and can be calculated quickly. More informed heuristics can be obtained by calculating and assigning this difference to every state that was expanded during the A* search and thus is in the closed list when the A* search terminates. (The states in the open list are not updated since the distance from the start state to these states can be smaller than their g-values when the A* search terminates.) We evaluated this idea experimentally in [7]. We now use it to develop a novel real-time heuristic search method, called Real-Time Adaptive A* (RTAA*), which reduces to the case discussed above if its lookahead is infinity.

3. NOTATION

We use the following notation to describe search tasks: S denotes the finite set of states. $s_{\text{curr}} \in S$ denotes the start state of the search, and $GOAL \subset S$ denotes the set of goal states. $A(s)$ denotes the finite set of actions that can be executed in state $s \in S$. $c[s, a] > \epsilon$ (for a given constant $\epsilon > 0$) denotes the cost of executing action $a \in A(s)$ in state $s \in S$, whereas $\text{succ}(s, a) \in S$ denotes the resulting successor state. The action costs c can increase during the search but we require the goal distances of all states to remain bounded from above by some constant.

4. REAL-TIME ADAPTIVE A*

Figure 1 shows the pseudo code of RTAA*. The legend explains the constants, variables, and functions that we will refer to in the following. The variables annotated with [USER] have to be initialized before RTAA* is called. s_{curr} needs to be set to the start state of the agent, c to the initial action costs, and h to the initial heuristics, which need to be consistent for the initial action costs, that is, need to satisfy $h[s] = 0$ for all goal states s and $h[s] \leq h[\text{succ}(s, a)] + c[s, a]$ for all non-goal states s and actions a that can be executed in them [10]. Variables annotated with [A*] are updated during the call to $\text{astar}()$ {03} (= line 3 in the pseudo code), which performs a (forward) A* search guided by the current heuristics from the current state of the agent toward the goal states until a goal state is about to be expanded or $lookahead > 0$ states have been expanded. After the A* search, we require \bar{s} to be the state that was about to be expanded when the A* search terminated. We denote this state with \bar{s} consistently throughout this paper. We require that $\bar{s} = \text{FAILURE}$ if the A* search terminated

constants and functions

S	set of states of the search task, a set of states
$GOAL$	set of goal states, a set of states
$A()$	sets of actions, a set of actions for every state
$\text{succ}()$	successor function, a state for every state-action pair
variables	
$lookahead$	number of states to expand at most, an integer larger than zero
$movements$	number of actions to execute at most, an integer larger than zero
s_{curr}	current state of the agent, a state [USER]
c	current action costs, a float for every state-action pair [USER]
h	current (consistent) heuristics, a float for every state [USER]
g	g-values, a float for every state [A*]
$CLOSED$	closed list of A* (= all expanded states), a set of states [A*]
\bar{s}	state that A* was about to expand when it terminated, a state [A*]

```

procedure realtime_adaptive_astar():
{01} while ( $s_{\text{curr}} \notin GOAL$ ) do
{02}    $lookahead :=$  any desired integer greater than zero;
{03}    $\text{astar}();$ 
{04}   if  $\bar{s} = \text{FAILURE}$  then
{05}     return  $\text{FAILURE};$ 
{06}   for all  $s \in CLOSED$  do
{07}      $h[s] := g[\bar{s}] + h[\bar{s}] - g[s];$ 
{08}    $movements :=$  any desired integer greater than zero;
{09}   while ( $s_{\text{curr}} \neq \bar{s}$  AND  $movements > 0$ ) do
{10}      $a :=$  the action in  $A(s_{\text{curr}})$  on the cost-minimal trajectory from  $s_{\text{curr}}$  to  $\bar{s};$ 
{11}      $s_{\text{curr}} := \text{succ}(s_{\text{curr}}, a);$ 
{12}      $movements := movements - 1;$ 
{13}   for any desired number of times (including zero) do
{14}     increase any desired  $c[s, a]$  where  $s \in S$  and  $a \in A(s);$ 
{15}   if any increased  $c[s, a]$  is on the cost-minimal trajectory from  $s_{\text{curr}}$  to  $\bar{s}$  then
{16}     break;
{17} return  $SUCCESS;$ 

```

Figure 1: Real-Time Adaptive A*

due to an empty open list, in which case there is no finite-cost trajectory from the current state to any goal state and RTAA* thus returns failure {05}. We require $CLOSED$ to contain the states expanded during the A* search and the g-value $g[s]$ to be defined for all generated states s , including all expanded states. We define the f-values $f[s] := g[s] + h[s]$ for these states s . The expanded states s form the local search space, and RTAA* updates their heuristics by setting $h[s] := f[\bar{s}] - g[s] = g[\bar{s}] + h[\bar{s}] - g[s]$ {06-07}. (The heuristics of the other states are not changed.) RTAA* then executes actions along the trajectory found by the A* search until state \bar{s} is reached (or, equivalently, a state is reached that was not expanded or, also equivalently, the local search space is left), $movements > 0$ actions have been executed, or the cost of an action on the trajectory increases {09-16}. It then repeats the process until it reaches a goal state, in which case it returns success {17}.

The values of $lookahead$ and $movements$ determine the behavior of RTAA*. For example, RTAA* performs a single A* search from the start state to a goal state and then moves the agent along the trajectory found by the A* search to the goal state if it always chooses infinity for $lookahead$ and $movements$ and no action costs increase.

We now prove several important properties of RTAA* that hold no matter how it chooses its values of $lookahead$ and $movements$. We make use of the following known properties of A* searches with consistent heuristics: First, they expand every state at most once. Second, the g-values of every expanded state and state \bar{s} are equal to the distance from the start state to state s and state \bar{s} , respectively. Thus, one knows cost-minimal trajectory from the start state to all expanded states and state \bar{s} . Third, the f-values of the series of expanded states over time are monotonically nondecreasing. Thus, $f[s] \leq f[\bar{s}]$ for all expanded states s and $f[\bar{s}] \leq f[s]$ for all generated states s that remained unexpanded.

THEOREM 1. *The heuristics of the same state are monotonically nondecreasing over time and thus indeed become more informed over time.*

Proof: Assume that the heuristic of state s is updated on line {07}. Then, state s was expanded and it thus holds that $f[s] \leq f[\bar{s}]$. Consequently, $h[s] = f[s] - g[s] \leq f[\bar{s}] - g[s] = g[\bar{s}] + h[\bar{s}] - g[s]$ and the update cannot decrease the heuristic of state s since it changes the heuristic from $h[s]$ to $g[\bar{s}] + h[\bar{s}] - g[s]$. ■

THEOREM 2. *The heuristics remain consistent.*

Proof: We prove this property by induction on the number of A* searches. The initial heuristics are provided by the user and consistent. It thus holds that $h[s] = 0$ for all goal states s . This continues to hold since goal states are not expanded and their heuristics thus not updated. (Even if RTAA* updated the heuristic of state \bar{s} , it would leave the h-value of that state unchanged since $f[\bar{s}] - g[\bar{s}] = g[\bar{s}] + h[\bar{s}] - g[\bar{s}] = h[\bar{s}]$. Thus, the heuristics of goal states would remain zero even in that case.) It also holds that $h[s] \leq h[\text{succ}(s, a)] + c[s, a]$ for all non-goal states s and actions a that can be executed in them. Assume that some action costs increase on lines {13-14}. Let c denote the action costs before all increases and c' denote the action costs after all increases. Then, $h[s] \leq h[\text{succ}(s, a)] + c[s, a] \leq h[\text{succ}(s, a)] + c'[s, a]$ and the heuristics thus remain consistent. Now assume that the heuristics are updated on lines {06-07}. Let h denote the heuristics before all updates and h' denote the heuristics after all updates. We distinguish three cases:

- First, both s and $\text{succ}(s, a)$ were expanded, which implies that $h'[s] = g[\bar{s}] + h[\bar{s}] - g[s]$ and $h'[\text{succ}(s, a)] = g[\bar{s}] + h[\bar{s}] - g[\text{succ}(s, a)]$. Also, $g[\text{succ}(s, a)] \leq g[s] + c[s, a]$ since the A* search discovers a trajectory from the current state via state s to state $\text{succ}(s, a)$ of cost $g[s] + c[s, a]$ during the expansion of state s . Thus, $h'[s] = g[\bar{s}] + h[\bar{s}] - g[s] \leq g[\bar{s}] + h[\bar{s}] - g[\text{succ}(s, a)] + c[s, a] = h'[\text{succ}(s, a)] + c[s, a]$.
- Second, s was expanded but $\text{succ}(s, a)$ was not, which implies that $h'[s] = g[\bar{s}] + h[\bar{s}] - g[s]$ and $h'[\text{succ}(s, a)] = h[\text{succ}(s, a)]$. Also, $g[\text{succ}(s, a)] \leq g[s] + c[s, a]$ for the same reason as in the first case, and $f[\bar{s}] \leq f[\text{succ}(s, a)]$ since state $\text{succ}(s, a)$ was generated but not expanded. Thus, $h'[s] = g[\bar{s}] + h[\bar{s}] - g[s] = f[\bar{s}] - g[s] \leq f[\text{succ}(s, a)] - g[s] = g[\text{succ}(s, a)] + h[\text{succ}(s, a)] - g[s] = g[\text{succ}(s, a)] + h'[\text{succ}(s, a)] - g[s] \leq g[\text{succ}(s, a)] + h'[\text{succ}(s, a)] - g[\text{succ}(s, a)] + c[s, a] = h'[\text{succ}(s, a)] + c[s, a]$.
- Third, s was not expanded, which implies that $h'[s] = h[s]$. Also, $h[\text{succ}(s, a)] \leq h'[\text{succ}(s, a)]$ since the heuristics of the same state are monotonically nondecreasing over time. Thus, $h'[s] = h[s] \leq h[\text{succ}(s, a)] + c[s, a] \leq h'[\text{succ}(s, a)] + c[s, a]$.

Thus, $h'[s] \leq h'[\text{succ}(s, a)] + c[s, a]$ in all three cases and the heuristics thus remain consistent. ■

THEOREM 3. *The agent reaches a goal state.*

Proof: Assume that the heuristics are updated on lines {06-07}. Let h denote the heuristics of RTAA* before all updates and h' denote the heuristics after all updates. The heuristics of the same state are monotonically nondecreasing over time, according to Theorem 1. Assume that the agent moves from its current state s to some state s' (with $s \neq s'$) along a cost-minimal trajectory from state s to state \bar{s} . It holds that $h'[s] = f[\bar{s}] - g[s] = f[\bar{s}]$ since state s is the start of the search and its g-value is thus zero and since it was expanded and its heuristic was thus updated. Furthermore, it holds that $h'[s'] = f[\bar{s}] - g[s']$ since either state s' was expanded and its heuristic was thus updated or $s' = \bar{s}$ and then $h'[s'] = h'[\bar{s}] = h[\bar{s}] = f[\bar{s}] - g[\bar{s}] = f[\bar{s}] - g[s']$. Thus, after the agent moved from state s to state s' and its current state thus changed from state s to state s' , the heuristic of the current state decreased by $h'[s] - h'[s'] = f[\bar{s}] - (f[\bar{s}] - g[s']) = g[s']$ and the sum of the heuristics of all states but its current state thus increased by $g[s']$, which is bounded from below by a positive constant since $s \neq s'$ and we assumed that all action costs are bounded from below by a positive constant. Thus, the sum of the heuristics of all states but the current state of the agent increases over time beyond any bound if the agent does not reach a goal state. At the same time, the heuristics remain consistent, according to Theorem 2 (since consistent heuristics are admissible), and are thus no larger than the goal distances which we assumed to be bounded from above, which is a contradiction. Thus, the agent is guaranteed to reach a goal state. ■

THEOREM 4. *If the agent is reset into the start state whenever it reaches a goal state then the number of times that it does not follow a cost-minimal trajectory from the start state to a goal state is bounded from above by a constant if the cost increases are bounded from below by a positive constant.*

Proof: Assume for now that the cost increases leave the goal distances of all states unchanged. Under this assumption, it is easy to see that the agent follows a cost-minimal trajectory from the start state to a goal state if it follows a trajectory from the start state to a goal state where the h-value of every state is equal to its goal distance. If the agent does not follow such a trajectory, then it transitions at least once from a state s whose h-value is not equal to its goal distance to a state s' whose h-value is equal to its goal distance since it reaches a goal state according to Theorem 3 and the h-value of the goal state is zero since the heuristics remain consistent according to Theorem 2. We now prove that the h-value of state s is then set to its goal distance. When the agent executes some action $a \in A(s)$ in state s and transitions to state s' , then state s is a parent of state s' in the A* search tree produced during the last call of `astar()` and it thus holds that (1) state s was expanded during the last call of `astar()`, (2) either state s' was also expanded during the last call of `astar()` or $s' = \bar{s}$, (3) $g[s'] = g[s] + c[s, a]$. Let h denote the heuristics before all updates and h' denote the heuristics after all updates. Then, $h'[s] = f[\bar{s}] - g[s]$ and $h'[s'] = f[\bar{s}] - g[s'] = gd[s']$. The last equality holds because we assumed that the h-value of state s' was equal to its goal distance and thus can no longer change since it could only increase according to Theorem 1 but would then make the heuristics inadmissible and thus inconsistent, which is impossible according to Theorem 2. Consequently, $h'[s] = f[\bar{s}] - g[s] = gd[s'] + g[s'] - g[s] = gd[s'] + c[s, a] \geq gd[s]$, proving that $h'[s] = gd[s]$ since a larger h-value would make the heuristics inadmissible and thus inconsistent, which is impossible according to Theorem 2. Thus, the h-value of state s is indeed set to its goal

distance. After the h-value of state s is set to its goal distance, the h-value can no longer change since it could only increase according to Theorem 1 but would then make the heuristics inadmissible and thus inconsistent, which is impossible according to Theorem 2. Since the number of states is finite, it can happen only a bounded number of times that the h-value of a state is set to its goal distance. Thus, the number of times that the agent does not follow a cost-minimal trajectory from the start state to a goal state is bounded. The theorem then follows since the number of times that a cost increase does not leave the goal distances of all states unchanged is bounded since we assumed that the cost increases are bounded from below by a positive constant but the goal distances are bounded from above. After each such change, the number of times that the agent does not follow a cost-minimal trajectory from the start state to a goal state is bounded. ■

5. RELATIONSHIP TO LRTA*

RTAA* is similar to a version of Learning Real-Time A* recently proposed in [5], an extension of the original Learning Real-Time A* algorithm [9] to larger lookaheads. For simplicity, we refer to this particular version of Learning Real-Time A* as LRTA*. RTAA* and LRTA* differ only in how they update the heuristics after an A* search. LRTA* replaces the heuristic of each expanded state with the sum of the distance from the state to a generated but unexpanded state s and the heuristic of state s , minimized over all generated but unexpanded states s . (The heuristics of the other states are not changed.) Let \bar{h}' denote the heuristics after all updates. Then, the heuristics of LRTA* after the updates satisfy the following system of equations for all expanded states s :

$$\bar{h}'[s] = \min_{a \in A(s)} (c[s, a] + \bar{h}'[\text{succ}(s, a)])$$

The properties of LRTA* are similar to the ones of RTAA*. For example, its heuristics for the same state are monotonically nondecreasing over time and remain consistent, and the agent reaches a goal state. We now prove that LRTA* and RTAA* behave exactly the same if their lookahead is one and they break ties in the same way. They can behave differently for larger lookaheads, and we give an informal argument why the heuristics of LRTA* tend to be more informed than the ones of RTAA* with the same lookaheads. On the other hand, it takes LRTA* more time to update the heuristics and it is more difficult to implement, for the following reason: LRTA* performs one search to determine the local search space and a second search to determine how to update the heuristics of the states in the local search space since it is unable to use the results of the first search for this purpose, as explained in [5]. Thus, there is a trade-off between the total search time and the cost of the resulting trajectory, and we need to compare both search methods experimentally to understand this trade-off better.

THEOREM 5. *RTAA* with lookahead one behaves exactly like LRTA* with the same lookahead if they break ties in the same way.*

Proof: We show the property by induction on the number of A* searches. The heuristics of both search methods are initialized with the heuristics provided by the user and are thus identical before the first A* search. Now consider any A* search. The A* searches of both search methods are identical if they break ties in the same way. Let \bar{s} be the state that was about to be expanded when their A* searches terminated. Let h denote the heuristics of RTAA* before

all updates and h' denote the heuristics after all updates. Similarly, let \bar{h} denote the heuristics of LRTA* before all updates and \bar{h}' denote the heuristics after all updates. Assume that $h[s] = \bar{h}[s]$ for all states s . We show that $h'[s] = \bar{h}'[s]$ for all states s . Both search methods expand only the current state s of the agent and thus update only the heuristic of this one state. Since $s \neq \bar{s}$, it holds that $h'[s] = g[\bar{s}] + h[\bar{s}] - g[s] = g[\bar{s}] + h[\bar{s}]$ and $\bar{h}'[s] = \min_{a \in A(s)} (c[s, a] + \bar{h}'[\text{succ}(s, a)]) = \min_{a \in A(s)} (g[\text{succ}(s, a)] + \bar{h}'[\text{succ}(s, a)]) = \min_{a \in A(s)} (g[\text{succ}(s, a)] + \bar{h}[\text{succ}(s, a)]) = g[\bar{s}] + \bar{h}[\bar{s}] = g[\bar{s}] + h[\bar{s}]$. Thus, both search methods set the heuristic of the current state to the same value and then move to state \bar{s} . Notice that $\text{lookahead} = 1$ implies without loss of generality that $\text{movements} = 1$. Consequently, they behave exactly the same. ■

We now give an informal argument why the heuristics of LRTA* with lookaheads larger than one tend to be more informed than the ones of RTAA* with the same lookahead (if both real-time heuristic search methods use the same value of *movements*). This is not a proof but gives some insight into the behavior of the two search methods. Assume that both search methods are in the same state and break ties in the same way. Let h denote the heuristics of RTAA* before all updates and h' denote the heuristics after all updates. Similarly, let \bar{h} denote the heuristics of LRTA* before all updates and \bar{h}' denote the heuristics after all updates. Assume that $h[s] = \bar{h}[s]$ for all states s . We now prove that $h'[s] \leq \bar{h}'[s]$ for all states s . The A* searches of both search methods are identical if they break ties in the same way. Thus, they expand the same states and thus also update the heuristics of the same states. We now show that the heuristics h' cannot be consistent if $h'[s] > \bar{h}'[s]$ for at least one state s . Assume that $h'[s] > \bar{h}'[s]$ for at least one state s . Pick a state s with the smallest $\bar{h}'[s]$ for which $h'[s] > \bar{h}'[s]$ and pick an action a with $a = \arg \min_{a \in A(s)} (c[s, a] + h'[\text{succ}(s, a)])$. State s must have been expanded since $h[s] = \bar{h}[s]$ but $h'[s] > \bar{h}'[s]$. Then, it holds that $\bar{h}'[s] = c[s, a] + \bar{h}'[\text{succ}(s, a)]$. Since $\bar{h}'[s] = c[s, a] + \bar{h}'[\text{succ}(s, a)] > h'[\text{succ}(s, a)]$ and state s is a state with the smallest $\bar{h}'[s]$ for which $h'[s] > \bar{h}'[s]$, it must be the case that $h'[\text{succ}(s, a)] \leq \bar{h}'[\text{succ}(s, a)]$. Put together, it holds that $h'[s] > \bar{h}'[s] = c[s, a] + \bar{h}'[\text{succ}(s, a)] \geq c[s, a] + h'[\text{succ}(s, a)]$. This means that the heuristics h' are inconsistent but we have earlier proved already that they remain consistent, which is a contradiction. Consequently, it holds that $h'[s] \leq \bar{h}'[s]$ for all states s . Notice that this proof does not imply that the heuristics of LRTA* always dominate the ones of RTAA* since the search methods can move the agent to different states and then update the heuristics of different states, but it suggests that the heuristics of LRTA* with lookaheads larger than one tend to be more informed than the ones of RTAA* with the same lookaheads and thus that the trajectories of LRTA* tend to be of smaller cost than the trajectories of RTAA* with the same lookaheads (if both real-time heuristic search methods use the same value of *movements*).

In the remainder of the paper, we assume that the agents always choose the same constant for *lookahead*, which is an external parameter, and always use infinity for *movements*, both for LRTA* and RTAA*.

6. APPLICATION

Real-time heuristic search methods are often used as alternative to traditional search methods for solving offline search tasks [9]. We, however, apply RTAA* to goal-directed navigation in unknown terrain, a search task that requires agents to execute actions in real time. Characters in real-time computer games, for example, of-



Figure 2: Total Annihilation

ten do not know the terrain in advance but automatically observe it within a certain range around them and then remember it for future use. Figure 2 shows an example. To make these agents easy to control, the users can click on some position in known or unknown terrain and the agents then move autonomously to this position. If the agents observe during execution that their current trajectory is blocked, then they need to search for another plan. The searches need to be fast since the agents need to move smoothly even if the processor is slow, the other game components use most of the available processor cycles and there are a large number of agents that all have to search repeatedly. Thus, there is a time limit per A* search, which suggests that real-time heuristic search methods are a good fit for our navigation tasks. To apply them, we discretize the terrain into cells that are either blocked or unblocked, a common practice in the context of real-time computer games [1]. The agents initially do not know which cells are blocked but use a navigation strategy from robotics [8]: They assume that cells are unblocked unless they have the cells already observed to be blocked (freespace assumption). They always know which (unblocked) cells they are in, observe the blockage status of their four neighboring cells, raise the action costs of actions that enter the newly observed blocked cells, if any, from one to infinity, and then move to any one of the unblocked neighboring cells with cost one. We therefore use the Manhattan distances as consistent heuristic estimates of the goal distances. The task of the agents is to move to the given goal cell, which we assume to be possible.

7. ILLUSTRATION

Figure 3 shows a simple goal-directed navigation task in unknown terrain that we use to illustrate the behavior of RTAA*. Black squares are blocked. All cells have their initial heuristic in the lower left corner. We first compare RTAA* with lookahead infinity to LRTA* with the same lookahead and forward A* searches. All search methods start a new search episode (= run another search) when the cost of an action on their current trajectory increases and break ties between cells with the same f-values in favor of cells with larger g-values and remaining ties in the following order, from highest to lowest priority: right, down, left and up. We do this since we believe that systematic rather than random tie-breaking helps the readers to understand the behavior of the different search methods better since all search methods then follow the same trajectories. Figures 4, 5, and 6 show the agent as a small black circle.

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4	3	2	1
4	3	2	1	goal 0

Figure 3: Example

8	7	6	5	4	9	8	7	6	5	4	9	8	7
7	6	5	4	3	10	9	8	7	6	5	4	9	8
6	5	4	3	2	10	9	8	7	6	5	4	9	8
5	4	3	2	1	10	9	8	7	6	5	4	9	8
4	3	2	1	0	10	9	8	7	6	5	4	9	8

Figure 4: Forward A* Searches

8	7	6	5	4	9	8	7	6	5	4	9	8	7
7	6	5	4	3	10	9	8	7	6	5	4	9	8
6	5	4	3	2	10	9	8	7	6	5	4	9	8
5	4	3	2	1	10	9	8	7	6	5	4	9	8
4	3	2	1	0	10	9	8	7	6	5	4	9	8

Figure 5: RTAA* with $lookahead = \infty$

8	7	6	5	4	9	8	7	6	5	4	9	8	7
7	6	5	4	3	10	9	8	7	6	5	4	9	8
6	5	4	3	2	10	9	8	7	6	5	4	9	8
5	4	3	2	1	10	9	8	7	6	5	4	9	8
4	3	2	1	0	10	9	8	7	6	5	4	9	8

Figure 6: LRTA* with $lookahead = \infty$

The arrows show the planned trajectories from the current cell of the agent to the goal cell, which is in the lower right corner. Cells that the agent has already observed to be blocked are black. All other cells have their heuristic in the lower left corner. Generated cells have their g-value in the upper left corner and their f-value in the upper right corner. Expanded cells are grey and, for RTAA* and LRTA*, have their updated heuristics in the lower right corner, which makes it easy for the readers to compare them to the heuristics before the update in the lower left corner. Notice that forward A* searches, RTAA* with lookahead infinity and LRTA* with the same lookahead follow the same trajectories if they break ties in

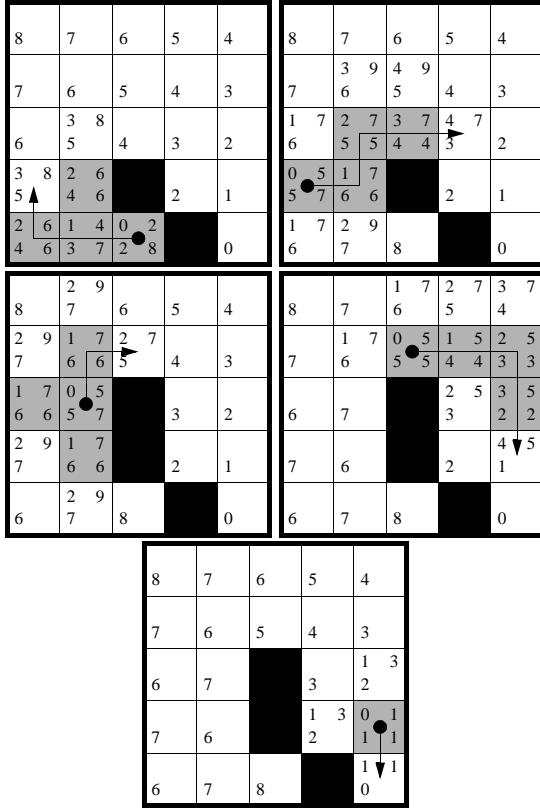


Figure 7: RTAA* with lookahead = 4

the same way. They differ only in the number of cell expansions, which is larger for forward A* searches (23) than for RTAA* (20) and larger for RTAA* (20) than for LRTA* (19). The first property is due to RTAA* and LRTA* updating the heuristics while forward A* searches do not. Thus, forward A* searches fall prey to the local minimum in the heuristic value surface and thus expand the three leftmost cells in the lowest row a second time, while RTAA* and LRTA* avoid these cell expansions. The second property is due to some updated heuristics of LRTA* being larger than the ones of RTAA*. Notice, however, that most updated heuristics are identical, although this is not guaranteed in general. We also compare RTAA* with lookahead four to RTAA* with lookahead infinity. Figures 5 and 7 show that decreasing the lookahead of RTAA* increases the trajectory cost (from 10 to 12) but decreases the number of cell expansions (from 20 to 17) because smaller lookaheads imply that less information is used during each search episode. (Notice that the last search episode of RTAA* with lookahead four expands only one cell since the goal cell is about to be expanded next.) We now perform systematic experiments to see whether they confirm the trends shown in our examples.

8. EXPERIMENTAL RESULTS

We now run experiments to compare RTAA* to forward A* searches and LRTA*, as in the previous section. All three search methods search forward. We also compare RTAA* to two search methods that search backward, namely backward A* searches and D* Lite [6] (an incremental version of backward A* searches that is similar to but simpler than D* [11]), which is possible because

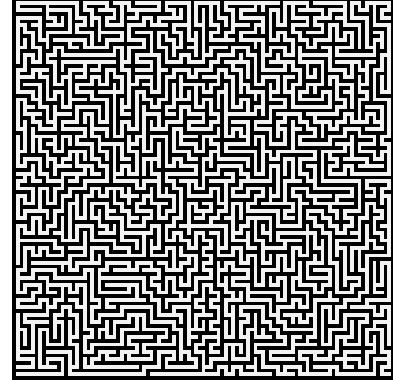


Figure 8: Test Terrain: Mazes

there is only one goal cell.¹ All search methods use binary heaps as priority queues and now break ties between cells with the same f-values in favor of cells with larger g-values (which is known to be a good tie-breaking strategy) and remaining ties randomly.

We perform experiments on a SUN PC with a 2.4 GHz AMD Opteron Processor 150 in randomly generated four-connected mazes of size 151×151 that are solvable. Their corridor structure is generated with depth-first search. The start and goal cells are chosen randomly. Figure 8 shows an example (of smaller size than used in the experiments). We use the Manhattan distances as heuristics. The performance measures in Table 1 are averaged over the same 2500 grids. We show the standard deviation of the mean for three key performance measures in square brackets to demonstrate the statistical significance of our results, namely the total number of cell expansions and the total search time on one hand and the resulting trajectory cost on the other hand. We first verify the properties suggested in the previous sections:

- RTAA* with lookahead infinity, LRTA* with the same lookahead, D* Lite and forward and backward A* searches follow the same trajectories if they break ties in the same way and their trajectory costs are indeed approximately equal. (The slight differences are due to remaining ties being broken randomly.) The total number of cell expansions and the total search time are indeed larger for forward A* searches than RTAA*, and larger for RTAA* than LRTA*, as suggested in “Illustration.”
- Decreasing the lookahead of RTAA* indeed increases the trajectory cost but initially decreases the total number of cell expansions and the total search time, as suggested in “Il-

¹Notice that the total number of cell expansions and total search time of forward A* searches are significantly smaller than the ones of backward A* searches. The big impact of the search direction can be explained as follows: The agent observes blocked cells close to itself. Thus, the blocked cells are close to the start of the search in the early phases of forward A* searches, but close to the goal of the search in the early phases of backward A* searches. The closer the blocked cells are to the start of the search, the more cells there are for which the Manhattan distances are perfectly informed and thus the faster A* searches are that break ties between cells with the same f-values in favor of cells with larger g-values since they expand only states along a cost-minimal trajectory from cells for which the Manhattan distances are perfectly informed to the goal of the search.

Table 1: Experiments in Mazes

(a) = lookahead (= bound on the number of cell expansions per search episode), (b) = total number of cell expansions (until the agent reaches the goal cell) [in square brackets: standard deviation of the mean], (c) = total number of search episodes (until the agent reaches the goal cell), (d) = trajectory cost (= total number of action executions until the agent reaches the goal cell) [in square brackets: standard deviation of the mean], (e) = number of action executions per search episode, (f) = total search time (until the agent reaches the goal cell) in microseconds [in square brackets: standard deviation of the mean], (g) = search time per search episode in microseconds, (h) = search time per action execution in microseconds, (i) = increase of heuristics per search episode and expanded cell (= per update) = $h'[s] - h[s]$ averaged over all search episodes and expanded cells s

(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)
Real-Time Adaptive A* (RTAA*)								
1	248537.79 [5454.11]	248537.79	248537.79 [5454.11]	1.00	48692.17 [947.08]	0.20	0.20	1.00
9	104228.97 [2196.69]	11583.50	56707.84 [1248.70]	4.90	23290.36 [360.77]	2.01	0.41	2.40
17	85866.45 [1701.83]	5061.92	33852.77 [774.35]	6.69	22100.20 [301.99]	4.37	0.65	3.10
25	89257.66 [1593.02]	3594.51	26338.21 [590.00]	7.33	24662.64 [310.42]	6.86	0.94	3.30
33	96839.84 [1675.46]	2975.65	22021.81 [521.77]	7.40	27994.45 [350.79]	9.41	1.27	3.28
41	105702.99 [1699.20]	2639.59	18628.66 [435.06]	7.06	31647.50 [382.30]	11.99	1.70	3.08
49	117035.65 [1806.59]	2473.55	16638.27 [390.09]	6.73	35757.69 [428.17]	14.46	2.15	2.90
57	128560.04 [1939.38]	2365.94	15366.63 [361.95]	6.49	39809.02 [477.72]	16.83	2.59	2.79
65	138640.02 [2019.98]	2270.38	14003.74 [314.38]	6.17	43472.99 [517.36]	19.15	3.10	2.63
73	150254.51 [2176.68]	2224.29	13399.01 [309.72]	6.02	47410.44 [567.88]	21.31	3.54	2.57
81	160087.23 [2269.20]	2172.94	12283.65 [270.03]	5.65	50932.60 [608.94]	23.44	4.15	2.39
89	172166.56 [2436.73]	2162.75	12078.40 [261.16]	5.58	54874.88 [663.96]	25.37	4.54	2.37
∞	642823.02 [20021.00]	1815.92	5731.72 [81.72]	3.16	226351.59 [7310.53]	124.65	39.49	4.22
Learning Real-Time A* (LRTA*)								
1	248537.79 [5454.11]	248537.79	248537.79 [5454.11]	1.00	67252.19 [1354.67]	0.27	0.27	1.00
9	87613.37 [1865.31]	9737.38	47290.61 [1065.07]	4.86	27286.14 [437.09]	2.80	0.58	2.93
17	79312.59 [1540.44]	4676.76	30470.32 [698.08]	6.52	29230.15 [409.96]	6.25	0.96	3.61
25	82850.86 [1495.61]	3338.86	23270.38 [551.75]	6.97	34159.80 [450.49]	10.23	1.47	3.74
33	92907.75 [1548.37]	2858.19	20015.55 [472.86]	7.00	40900.68 [516.47]	14.31	2.04	3.71
41	102787.86 [1619.33]	2570.83	17274.12 [403.65]	6.72	47559.60 [587.96]	18.50	2.75	3.54
49	113139.63 [1716.88]	2396.66	15398.47 [360.45]	6.42	54324.06 [665.02]	22.67	3.53	3.38
57	125013.41 [1829.10]	2307.68	14285.14 [328.39]	6.19	61590.97 [744.33]	26.69	4.31	3.25
65	133863.67 [1956.49]	2201.60	13048.50 [300.69]	5.93	67482.95 [829.44]	30.65	5.17	3.12
73	146549.69 [2080.76]	2181.76	12457.92 [277.60]	5.71	74868.92 [909.31]	34.32	6.01	3.02
81	157475.45 [2209.65]	2150.04	11924.96 [262.61]	5.55	81469.32 [989.84]	37.89	6.83	2.95
89	166040.29 [2355.33]	2102.91	11324.72 [246.94]	5.39	86883.98 [1077.54]	41.32	7.67	2.88
∞	348072.76 [7021.57]	1791.19	5611.09 [80.43]	3.13	203645.42 [3782.37]	113.69	36.29	8.20
D* Lite								
-	47458.83 [581.03]	1776.24	5637.46 [77.03]	3.17	37291.83 [378.20]	20.99	6.62	-
Forward A* Search								
-	1857468.48 [68324.90]	1732.07	5354.26 [76.91]	3.09	544065.45 [21565.61]	314.11	101.61	-
Backward A* Search								
-	5245087.82 [93697.15]	1795.72	5535.05 [77.09]	3.08	1698163.04 [31051.10]	945.67	306.80	-

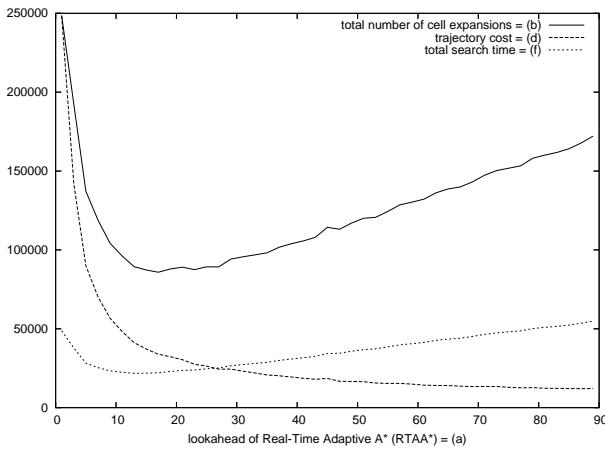


Figure 9: Performance of Real-Time Adaptive A*

lustration.” Increasing the trajectory cost increases the total number of search episodes. If the lookahead is already small and continues to decrease, then eventually the speed with which the total number of search episodes increases is larger than the speed with which the lookahead and the time per search episode decreases, so that the number of cell expansions and the total search time increase again, as the graphs in Figure 9 show. (The graph also shows that the to-

tal number of cell expansions and the total search time are proportional, as expected.)

- RTAA* with lookaheads larger than one and smaller than infinity indeed increases the heuristics less than LRTA* with the same lookaheads per update, as suggested in “Relationship to LRTA*.” Consequently, its trajectory costs and total number of cell expansions are larger than the ones of LRTA* with the same lookaheads. However, it updates the heuristics much faster than LRTA* with the same lookaheads, resulting in smaller total search times.
- RTAA* with lookahead one and LRTA* with the same lookahead follow the same trajectories and update the same states if they break ties in the same way and their trajectory costs and total number of cell expansions are indeed approximately equal, as suggested in “Relationship to LRTA*.”

One advantage of RTAA* is that its total planning time with a carefully chosen lookahead is smaller than that of all other search methods, although its trajectory costs then are not the smallest ones. This is important for applications where planning is slow but actions can be executed fast. However, the advantage of RTAA* over the other search methods for our particular application is a different one: Remember that there is a time limit per search episode so that the characters move smoothly. If this time limit is larger than 20.99 microseconds, then one should use D* Lite for the search because the resulting trajectory costs are smaller than the ones of all other

search methods whose search time per search episode is no larger than 20.99. (The table shows that the trajectory costs of forward A* search are smaller but, as argued before, this difference is due to noise.) However, if the time limit is smaller than 20.99 microseconds, then one has to use a real-time heuristics search method. In this case, one should use RTAA* rather than LRTA*. Assume, for example, that the time limit is 20.00 microseconds. Then one can use either RTAA* with a lookahead of 67, resulting in a trajectory cost of 13657.31, or LRTA* with a lookahead of 43, resulting in a trajectory cost of 16814.49. Thus, the trajectory cost of LRTA* is about 23 percent higher than the one of RTAA*, which means that RTAA* improves the state of the art in real-time heuristic search. A similar argument holds if the search time is amortized over the number of action executions, in which case there is a time limit per action execution. If this time limit is larger than 6.62 microseconds, then one should use D* Lite for the search because the resulting trajectory costs are smaller than the ones of all other search methods whose search time per action execution is no larger than 6.62. However, if the time limit is smaller than 6.62 microseconds, then one has to use a real-time heuristics search method. In this case, one should use RTAA* rather than LRTA*. Assume, for example, that the time limit is 4.00 microseconds. Then one can use either RTAA* with a lookahead of 79, resulting in a trajectory cost of 12699.99, or LRTA* with a lookahead of 53, resulting in a trajectory cost of 14427.80. Thus, the trajectory cost of LRTA* is about 13 percent higher than the one of RTAA*, which again means that RTAA* improves the state of the art in real-time heuristic search.

9. CONCLUSIONS

In this paper, we developed Real-Time Adaptive A* (RTAA*). This real-time heuristic search method is able to choose its local search spaces in a fine-grained way. It updates the values of all states in its local search spaces and can do so very quickly. Our experimental results for goal-directed navigation tasks in unknown terrain demonstrated that this property allows RTAA* to move to the goal with smaller total search times than a variety of tested alternative search methods. Furthermore, we showed that RTAA* follows trajectories of smaller cost for given time limits per search episode than a recently proposed real-time heuristic search method because it updates the heuristics more quickly, which allows it to use larger local search spaces and overcompensate for its slightly less informed heuristics. It is future work to extend RTAA* to inconsistent heuristics, compare it to real-time heuristic search methods besides LRTA* and combine the idea behind RTAA* with other ideas of enhancing real-time heuristic search methods, for example, the ones described in [2] and [4]. In particular, it would be interesting to study a hierarchical version of RTAA*.

Acknowledgments

This research has been partly supported by an NSF award to Sven Koenig under contract IIS-0350584. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, companies or the U.S. government.

10. REFERENCES

- [1] M. Bjornsson, M. Enzenberger, R. Holte, J. Schaeffer, and P. Yap. Comparison of different abstractions for pathfinding on maps. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1511–1512, 2003.
- [2] V. Bulitko and G. Lee. Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research*, page (in press), 2005.
- [3] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 2:100–107, 1968.
- [4] T. Ishida. *Real-Time Search for Learning Autonomous Agents*. Kluwer Academic Publishers, 1997.
- [5] S. Koenig. A comparison of fast search methods for real-time situated agents. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*, pages 864–871, 2004.
- [6] S. Koenig and M. Likhachev. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence*, pages 476–483, 2002.
- [7] S. Koenig and M. Likhachev. Adaptive A* [poster abstract]. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*, pages 1311–1312, 2005.
- [8] S. Koenig, C. Tovey, and Y. Smirnov. Performance bounds for planning in unknown terrain. *Artificial Intelligence*, 147:253–279, 2003.
- [9] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.
- [10] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.
- [11] A. Stentz. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659, 1995.
- [12] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, Computer Science Department, University of California at Berkeley, Berkeley (California), 1993.

Monte Carlo Localization: Efficient Position Estimation for Mobile Robots

Dieter Fox, Wolfram Burgard[†], Frank Dellaert, Sebastian Thrun

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

[†]Computer Science Department III
University of Bonn
Bonn, Germany

Abstract

This paper presents a new algorithm for mobile robot localization, called Monte Carlo Localization (MCL). MCL is a version of Markov localization, a family of probabilistic approaches that have recently been applied with great practical success. However, previous approaches were either computationally cumbersome (such as grid-based approaches that represent the state space by high-resolution 3D grids), or had to resort to extremely coarse-grained resolutions. Our approach is computationally efficient while retaining the ability to represent (almost) arbitrary distributions. MCL applies sampling-based methods for approximating probability distributions, in a way that places computation “where needed.” The number of samples is adapted on-line, thereby invoking large sample sets only when necessary. Empirical results illustrate that MCL yields improved accuracy while requiring an order of magnitude less computation when compared to previous approaches. It is also much easier to implement.

Introduction

Throughout the last decade, sensor-based localization has been recognized as a key problem in mobile robotics (Cox 1991; Borenstein, Everett, & Feng 1996). Localization is a version of on-line temporal state estimation, where a mobile robot seeks to estimate its position in a global coordinate frame. The localization problem comes in two flavors: *global localization* and *position tracking*. The second is by far the most-studied problem; here a robot knows its initial position and “only” has to accommodate small errors in its odometry as it moves. The global localization problem involves a robot which is not told its initial position; hence, it has to solve a much more difficult localization problem, that of estimating its position from scratch (this is sometimes referred to as the *hijacked robot problem* (Engelson 1994)). The ability to localize itself—both locally and globally—played an important role in a collection of recent mobile robot applications (Burgard *et al.* 1998a; Endres, Feiten, & Lawitzky 1998; Kortenkamp, Bonasso, & Murphy 1997).

While the majority of early work focused on the tracking problem, recently several researchers have developed

what is now a highly successful family of approaches capable of solving both localization problems: *Markov localization* (Nourbakhsh, Powers, & Birchfield 1995; Simmons & Koenig 1995; Kaelbling, Cassandra, & Kurien 1996; Burgard *et al.* 1996). The central idea of Markov localization is to represent the robot’s belief by a probability distribution over possible positions, and use Bayes rule and convolution to update the belief whenever the robot senses or moves. The idea of probabilistic state estimation goes back to Kalman filters (Gelb 1974; Smith, Self, & Cheeseman 1990), which use multivariate Gaussians to represent the robot’s belief. Because of the restrictive nature of Gaussians (they can basically represent one hypothesis only annotated by its uncertainty) Kalman-filters usually are only applied to position tracking. Markov localization employs discrete, but *multi-modal* representations for representing the robot’s belief, hence can solve the global localization problem. Because of the real-valued and multi-dimensional nature of kinematic state spaces these approaches can only *approximate* the belief, and accurate approximation usually requires prohibitive amounts of computation and memory.

In particular, *grid-based* methods have been developed that approximate the kinematic state space by fine-grained piecewise constant functions (Burgard *et al.* 1996). For reasonably-sized environments, these approaches often require memory in the excess of 100MB, and high-performance computing. At the other extreme, various researchers have resorted to coarse-grained *topological* representations, whose granularity is often an order of magnitude lower than that of the grid-based approach. When high resolution is needed (see e.g., (Fox *et al.* 1998), who uses localization to avoid collisions with static obstacles that cannot be detected by sensors), such approaches are inapplicable.

In this paper we present Monte Carlo Localization (in short: MCL). Monte Carlo methods were introduced in the Seventies (Handschin 1970), and recently rediscovered independently in the target-tracking (Gordon, Salmond, & Smith 1993), statistical (Kitagawa 1996) and computer vision literature (Isard & Blake 1998), and they have also been applied in dynamic probabilistic networks (Kanazawa, Koller, & Russell 1995). MCL uses fast sampling techniques to represent the robot’s belief. When the robot moves or senses, impor-

tance re-sampling (Rubin 1988) is applied to estimate the posterior distribution. An adaptive sampling scheme (Koller & Fratkina 1998), which determines the number of samples on-the-fly, is employed to trade-off computation and accuracy. As a result, MCL uses many samples during global localization when they are most needed, whereas the sample set size is small during tracking, when the position of the robot is approximately known.

By using a sampling-based representation, MCL has several key advantages over earlier work in the field:

1. In contrast to existing Kalman filtering based techniques, it is able to represent multi-modal distributions and thus can *globally* localize a robot.
2. It drastically reduces the amount of memory required compared to grid-based Markov localization and can integrate measurements at a considerably higher frequency.
3. It is more *accurate* than Markov localization with a fixed cell size, as the state represented in the samples is not discretized.
4. It is much easier to implement.

Markov Localization

This section briefly outlines the basic Markov localization algorithm upon which our approach is based. The key idea of Markov localization—which has recently been applied with great success at various sites (Nourbakhsh, Powers, & Birchfield 1995; Simmons & Koenig 1995; Kaelbling, Cassandra, & Kurien 1996; Burgard *et al.* 1996; Fox 1998)—is to compute a probability distribution over all possible positions in the environment. Let $l = \langle x, y, \theta \rangle$ denote a position in the state space of the robot, where x and y are the robot’s coordinates in a world-centered Cartesian reference frame, and θ is the robot’s orientation. The distribution $Bel(l)$ expresses the robot’s belief for being at position l . Initially, $Bel(l)$ reflects the initial state of knowledge: if the robot knows its initial position, $Bel(l)$ is centered on the correct position; if the robot does not know its initial position, $Bel(l)$ is uniformly distributed to reflect the global uncertainty of the robot. As the robot operates, $Bel(l)$ is incrementally refined.

Markov localization applies two different probabilistic models to update $Bel(l)$, an action model to incorporate movements of the robot into $Bel(l)$ and a perception model to update the belief upon sensory input:

Robot motion is modeled by a conditional probability $P(l | l', a)$ (a kernel), specifying the probability that a measured movement action a , when executed at l' , carries the robot to l . $Bel(l)$ is then updated according to the following general formula, commonly used in Markov chains (Chung 1960):

$$Bel(l) \leftarrow \int P(l | l', a) Bel(l') dl' \quad (1)$$

The term $P(l | l', a)$ represents a model of the robot’s kinematics, whose probabilistic component accounts for errors

in odometry. Following (Burgard *et al.* 1996), we assume odometry errors to be distributed normally.

Sensor readings are integrated with Bayes rule. Let s denote a sensor reading and $P(s | l)$ the likelihood of perceiving s given that the robot is at position l , then $Bel(l)$ is updated according to the following rule:

$$Bel(l) \leftarrow \alpha P(s | l) Bel(l) \quad (2)$$

Here α is a normalizer, which ensures that $Bel(l)$ integrates to 1.

Strictly speaking, both update steps are only applicable if the environment is *Markovian*, that is, if past sensor readings are conditionally independent of future readings given the true position of the robot. Recent extensions to non-Markovian environments (Fox *et al.* 1998) can easily be stipulated to the MCL approach; hence, throughout this paper will assume that the environment is Markovian and will not pay further attention to this issue.

Prior Work

Existing approaches to mobile robot localization can be distinguished by the way they represent the state space of the robot.

Kalman filter-based techniques. Most of the earlier approaches to robot localization apply Kalman filters (Kalman 1960). The vast majority of these approaches is based on the assumption that the uncertainty in the robot’s position can be represented by a unimodal Gaussian distribution. Sensor readings, too, are assumed to map to Gaussian-shaped distributions over the robot’s position. For these assumptions, Kalman filters provide extremely efficient update rules that can be shown to be optimal (relative to the assumptions) (Maybeck 1979). Kalman filter-based techniques (Leonard & Durrant-Whyte 1992; Schiele & Crowley 1994; Gutmann & Schlegel 1996) have proven to be robust and accurate for keeping track of the robot’s position. However, since these techniques do not represent multi-modal probability distributions, which frequently occur during global localization. In practice, localization approaches using Kalman filters typically require that the starting position of the robot is known. In addition, Kalman filters rely on sensor models that generate estimates with Gaussian uncertainty—which is often unrealistic.

Topological Markov localization. To overcome these limitations, different approaches have used increasingly richer schemes to represent uncertainty, moving beyond the Gaussian density assumption inherent in the vanilla Kalman filter. These different methods can be roughly distinguished by the type of discretization used for the representation of the state space. In (Nourbakhsh, Powers, & Birchfield 1995; Simmons & Koenig 1995; Kaelbling, Cassandra, & Kurien 1996), Markov localization is used for landmark-based corridor navigation and the state space is organized according to the coarse, topological structure of the environment. The

coarse resolution of the state representation limits the accuracy of the position estimates. Topological approaches typically give only a rough sense as to where the robot is.

Grid-based Markov localization. To deal with multi-modal and non-Gaussian densities at a fine resolution (as opposed to the coarser discretization in the above methods), grid-based approaches perform numerical integration over an evenly spaced grid of points (Burgard *et al.* 1996; 1998b; Fox 1998). This involves discretizing the interesting part of the state space, and use it as the basis for an approximation of the state space density, e.g. by a piece-wise constant function. Grid-based methods are powerful, but suffer from excessive computational overhead and *a priori* commitment to the size and resolution of the state space. In addition, the resolution and thereby also the precision at which they can represent the state has to be fixed beforehand. The computational requirements have an effect on accuracy as well, as not all measurements can be processed in real-time, and valuable information about the state is thereby discarded. Recent work (Burgard *et al.* 1998b) has begun to address some of these problems, using *oct-trees* to obtain a variable resolution representation of the state space. This has the advantage of concentrating the computation and memory usage where needed, and addresses the limitations arising from fixed resolutions.

Monte Carlo Localization

Sample-Based Density Approximation

MCL is a version of sampling/importance re-sampling (SIR) (Rubin 1988). It is known alternatively as the bootstrap filter (Gordon, Salmond, & Smith 1993), the Monte-Carlo filter (Kitagawa 1996), the Condensation algorithm (Isard & Blake 1998), or the survival of the fittest algorithm (Kanazawa, Koller, & Russell 1995). All these methods are generically known as *particle filters*, and a discussion of their properties can be found in (Doucet 1998).

The key idea underlying all this work is to represent the posterior belief $Bel(l)$ by a set of N weighted, random samples or *particles* $S = \{s_i \mid i = 1..N\}$. A sample set constitutes a discrete approximation of a probability distribution. Samples in MCL are of the type

$$\langle\langle x, y, \theta \rangle\rangle, p \rangle \quad (3)$$

where $\langle x, y, \theta \rangle$ denote a robot position, and $p \geq 0$ is a numerical weighting factor, analogous to a discrete probability. For consistency, we assume $\sum_{n=1}^N p_n = 1$.

In analogy with the general Markov localization approach outlined in the previous section, MCL proceeds in two phases:

Robot motion. When the robot moves, MCL generates N new samples that approximate the robot's position after the motion command. Each sample is generated by *randomly* drawing a sample from the previously computed sample set, with likelihood determined by their p -values. Let l' denote

the position of this sample. The new sample's l is then generated by generating a single, random sample from $P(l \mid l', a)$, using the action a as observed. The p -value of the new sample is N^{-1} .

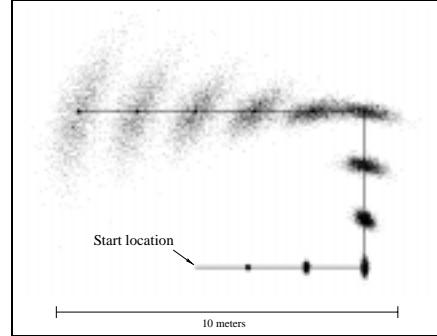


Fig. 1: Sampling-based approximation of the position belief for a non-sensing robot.

Figure 1 shows the effect of this sampling technique, starting at an initial known position (bottom center) and executing actions as indicated by the solid line. As can be seen there, the sample sets approximate distributions with increasing uncertainty, representing the gradual loss of position information due to slippage and drift.

Sensor readings are incorporated by re-weighting the sample set, in a way that implements Bayes rule in Markov localization. More specifically, let $\langle l, p \rangle$ be a sample. Then

$$p \leftarrow \alpha P(s \mid l) \quad (4)$$

where s is the sensor measurement, and α is a normalization constant that enforces $\sum_{n=1}^N p_n = 1$. The incorporation of sensor readings is typically performed in two phases, one in which p is multiplied by $P(s \mid l)$, and one in which the various p -values are normalized. An algorithm to perform this re-sampling process efficiently in $O(N)$ time is given in (Carpenter, Clifford, & Fernhead 1997).

In practice, we have found it useful to add a small number of uniformly distributed, random samples after each estimation step. Formally, this is legitimate because the SIR methodology (Rubin 1988) can accommodate arbitrary distributions for sampling as long as samples are weighted appropriately (using the factor p), and as long as the distribution from which samples are generated is non-zero at places where the distribution that is being approximated is non-zero—which is actually the case for MCL. The added samples are essential for relocalization in the rare event that the robot loses track of its position. Since MCL uses finite sample sets, it may happen that no sample is generated close to the correct robot position. In such cases, MCL would be unable to re-localize the robot. By adding a small number of random samples, however, MCL can effectively re-localize the robot, as documented in the experimental results section of this paper.

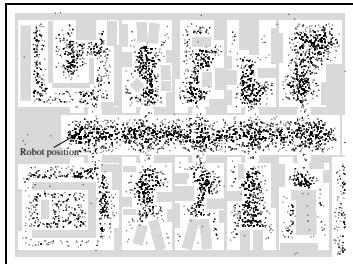


Fig. 2: Global localization: Initialization.



Fig. 3: Ambiguity due to symmetry.

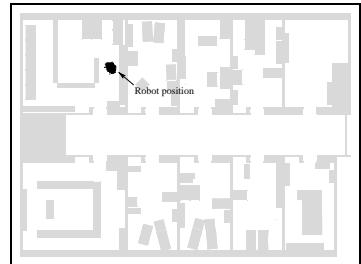


Fig. 4: Successful localization.

Properties of MCL

A nice property of the MCL algorithm is that it can universally approximate arbitrary probability distributions. As shown in (Tanner 1993), the variance of the importance sampler converges to zero at a rate of $1/\sqrt{N}$ (under conditions that are true for MCL). The sample set size naturally trades off accuracy and computational load. The true advantage, however, lies in the way MCL places computational resources. By sampling in proportion to likelihood, MCL focuses its computational resources on regions with high likelihood, where things really matter.

MCL is an online algorithm. It lends itself nicely to an any-time implementation (Dean & Boddy 1988; Zilberstein & Russell 1995). Any-time algorithms can generate an answer at *any* time; however, the quality of the solution increases over time. The sampling step in MCL can be terminated at any time. Thus, when a sensor reading arrives, or an action is executed, sampling is terminated and the resulting sample set is used for the next operation.

Adaptive Sample Set Sizes

In practice, the number of samples required to achieve a certain level of accuracy varies drastically. During global localization, the robot is completely ignorant as to where it is; hence, its belief uniformly covers its full three-dimensional state space. During position tracking, on the other hand, the uncertainty is typically small and often focused on lower-dimensional manifolds. Thus, many more samples are needed during global localization to approximate the true density with high accuracy, than are needed for position tracking.

MCL determines the sample set size on-the-fly. As in (Koller & Fratkina 1998), the idea is to use the divergence of $P(l)$ and $P(l | s)$, the belief *before* and *after* sensing, to determine the sample sets. More specifically, both motion data and sensor data is incorporated in a single step, and sampling is stopped whenever the sum of weights p (before normalization!) exceeds a threshold η . If the position predicted by odometry is well in tune with the sensor reading, each individual p is large and the sample set remains small. If, however, the sensor reading carries a lot of surprise, as is typically the case when the robot is globally uncertain or when it lost track of its position, the individual p -values are small and the sample set is large.

Our approach directly relates to the well-known property that the variance of the importance sampler is a function of the mismatch of the sampling distribution (in our case $P(l)$) and the distribution that is being approximated with the weighted sample (in our case $P(l | s)$) (Tanner 1993). The less these distributions agree, the larger the variance (approximation error). The idea is here to compensate such error by larger sample set sizes, to obtain approximately uniform error.

A Graphical Example

Figures 2 to 4 illustrate MCL in practice. Shown there is a series of sample sets (projected into 2D) generated during global localization of our robot RHINO (Figure 5), as it operates in an office building. In Figure 2, the robot is globally uncertain; hence the samples are spread uniformly through the free-space. Figure 3 shows the sample set after approximately 1 meter of robot motion, at which point MCL has disambiguated the robot's position up to a single symmetry. Finally, after another 2 meters of robot motion, the ambiguity is resolved, the robot knows where it is. The majority of samples is now centered tightly around the correct position, as shown in Figure 4.

Experimental Results

To evaluate the utility of sampling in localization, we thoroughly tested MCL in a range of real-world environments, applying it to three different types of sensors (cameras, sonar, and laser proximity data). The two primary results are:

1. MCL yields significantly more accurate localization results than the most accurate previous Markov localization algorithm, while consuming an order of magnitude less memory and computational resources. In some cases, MCL reliably localizes the robot whereas previous methods fail.
2. By and large, adaptive sampling performs equally well as MCL with fixed sample sets. In scenarios involving a large range of different uncertainties (global vs. local), however, adaptive sampling is superior to fixed sample sizes.

Our experiments have been carried out using several B21, B18, and Pioneer robots manufactured by ISR/RWI, shown in Figure 5. These robots are equipped with arrays of sonar



Fig. 5: Four of the robots used for testing: Rhino, Minerva, Robin, and Marian.

sensors (from 7 to 24), one or two laser range finders, and in the case of Minerva, shown in Figure 5, a B/W camera pointed at the ceiling. Even though all experimental results discussed here use pre-recorded data sets (to facilitate the analysis), all evaluations have been performed strictly under run-time conditions (unless explicitly noted). In fact, we have routinely ran cooperative teams of mobile robots using MCL for localization (Fox *et al.* 1999).

Comparison to Grid-Based Localization

The first series of experiments characterizes the different capabilities of MCL and compares it to grid-based Markov localization, which presumably is the most accurate Markov localization technique to date (Burgard *et al.* 1996; 1998b; Fox 1998).

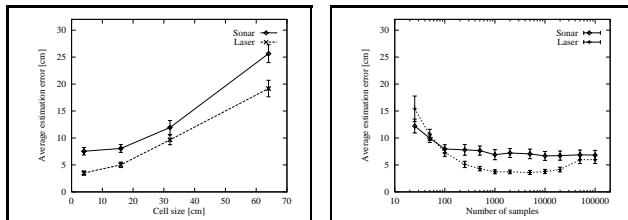


Fig. 6: Accuracy of (a) grid-based Markov localization using different spatial resolutions and (b) MCL for different numbers of samples (log scale).

Figure 6 (a) plots the localization accuracy for grid-based localization as a function of the grid resolution. These results were obtained using data recorded in the environment shown in Figure 2. They are nicely suited for our experiments because the exact same data has already been used to compare different localization approaches, including grid-based Markov localization (which was the only one that solved the global localization problem) (Gutmann *et al.* 1998). Notice that the results for grid-based localization shown in Figure 6 were not generated in real-time. As shown there, the accuracy increases with the resolution of the grid, both for sonar (solid line) and for laser data (dashed line). However, grid sizes below 8 cm do not permit updating in real-time, even when highly efficient, selective update schemes are used (Fox, Burgard, & Thrun 1999). Results for MCL with fixed sample set sizes are shown in Fig-

ure 6 (b). These results have been generated using real-time conditions. Here very small sample sets are disadvantageous, since they infer too large an error in the approximation. Large sample sets are also disadvantageous, since processing them requires too much time and fewer sensor items can be processed in real-time. The “optimal” sample set size, according to Figure 6 (b), is somewhere between 1,000 and 5,000 samples. Grid-based localization, to reach the same level of accuracy, has to use grids with 4cm resolution—which is infeasible given even our best computers.

In comparison, the grid-based approach, with a resolution of 20 cm, requires almost exactly ten times as much memory when compared to MCL with 5,000 samples. During global localization, integrating a single sensor scan requires up to 120 seconds using the grid-based approach, whereas MCL consumes consistently less than 3 seconds under otherwise equal conditions. Once the robot has been localized globally, however, grid-based localization updates grid-cells *selectively* as described in (Burgard *et al.* 1998b; Fox 1998), and both approaches are about equally fast.

Vision-based Localization

To test MCL in extreme situations, we evaluated it in a populated public place. During a two-week exhibition, our robot Minerva was employed as a tour-guide in the Smithsonian’s Museum of Natural History (Thrun *et al.* 1999). To aid localization, Minerva is equipped with a camera pointed towards the ceiling. Figure 7 shows a mosaic of the museum’s ceiling, constructed using a method described in (Thrun *et al.* 1999). The data used here is the most difficult data set in our possession, as the robot traveled with speeds of up to 163 cm/sec. Whenever it entered or left the carpeted area in the center of the museum, it crossed a 2cm bump which introduced significant errors in the robot’s odometry. Figure 8 shows the path measured by Minerva’s odometry.

When *only* using vision information, grid-based localization fails to track the robot accurately. This is because the computational overhead makes it impossible to incorporate sufficiently many images. MCL, however, succeeded in globally localizing the robot, and tracking the robot’s position (see also (Dellaert *et al.* 1999a)). Figure 9 shows the path estimated by our MCL technique. Although the localization error is sometimes above 1 meter, the system is able to keep track of multiple hypotheses and thus to recover from localization errors. The grid-based Markov localization system, however, was not able to track the whole 700m long path of the trajectory. In all our experiments, which were carried out under real-time conditions, the grid-based technique quickly lost track of the robot’s position (which, as was verified, would not be the case if the grid-based approach was given unlimited computational power). These results document that MCL is clearly superior to our previous grid-based approach.

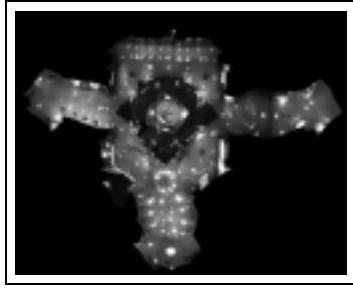


Fig. 7: Ceiling map of the NMAH

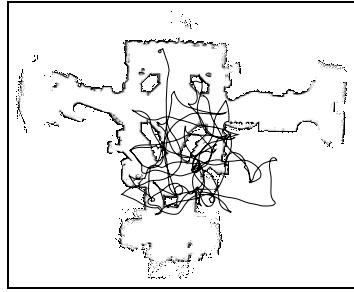


Fig. 8: Odometry information recorded by Minerva on a 700 m long trajectory

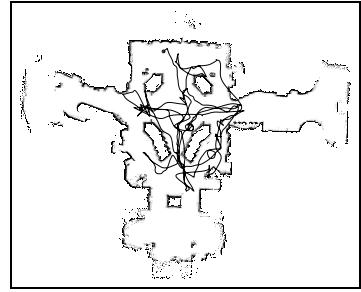


Fig. 9: Trajectory estimated given the ceiling map and the center pixels of on-line images.

Adaptive Sampling

Finally, we evaluated the utility of MCL's *adaptive* approach to sampling. In particular, were were interested in determining the relative merit of the adaptive sampling scheme, if any, over a fixed, static sample set (as used in some of the experiments above and in an earlier version of MCL (Dellaert *et al.* 1999b)). In a final series of experiments, we applied MCL with adaptive and fixed sample set sizes using data recorded with Minerva in the Smithsonian museum. Here we use the laser range data instead of the vision data, to illustrate that MCL also works well with laser range data in environments as challenging as the one studied here.

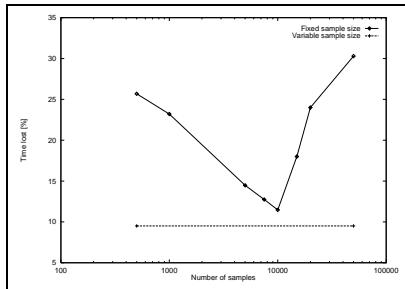


Fig. 10: Localization error for MCL with fixed sample set sizes (top figure) and adaptive sampling (bottom line)

In the first set of experiments we tested the ability of MCL to track the robot as it moved through the museum. In this case it turned out that adaptive sampling has no significant impact on the tracking ability of the Monte Carlo Localization. This result is not surprising since during tracking the position of the robot is concentrated on a small area.

We then evaluated the influence of adapting the sample size on the ability to *globally* localize the robot, and to recover from extreme localization failure. For the latter, we manually introduced severe errors into the data, to test the robustness of MCL in the extreme. In our experiments we “tele-ported” the robot at random points in time to other locations. Technically, this was done by changing the robot's orientation by 180 ± 90 degrees and shifting it by ± 200 cm, without letting the robot know. These perturbations were introduced randomly, with a probability of 0.01 per meter of robot motion. Obviously, such incidents make the robot lose its position, and therefore are well suited to test localization

under extreme situations.

Here we found adaptive sampling to be superior to MCL with fixed sample sets. Figure 10 shows the comparison. The top curve depicts the frequency with which the error was larger than 1 meter (our tolerance threshold), for different sample set sizes. The bottom line gives the same result for the adaptive sampling approach. As is easy to be seen, adaptive sampling yields smaller error than the best MCL with fixed sample set sizes. Our results have been obtained by averaging data collected along 700 meters of high-speed robot motion.

Conclusion and Future Work

This paper presented Monte Carlo Localization (MCL), a sample-based algorithm for mobile robot localization. MCL differs from previous approaches in that it uses randomized samples (particles) to represent the robot's belief. This leads to a variety of advantages over previous approaches: A significant reduction in computation and memory consumption, which leads to a higher frequency at which the robot can incorporate sensor data, which in turn implies much higher accuracy. MCL is also much easier to implement than previous Markov localization approaches. Instead of having to reason about entire probability distributions, MCL randomly *guesses* possible positions, in a way that favors likely positions over unlikely ones. An adaptive sampling scheme was proposed that enables MCL to adjust the number of samples in proportion to the amount of surprise in the sensor data. Consequently, MCL uses few samples when tracking the robot's position, but increases the sample set size when the robot loses track of its position, or otherwise is forced to globally localize the robot.

MCL has been tested thoroughly in practice. As our empirical results suggest, MCL beats previous Markov localization methods by an order of magnitude in memory and computation requirements, while yielding significantly more accurate results. In some cases, MCL succeeds where grid-based Markov localization fails.

In future work, the increased efficiency of our sample-based localization will be applied to multi robot scenarios, where the sample sets of the different robots can be synchronized whenever one robot detects another. First experiments conducted with two robots show that the robots are able to

localize themselves much faster when combining their sample sets (Fox *et al.* 1999). Here, the robots were equipped with laser range-finders and cameras to detect each other. We also plan to apply Monte Carlo methods to the problem of map acquisition, where recent work has led to new statistical frameworks that have been successfull applied to large, cyclic environments using grid representations (Thrun, Fox, & Burgard 1998).

Acknowledgment

This research is sponsored in part by NSF, DARPA via TACOM (contract number DAAE07-98-C-L032) and Rome Labs (contract number F30602-98-2-0137), and also by the EC (contract number ERBFMRX-CT96-0049) under the TMR programme.

References

- Borenstein, J.; Everett, B.; and Feng, L. 1996. *Navigating Mobile Robots: Systems and Techniques*. A. K. Peters, Ltd..
- Burgard, W.; Fox, D.; Hennig, D.; and Schmidt, T. 1996. Estimating the absolute position of a mobile robot using position probability grids. Proc. of AAAI-96.
- Burgard, W.; Cremers, A.; Fox, D.; Hähnel, D.; Lakemeyer, G.; Schulz, D.; Steiner, W.; and Thrun, S. 1998a. The Interactive Museum Tour-Guide Robot. Proc. of AAAI-98.
- Burgard, W.; Derr, A.; Fox, D.; and Cremers, A. 1998b. Integrating global position estimation and position tracking for mobile robots: the Dynamic Markov Localization approach. Proc. of IROS-98.
- Carpenter, J.; Clifford, P.; and Fernhead, P. 1997. An improved particle filter for non-linear problems. TR, Dept. of Statistics, Univ. of Oxford.
- Chung, K. 1960. *Markov chains with stationary transition probabilities*. Springer.
- Cox, I. 1991. Blanche—an experiment in guidance and navigation of an autonomous robot vehicle. *IEEE Transactions on Robotics and Automation* 7(2).
- Dean, T. L., and Boddy, M. 1988. An analysis of time-dependent planning. Proc. of AAAI-92.
- Dellaert, F.; Burgard, W.; Fox, D.; and Thrun, S. 1999a. Using the condensation algorithm for robust, vision-based mobile robot localization. Proc. of CVPR-99.
- Dellaert, F.; Fox, D.; Burgard, W.; and Thrun, S. 1999b. Monte Carlo localization for mobile robots. Proc. of ICRA-99.
- Doucet, A. 1998. On sequential simulation-based methods for Bayesian filtering. TR CUED/F-INFENG/TR.310, Dept. of Engineering, Univ. of Cambridge.
- Endres, H.; Feiten, W.; and Lawitzky, G. 1998. Field test of a navigation system: Autonomous cleaning in supermarkets. Proc. of ICRA-98.
- Engelson, S. 1994. *Passive Map Learning and Visual Place Recognition*. Ph.D. Diss., Dept. of Computer Science, Yale University.
- Fox, D.; Burgard, W.; Thrun, S.; and Cremers, A. 1998. Position estimation for mobile robots in dynamic environments. Proc. of AAAI-98.
- Fox, D.; Burgard, W.; Kruppa, H.; and Thrun, S. 1999. A monte carlo algorithm for multi-robot localization. TR CMU-CS-99-120, Carnegie Mellon University.
- Fox, D.; Burgard, W.; and Thrun, S. 1999. Active markov localization for mobile robots. *Robotics and Autonomous Systems* 25:3-4.
- Fox, D. 1998. *Markov Localization: A Probabilistic Framework for Mobile Robot Localization and Naviagation*. Ph.D. Diss, University of Bonn, Germany.
- Gelb, A. 1974. *Applied Optimal Estimation*. MIT Press.
- Gordon, N.; Salmond, D.; and Smith, A. 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F* 140(2).
- Gutmann, J.-S., and Schlegel, C. 1996. AMOS: Comparison of scan matching approaches for self-localizati on in indoor environments. Proc. of Euromicro. IEEE Computer Society Press.
- Gutmann, J.-S.; Burgard, W.; Fox, D.; and Konolige, K. 1998. An experimental comparison of localization methods. Proc. of IROS-98.
- Handschin, J. 1970. Monte Carlo techniques for prediction and filtering of non-linear stochastic processes. *Automatica* 6.
- Isard, M., and Blake, A. 1998. Condensation–conditional density propagation for visual tracking. *International Journal of Computer Vision* 29(1).
- Kaelbling, L.; Cassandra, A.; and Kurien, J. 1996. Acting under uncertainty: Discrete bayesian models for mobile-robot navigation. Proc. of IROS-96.
- Kalman, R. 1960. A new approach to linear filtering and prediction problems. *Tansaction of the ASME–Journal of basic engineering* 35–45.
- Kanazawa, K.; Koller, D.; and Russell, S. 1995. Stochastic simulation algorithms for dynamic probabilistic networks. Proc. of UAI-95.
- Kitagawa, G. 1996. Monte carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics* 5(1).
- Koller, D., and Fratkina, R. 1998. Using learning for approximation in stochastic processes. Proc. of ICML-98.
- Kortenkamp, D.; Bonasso, R.; and Murphy, R., eds. 1997. *AI-based Mobile Robots: Case studies of successful robot systems*. MIT Press.
- Leonard, J., and Durrant-Whyte, H. 1992. *Directed Sonar Sensing for Mobile Robot Navigation*. Kluwer Academic.
- Maybeck, P. 1979. *Stochastic Models, Estimation and Control*, Vol. 1. Academic Press.
- Nourbakhsh, I.; Powers, R.; and Birchfield, S. 1995. DERVISH an office-navigating robot. *AI Magazine* 16(2).
- Rubin, D. 1988. Using the SIR algorithm to simulate posterior distributions. *Bayesian Statistics 3*. Oxford University Press.
- Schiele, B., and Crowley, J. 1994. A comparison of position estimation techniques using occupancy grids. Proc. of ICRA-94.
- Simmons, R., and Koenig, S. 1995. Probabilistic robot navigation in partially observable environments. Proc. of ICML-95.
- Smith, R.; Self, M.; and Cheeseman, P. 1990. Estimating uncertain spatial relationships in robotics. Cox, I., and Wilfong, G., eds., *Autonomous Robot Vehicles*. Springer.
- Tanner, M. 1993. *Tools for Statistical Inference*. Springer.
- Thrun, S.; Bennewitz, M.; Burgard, W.; Cremers, A.; Dellaert, F.; Fox, D.; Hähnel, D.; Rosenberg, C.; Roy, N.; Schulte, J.; and Schulz, D. 1999. MINERVA: A second generation mobile tour-guide robot. Proc. of ICRA-99.
- Thrun, S.; Fox, D.; and Burgard, W. 1998. A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning* 31.
- Zilberstein, S., and Russell, S. 1995. Approximate reasoning using anytime algorithms. *Inprecise and Approximate Computation*. Kluwer.

An Efficient FastSLAM Algorithm for Generating Maps of Large-Scale Cyclic Environments from Raw Laser Range Measurements

Dirk Hähnel¹ and Wolfram Burgard¹ and Dieter Fox² and Sebastian Thrun³

¹University of Freiburg, Department of Computer Science, Freiburg, Germany

²University of Washington, Computer Science and Engineering, Seattle, WA, USA

³Stanford University, Computer Science Department, Stanford, CA, USA

Abstract

The ability to learn a consistent model of its environment is a prerequisite for autonomous mobile robots. A particularly challenging problem in acquiring environment maps is that of closing loops; loops in the environment create challenging data association problems [9]. This paper presents a novel algorithm that combines Rao-Blackwellized particle filtering and scan matching. In our approach scan matching is used for minimizing odometric errors during mapping. A probabilistic model of the residual errors of scan matching process is then used for the resampling steps. This way the number of samples required is seriously reduced. Simultaneously we reduce the particle depletion problem that typically prevents the robot from closing large loops. We present extensive experiments that illustrate the superior performance of our approach compared to previous approaches.

I. Introduction

Learning maps with mobile robots is one of the fundamental problems in mobile robotics. In the literature, the mobile robot mapping problem is often referred to as the *simultaneous localization and mapping problem (SLAM)* [5], [6], [9], [12], [13], [19]. This is because mapping includes both, estimating the position of the robot relative to the map and generating a map using the sensory input and the estimates about the robot's pose.

One of the hardest problems in robotic mapping is that of loop closure. As a robot traverses a large cycle in the environment, it faces the hard data association of correctly connecting to its own map under large position errors. This problem has long been acknowledged for its hardness, and a number of approaches have addressed it [4], [9], [20]. Recently, Murphy and colleagues have presented Rao-Blackwellized particle filters [8], [17] as an effective way of representing alternative hypotheses on robot paths and associated maps. Montemerlo *et al.* [14], [15] extended this idea to efficient landmark-based SLAM using Gaussian representations to and were the first to successfully realize it on real robots.

In this paper we present a highly efficient approach to simultaneous localization and mapping with laser scans.

As previously proposed by Murphpy [17], our approach applies a Rao-Blackwellized particle filter to estimate a posterior of the path of the robot, in which each particle has associated to it an entire map. This differs from work in [20], where only a single map is retained. To scale to large-scale environments, we transform sequences of laser range-scans into odometry measurements using range-scan registration techniques [10]. This way our system can deal with significantly larger environments than Murphpy's approach [17], since the scan matching yields odometry estimates that are an order of magnitude more accurate than the raw wheel encoder data. Simultaneously, the transformation of sequences of scans into odometry measurements reduces the well-known particle deprivation problem [21], since the number of resampling operations is significantly reduced. By using a learned model of the residual errors of the range registration our approach can correctly integrate the corrected odometry into the particle filtering process. As a result, we obtain a drastic reduction in the number of particles needed to build large-scale maps, or, put differently, an improved ability to map large environments. This is demonstrated in our experimental results section, in which we compare our approach to previous techniques.

This paper is organized as follows. In the following section, we will discuss techniques for incremental probabilistic mapping and localization. In Section III, we describe our approach to integrate scan matching with Rao-Blackwellized particle filters to achieve a robust approach for simultaneous mapping and localization. Section IV presents several experiments illustrating that our approach can successfully learn accurate maps with range scanners in large-scale environments. Additionally, we present experiments illustrating that our technique outperforms existing approaches.

II. Incremental Probabilistic Mapping and Localization

In probabilistic terms the goal of map learning is to find the map and the robot positions which yield the best interpretation of the data d_t gathered by the robot [19]. Here the data $d_t = \{u_{0:t-1}, z_{1:t}\}$ consists of a stream

of odometry measurements $u_{0:t-1}$ and perceptions of the environment $z_{1:t}$. The mapping problem can be phrased as recursive Bayesian filtering for estimating the robot positions along with a map of the environment:

$$p(x_{1:t}, m \mid z_{1:t}, u_{0:t-1}) = \alpha \cdot p(z_t \mid x_t, m) \cdot \int p(x_t \mid x_{t-1}, u_{t-1}) p(x_{1:t-1}, m \mid z_{1:t-1}, u_{0:t-2}) dx_{1:t-1} \quad (1)$$

In probabilistic mapping and localization it is typically assumed that the odometry measurements are governed by a so-called probabilistic motion model $p(x_t \mid x_{t-1}, u_{t-1})$ which specifies the likelihood that the robot is at x_t given that it previously was at x_{t-1} and the motion u_{t-1} was measured. On the other hand, the observations follow the so-called observation model $p(z \mid x)$, which defines for every possible location x in the environment the likelihood of the observation z .

Unfortunately, estimating the full posterior in Equation 1 is not tractable in general. One popular approach is to restrict observations to landmark detections, and represents robot positions by Gaussians [6]. In this context, the Bayes filter can be approximated efficiently by an EKF for which the state consists of the robot positions along with positions of the landmarks. Other researchers attempted to overcome the restrictions to landmark observations by using laser range-finders and incremental scan matching [2], [18], [22]. The general idea of these approaches can be summarized as follows (see also [19]). At any point $t-1$ in time, the robot is given an estimate of its pose \hat{x}_{t-1} and a map $\hat{m}(\hat{x}_{1:t-1}, z_{1:t-1})$. After the robot moved further on and after taking a new measurement z_t , the robot determines the most likely new pose \hat{x}_t such that

$$\begin{aligned} \hat{x}_t &= \underset{x_t}{\operatorname{argmax}} \{p(z_t \mid x_t, \hat{m}(\hat{x}_{1:t-1}, z_{1:t-1})) \\ &\quad \cdot p(x_t \mid u_{t-1}, \hat{x}_{t-1})\}. \end{aligned} \quad (2)$$

It does this by trading off the consistency of the measurement with the map (first term on the right-hand side in (2)) and the consistency of the new pose with the control action and the previous pose (second term on the right-hand side in (2)). The map is then extended by the new measurement z_t , using the pose \hat{x}_t as the pose at which this measurement was taken.

Whereas this approach has the advantage that it yields accurate results and can be implemented efficiently if the registration is performed with respect to a global map or with respect to only a fixed number of scans. Its major disadvantage lies in the greedy maximization step. When the robot has to close larger loops, this approach suffers from registration errors during loop closures and therefore tends to fail in large environments. To overcome this problem, extensions of this approach have been developed which maintain a posterior about the position of the vehicle [9], [19]. The key idea of these techniques is to delay the maximization until the robot detects that a loop has been closed. This is usually done by identifying that the robot enters an already known area from an

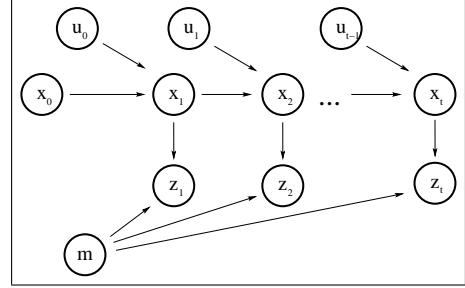


Fig. 1. Graphical model of concurrent mapping and localization as filtering process.

unknown area and simultaneously observes a high likelihood of its observations for potential positions that are under consideration. If the registration of the vehicle in its map can be done with high likelihood, the previous poses are corrected backwards in time according to the pose correction that is necessary to properly close the loop. The position posterior is then replaced by a Dirac distribution which has its mode at the most likely position when the robot closes the loop. Furthermore, subsequent backwards corrections are stopped when the robot reaches this node. Whereas this approach can reliably close even large loops, it has the disadvantage that it under-estimates the uncertainty in the robot pose when closing loops.

More recently, Murphy and colleagues [17], [8] have presented Rao-Blackwellized particle filters as an efficient way to represent the full posterior of the robot pose. Figure 1 depicts a graphical model of Rao-Blackwellized simultaneous mapping and localization. The key idea of this approach is to solve the recursive Bayes filter update by the following equation:

$$\begin{aligned} p(x_{1:t}, m \mid z_{1:t}, u_{0:t-1}) &= \\ p(m \mid x_{1:t}, z_{1:t}, u_{0:t-1}) p(x_{1:t} \mid z_{1:t}, u_{0:t-1}) \end{aligned} \quad (3)$$

Here, a particle filter is used to represent robot trajectories $x_{1:t}$ and a different map is conditioned on each sample of the particle filter. The importance weights of the samples are computed according to the likelihoods of the observations in the maximum likelihood map constructed using exactly the positions this particular particle has taken. The key advantage of this approach is that the samples approximate at every point in time the full posterior over robot poses and maps. The first successful realization on real robots of an extended version of this technique has been presented by Montemerlo *et al.* [15].

However, particle filters are known to be subject to major approximation errors. One of these errors is known as the particle depletion problem [21]. This problem can lead to a divergence of the filter and can result in the lack of particles in the vicinity of the correct state. In the SLAM context this can prevent the robot from closing a given loop. There are two parameters that have a major influence on the approximation error. First, the number of particles needs to be high enough to represent the posterior. However, too many samples can prevent

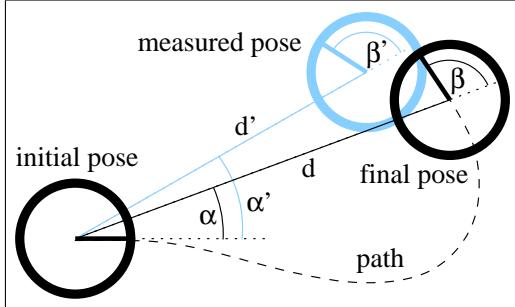


Fig. 2. Parameters of the probabilistic motion model.

the filtering process from being fast enough for online-processing. Furthermore, the number of resampling steps needs to be limited in order to avoid that the samples converge too quickly to the maximum likelihood state which is undesirable especially in ambiguous situations. On the other side, too few resampling steps could result in a divergence of the filter since many samples are wasted on unlikely states and the uncertainty typically introduced by robot motions would exceed the certainty gained by incorporating observations of the environment.

In the following section we will describe our solution to this problem. This approach transforms sequences of laser measurements into odometry measurements using a scan matching procedure and utilizes the remaining laser scans for map estimation.

III. Combining Laser-based FastSLAM with Scan Matching

Our approach to reduce the problems described above is to use a scan matching routine to correct the odometry and to use this corrected path information as input for the sampling step in the Rao-Blackwellized particle filter.

The 2d scan matching we apply, which is described in detail in [10], aligns a scan relative to the previous scans by computing an occupancy grid map [16] from the previous measurements. To avoid the time consuming ray-tracing operation required to compute the likelihood of a measurement $p(z | x)$ we apply an approximation which considers only the endpoint of a beam [11], [19]. This way, $p(z | x)$ can be computed efficiently using fast look-up operations. To also be able to incorporate maximum range measurements, our system assumes that the cell 20cm in front of that in which the maximum range measurement ends must be unoccupied.

A key question when combining a scan matching routine with a probabilistic technique is how to estimate the uncertainty of the scan matching process so as to correctly incorporate the corresponding uncertainty during the prediction step of the sampling procedure. In our current system we use a parametric model of the odometry error and learn the parameters of this model using data acquired during experiments. To learn the parameters of the model used for the experiments described here we performed an experiment in which we generated a statistics of alignment

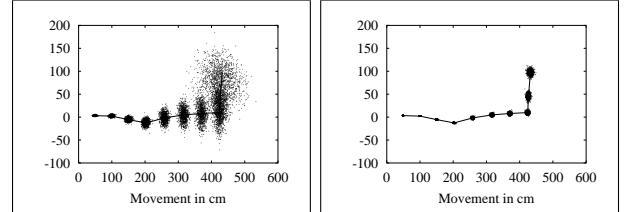


Fig. 3. Sample densities obtained with the models for the raw odometry (left image) and for scan matching (right image) for ten incremental movements of a real robot.

errors after convergence of scan matching. Using a data set recorded in the Intel Research Lab Seattle, we applied our system equipped with a manually designed motion model. We then took the resulting map (see Figure 9) as ground truth and compared the raw odometry and the results of the scan matching with the positions corrected by the routine. The error model we use has three parameters as it assumes that in every single movement there are three errors involved (see also Figure 2). First, whenever the robot starts to move, it makes a small rotational error $\alpha' - \alpha$. Second, the robot introduces a certain error $d' - d$ to the distance between the final location and the starting position. Finally, the true final orientation differs by a certain amount from the measured orientation which is expressed by a non-zero difference $\beta - \beta'$. The means and the variances of the relative errors in these three parameters were learned by comparing the approximated displacement after convergence of the scan matching routine with the (estimated) ground truth information. Alternative models of odometry errors and corresponding techniques for parameter estimation have been proposed by Borenstein and Feng [3], Doh *et al.* [7], as well as Bengtsson and Baerveldt [1].

Figure 3 plots the resulting sample densities obtained when relying on pure odometry (left image) and the densities obtain with the error model for the scan matching process (right image). As the figure shows, the samples are much more focused if the scan matching routine is used. This leads to the desired effect that the variance of the posterior is reduced, that fewer samples can be used, and that larger loops can be closed.

A graphical model of our approach to integrate results from the scan matching process into the Rao-Blackwellized sampling routine is depicted in Figure 4 (c.f. 1). The key idea is to compute every k steps a new odometry measurement u'_j out of the $k - 1$ previous observations z and the k most recent odometry readings. The k -th observation is then used to compute the weights of the samples in the particle filter. Note that this clear separation between laser scans used for odometry and laser scans used for map estimation ensures that all information is used only once.

One important aspect when using Rao-Blackwellized particle filters for mapping is the efficient update of the maps of the individual particles. Montemerlo *et al.* [15]

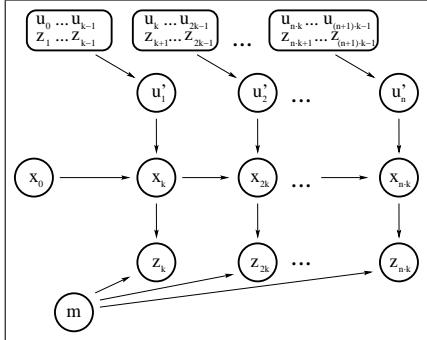


Fig. 4. Graphical model of the integration of scan matching and probabilistic mapping.

proposed a tree-structure to efficiently update the map. In the system described here, we only use a limited number of scans gathered by the robot to update the map of a particle. The scans chosen need to intersect with the area visible according to the pose of the corresponding particle. This way, the update of the map of every particle can be achieved in constant time. Although this is an approximation only, we never found any evidence that the quality of the resulting maps was decreased significantly.

IV. Experimental Results

The approach described above has been implemented and tested using different robotic platforms and in different environments as well as in extensive simulation runs. In all experiments, we found out that the system can operate online and can also robustly close large and nested loops.

A. Mapping Large-Scale Environments with Multiple Cycles

The first experiment was carried out using a Pioneer 2 robot equipped with a SICK LMS laser range-finder in the Intel Research Lab, Seattle, WA. The size of this environment is $28m \times 28m$. The robot traveled 491m with an average speed of 0.19m/s. Figure 5 shows the map generated based on the raw odometry data provided by the robot. As can be seen from the figure, the robot suffers from serious errors in odometry so that the resulting map is useless without any correction. Figure 6 (left) shows the map created with our scan matching technique. Although local structures of the map appear to be very accurate, the map is globally inconsistent. For example many structures like walls, doors etc. can be found twice and with a small offset between them. Finally, the right image of Figure 6 shows the resulting map obtained with our system. Although the sharpness of this map is not as high as that of the map created only with scan matching, they are globally consistent. The map was created in real-time, i.e. the computation time needed to process the data did not exceed the time to record them. We used 100 samples, a number we found to yield satisfactory results on all data sets. Figure 9 shows a map created using



Fig. 5. Mapping of the Intel Research Lab with the raw odometry data.



Fig. 6. Map of the Intel Research Lab after scan matching (left) and obtained in real-time with 100 samples (right).

500 particles. Whereas this map is more accurate and has a similar crispness as the scan matching map, the time to compute this map was several hours. Figure 7 visualizes the trajectories of all samples shortly before and after closing the major loop in this data set. As the left image illustrates, the robot is quite uncertain about its position relative to the starting position upon its return. However, after a few resampling steps the uncertainty has been reduced drastically (right image).

A second example map obtained with our approach is depicted in Figure 8. The map shows the fourth floor of the $50m \times 12m$ large Sieg Hall of the University of Washington. As can be seen from the figure, the robot went several times around the circle and still successfully learned a consistent map. This map was generated in real-time using 100 samples. The grid resolution was 10cm.

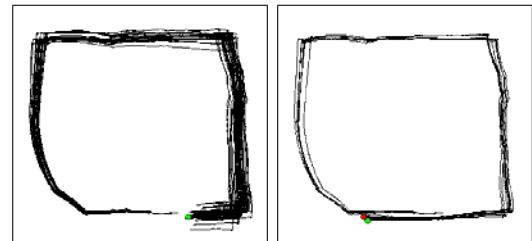


Fig. 7. Trajectories of all 100 samples shortly before (left) and after (right) closing the loop.



Fig. 8. Map of the Sieg Hall at the University of Washington created in real-time.

B. Comparison to Previous Online Techniques

The second experiment is designed to show the advantage of our integrated technique over previous approaches that represent a posterior over poses in a single map only [19], [9]. For this experiment we used a data set generated for the Wean Hall of the Carnegie Mellon University using our B21r simulator. The size of this environment is $32m \times 10m$. In the simulation the robot moved 251m with an average speed of 0.78m/s. To obtain realistic data, we added a serious amount of noise to the ground truth data provided by the simulation system. The resulting input trajectory is depicted in Figure 10 (left). Please note, that pure scan matching again failed to correctly close the loop using this data set. We implemented the particle filter strategy presented by Thrun *et al.* [20], [19]. In our system we achieve this by using only that sample with the highest likelihood during the resampling process whenever the robot closes a loop. After this, we continue with the normal resampling procedure described above. The middle image in Figure 10 shows the result of this procedure. The point in time when the system discovered that it closed the loop and the resulting inconsistencies are also labeled. The inconsistencies are a consequence of the fact that only the particle with the highest importance factor survives at the time the robot closes the loop. Since this particle does not always correspond to the correct position of the robot (as it is the case in this example) and since the motion model cannot compensate for this error, the resulting map contains errors. In contrast to that, our approach, that integrates scan matching with a Rao-Blackwellized particle filter, yields a consistent map of the environment (see right image of Figure 10) since it provides accurate predictions and simultaneously maintains the robot pose uncertainty in the posterior.

Finally, we analyzed whether the standard Rao-Blackwellized particle filter without odometry correction by scan matching provides the same performance as our approach. For this purpose we run the standard procedure using the input data for the Intel Research Lab. We used shorter resampling steps (three times more often than in the other runs to avoid a fast divergence) and 200 samples which was the maximum number of samples that allowed updates in real-time on our 1.8GHz Pentium IV PC. Since the standard procedure was not able to learn a consistent map, we repeated this experiment with increasing numbers of samples. It turned out that under 1000 samples, which was the maximum number our PC equipped with 768MB of main memory could handle, we



Fig. 9. Map of the Intel Research Lab obtained offline with 500 samples.



Fig. 11. Map created with the standard Rao-Blackwellized particle filtering technique in real-time using 100 samples and based on the raw odometry data.

could not observe a case in which the standard algorithm converged. An example map typically obtained using the standard algorithm is depicted in Figure 11. The fact that our algorithm reliably converges with 100 samples indicates that the integration of the scan matching routine yields an enormous improvement.

V. Conclusions

In this paper we presented a highly efficient algorithm for simultaneous mapping and localization using laser scans that combines a scan matching procedure with Rao-Blackwellized particle filtering. The scan matching routine is used to transform sequences of laser measurements into odometry measurements. The corrected odometry and the remaining laser scans are then used for map estimation in the particle filter. The lower variance in the corrected odometry reduces the number of necessary resampling steps and this way decreases the particle depletion problem. In practical experiments we demonstrated that our approach allows to learn maps of large-scale environments in real-time with as few as 100 samples. Simultaneously, it outperforms previous approaches with

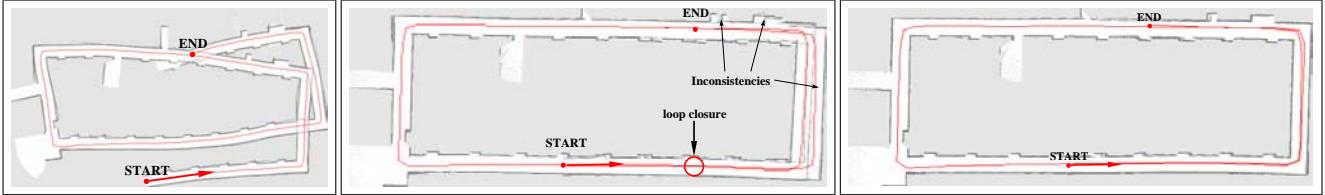


Fig. 10. (left) Map obtained from the raw input data in the simulation experiment. In the middle, the resulting map is shown if the mapping process is continued with the maximum likelihood sample after closing the loop at the marked place. The inconsistencies in the right part of the map show that the loop is not correctly closed. The map on the right was built using our approach.

respect to robustness and efficiency.

Acknowledgments

This work has partly been supported by the EC under contract number IST-2000-29456 and by the German Science Foundation (DFG) under contract number SFB/TR8-03. It has also been sponsored by DARPA's MARS, CoABS, MICA, and SDR Programme (contract numbers N66001-01-C-6018, NBCH1020014, F30602-98-2-0137, F30602-01-C-0219, and NBCHC020073) and by the NSF under grant numbers IIS-9876136, IIS-9877033, and IIS-0093406.

VI. REFERENCES

- [1] O. Bengtsson and A. Baerveldt. Localization in changing environment - estimation of a covariance matrix for the idc algorithm. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1931–1937, 2001.
- [2] P. Besl and N. McKay. A method for registration of 3d shapes. *Trans. Patt. Anal. Mach. Intell.* 14(2), pages 239–256, 1992.
- [3] J. Borenstein and Feng. L. Measurement and correction of systematic odometry errors in mobile robots. *IEEE Journal of Robotics and Automation*, 12(6):869–880, 1996.
- [4] M. Bosse, P. Newman, M. Soika, W. Feiten, J. Leonard, and S. Teller. An atlas framework for scalable mapping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2003.
- [5] J.A. Castellanos, J.M.M. Montiel, J. Neira, and J.D. Tardós. The SPmap: A probabilistic framework for simultaneous localization and map building. *IEEE Transactions on Robotics and Automation*, 15(5):948–953, 1999.
- [6] G. Dissanayake, H. Durrant-Whyte, and T. Bailey. A computationally efficient solution to the simultaneous localisation and map building (SLAM) problem. In *ICRA'2000 Workshop on Mobile Robot Navigation and Mapping*, 2000.
- [7] N. Doh, H. Choset, and W. K. Chung. Accurate relative localization using odometry. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 2003.
- [8] A. Doucet, J.F.G. de Freitas, K. Murphy, and S. Russell. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *Proc. of the Conference on Uncertainty in Artificial Intelligence (UAI)*, 2000.
- [9] J.-S. Gutmann and K. Konolige. Incremental mapping of large cyclic environments. In *Proc. of the IEEE Int. Symp. on Computational Intelligence in Robotics and Automation (CIRA)*, 1999.
- [10] D. Hähnel, D. Schulz, and W. Burgard. Mapping with mobile robots in populated environments. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.
- [11] K. Konolige. Markov localization using correlation. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.
- [12] J.J. Leonard and H.J.S. Feder. A computationally efficient method for large-scale concurrent mapping and localization. In *Proc. of the Ninth Int. Symp. on Robotics Research (ISRR)*, 1999.
- [13] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4:333–349, 1997.
- [14] M. Montemerlo and S. Thrun. Simultaneous localization and mapping with unknown data association using Fast-SLAM. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 2003.
- [15] M. Montemerlo, S. Thun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to simultaneous mapping and localization. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2002.
- [16] H.P. Moravec. Sensor fusion in certainty grids for mobile robots. *AI Magazine*, pages 61–74, Summer 1988.
- [17] K. Murphy. Bayesian map learning in dynamic environments. In *Neural Info. Proc. Systems (NIPS)*, 1999.
- [18] T. Röfer. Using histogram correlation to create consistent laser scan maps. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.
- [19] S. Thrun. A probabilistic online mapping algorithm for teams of mobile robots. *International Journal of Robotics Research*, 20(5):335–363, 2001.
- [20] S. Thrun, W. Burgard, and D. Fox. A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 2000.
- [21] R. van der Merwe, A. Doucet, N. de Freitas, and E. Wan. The unscented particle filter. Technical Report CUED/F-INFENG/TR 380, Cambridge University, Department of Engineering, 2000.
- [22] G. Weiß, C. Wetzel, and E. von Puttkamer. Keeping track of position and orientation of moving indoor systems by correlation of range-finder scans. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 595–601, 1994.

Carnegie Mellon University
Research Showcase

Robotics Institute

School of Computer Science

12-1-2005

Market-Based Multirobot Coordination: A Comprehensive Survey and Analysis

Nidhi Kalra

Carnegie Mellon University

Robert Zlot

Carnegie Mellon University

M. Bernardine Dias

Carnegie Mellon University

Anthony Stentz

Carnegie Mellon University

Recommended Citation

Kalra, Nidhi; Zlot, Robert; Dias, M. Bernardine; and Stentz, Anthony, "Market-Based Multirobot Coordination: A Comprehensive Survey and Analysis" (2005). *Robotics Institute*. Paper 151.
<http://repository.cmu.edu/robotics/151>

This Technical Report is brought to you for free and open access by the School of Computer Science at Research Showcase. It has been accepted for inclusion in Robotics Institute by an authorized administrator of Research Showcase. For more information, please contact kbehrman@andrew.cmu.edu.

Market-Based Multirobot Coordination: A Comprehensive Survey and Analysis

Nidhi Kalra Robert Zlot M. Bernardine Dias
 Anthony Stentz

CMU-RI-TR-05-16

December 2005*

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

© Carnegie Mellon University

*A version of this technical report was published in July 2006 as an article in the IEEE Special Issue on Multirobot Coordination. The original technical report has been updated with select publications that appeared between its original publication date in December 2005 and the article's publication date in July 2006.

Abstract

As robotic technology improves, we charge robots with increasingly varied and difficult tasks. Many of these tasks can potentially be completed better by a team of robots working together than by individual robots working alone. Coordination can lead to faster task completion, increased robustness, higher-quality solutions, and the completion of tasks impossible for single robots. Nevertheless, effective coordination can be difficult to achieve because of a range of adverse real-world conditions including dynamic events, changing task demands, resource failures, and limited deliberation time. The desire to overcome these challenges and harness the benefits of robot teams has made multirobot coordination a vital field in robotics research.

Of the resulting wealth of research, market-based multirobot coordination approaches in particular have received significant attention and are growing in popularity within the community. These approaches harness the principles of market economies—which have successfully governed human coordination for thousands of years—and use them to enable robot coordination. In market-based approaches, robots on the team act as self-interested agents operating in a virtual economy in which tasks and team resources are exchanged over the market in pursuit of individual profit. The essence of market-based approaches is that the process of robots trading tasks and resources with one another to maximize their wealth simultaneously improves the efficiency of the team.

Market-based approaches to multirobot coordination inherit many of the benefits associated with market economies, including flexibility, efficiency, responsiveness, robustness, scalability, and generality. In practice, they have been successfully implemented in a variety of domains ranging from mapping and exploration to robot soccer. The research literature on market-based approaches to coordination has now reached a critical mass that warrants a survey and analysis. This paper meets this need in three ways. First, it provides a tutorial on market-based approaches by discussing the motivating philosophy, defining the requirements and tradeoffs inherent in such approaches, analyzing their strengths and weaknesses, and placing them appropriately in the context of the larger set of approaches to multirobot coordination. Second, this paper surveys and analyzes the relevant literature. Third, it inspires and directs future research on this topic through a discussion of remaining challenges.

Contents

1	Introduction	1
2	Overview	3
2.1	Definition of a Market-based Approach	3
2.2	Auctions	4
2.3	Costs, Utilities, and Valuation	4
2.4	The Range of Coordination Approaches	5
3	Planning	6
3.1	Related Work	6
3.1.1	Planning and Task Allocation	6
3.1.2	Planning and Task Decomposition	10
3.1.3	Planning and Task Execution	11
3.2	Future Challenges	12
4	Quality of Solution	12
4.1	Related Work	13
4.1.1	Relationship to Operations Research	15
4.2	Future Challenges	17
5	Scalability	17
5.1	Related Work	17
5.1.1	Computation and Communication Considerations	18
5.1.2	Selective or Opportunistic Centralization	21
5.2	Future Challenges	21
6	Dynamic Events and Environments	21
6.1	Related Work	22
6.1.1	Robustness and Fluidity	22
6.1.2	Online Tasks	23
6.1.3	Uncertainty	24
6.2	Future Challenges	24
7	Heterogeneous Teams	25
7.1	Related Work	25
7.2	Future Challenges	26
8	Learning and Adaptation	26
8.1	Related Work	27
8.2	Future Challenges	27

9 Practical Considerations	27
9.1 Related Work	27
9.1.1 Flexibility	27
9.1.2 Extensibility	28
9.1.3 Implementation	28
9.1.4 Comparisons	28
9.2 Future Challenges	29
10 Conclusions and Future Directions	29
A Example Problems and Case Studies	37
A.1 Basic Aggregation	37
A.1.1 Aggregation of Multiple-Robot Tasks	38
A.2 Market-based Exploration	39

1 Introduction

As robots become an integral part of human life, we charge them with increasingly varied and difficult tasks including planetary exploration, manufacturing and construction, medical assistance, search and rescue, and port and warehouse automation. Like humans, robots working in challenging domains can potentially perform better by working together in teams than by working alone. Ideally, robots will coordinate to redistribute resources amongst themselves in a way that enables them to accomplish their mission efficiently and reliably. Coordination can lead to faster task completion, increased robustness, higher-quality solutions, and the completion of tasks impossible for single robots. However, these domains simultaneously present many obstacles to effective coordination, such as dynamic events, changing task demands, resource failures, the presence of adversaries, and limited time, energy, computation, communication, sensing, and mobility. Therefore, coordinating a multirobot team requires overcoming many formidable research challenges.

Humans have met these coordination challenges for thousands of years with increasingly sophisticated market economies. In these economies, self-interested individuals and groups trade goods and services to maximize their own profit; simultaneously, this redistribution results in an efficient production of output for the system as a whole. Researchers have recently applied the principles of market economies to multirobot coordination. In market-based multirobot systems, robots are designed as self-interested agents that operate in a virtual economy. Both the tasks that must be completed and the available resources are commodities of measurable worth that can be traded. For example, tasks can be assigned to robots via market mechanisms such as auctions. When a robot completes a task, it receives some payment in the form of virtual money for providing a service to the team. However, the robot must also pay for the resources it consumed to complete the task. The essence of market-based approaches is that, in a well-designed system, the process of robots trading tasks and resources with one another to maximize individual profit simultaneously improves the efficiency of the team.

To illustrate this more concretely, consider a team of robots performing a distributed sensing mission on Mars. As illustrated in Figure 1, the robots must gather data from specific sites of interest to scientists while consuming the least amount of energy. One important aspect of completing the mission is to determine which robot should visit each site. We can solve this problem using a market-based approach in which robots compete in auctions for each task of visiting a site. After estimating their resource usage for an offered task and submitting bids based on those expected costs, the robot with the best bid is awarded a contract for that site.

Specifically, suppose that we offer a maximum reward of \$50 for each task and that robots incur a cost of \$2 for each meter of travel (since the resource of concern is energy consumed). This \$50 is a *reserve price* that essentially says that the task should only be attempted if the site can be reached by increasing one's path length by less than 25 meters. Further suppose that a robot A is only 5 meters from a site S . Since A would have to spend \$10 to complete the task, it bids \$10. Meanwhile, a robot B that is 10 meters from the site bids \$20. A is awarded the contract because it can perform the task more efficiently and for less than the reserve price.

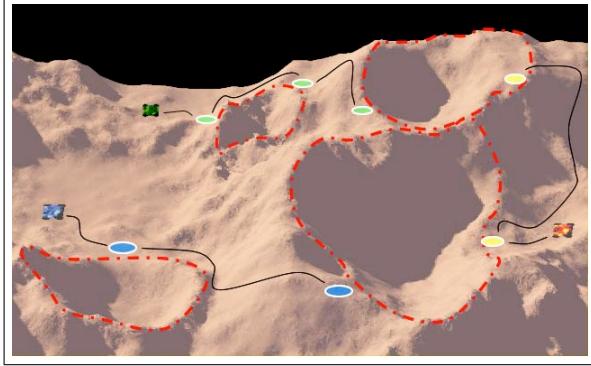


Figure 1: An illustration of three robots exploring Mars. The robots’ task is to gather data around the four craters, which can be achieved by visiting the highlighted target sites.

This simple example illustrates the basic mechanism of a market-based approach to coordination. As the problem increases in complexity with the addition of more robots, more resources (*e.g.* time, network bandwidth, computing power, sensors, etc.), added constraints between the tasks, dynamically changing tasks, and so forth, the coordination approach requires added functionality to produce efficient solutions. We use this distributed sensing scenario throughout the remainder of this paper to illustrate the complexities of coordination and the diversity of market-based approaches.

The earliest examples of market-based multiagent coordination appeared in the literature over thirty years ago [20, 62] and have been modified and adopted for multi-robot coordination in more recent years. This work is motivated by the growing popularity of market-based approaches and the lack of a comprehensive review of these approaches. This paper makes three contributions to the robotics literature. First, it provides a tutorial on market-based approaches by discussing the motivating philosophy, defining the requirements and tradeoffs inherent in such approaches, analyzing their strengths and weaknesses, and placing them appropriately in the context of the larger set of approaches to multirobot coordination. Second, this paper surveys and analyzes the relevant literature. Third, it inspires and directs future research on this topic through a discussion of remaining challenges. This paper in particular extends our shorter technical report [19]: it presents more concrete results, more thoroughly explores the topics, and includes new topics of discussion. Additionally, the appendix includes detailed example problems that demonstrate the step-by-step application of market-based approaches and a case-study of market-based multirobot exploration that highlights the variety of ways markets can be applied to one domain.

The scope of this paper is limited to market-based approaches for coordinating teams that include robots. Moreover, this review principally considers approaches that actively reason about the existence of other agents when coordinating the team, in contrast to approaches in which agents coexist. Nevertheless, related publications outside the stated scope of this paper are included as necessary to augment the discussion.

The following section provides an introduction to market-based mechanisms for

readers less familiar with the field. This overview is followed by an extensive review of market-based multirobot coordination approaches to date, categorized and analyzed across several relevant dimensions: planning, solution quality, scalability, dynamic events and environments, and heterogeneity. The paper concludes with a summary of the survey and future challenges in this research area. Further material on example problems and case studies is included in Appendix A.

2 Overview

In this section, we discuss key concepts that will provide a foundation for the remainder of the article, including a definition of market-based approaches and an introduction to auctions. We then place market-based approaches in the larger spectrum of coordination approaches.

2.1 Definition of a Market-based Approach

Most market-based multirobot and multiagent coordination approaches share a set of underlying elements. Market theory provides precise definitions for several of these elements. Borrowing from both bodies of literature, we define a market-based multirobot coordination approach based on the following requirements:

- The team is given an objective that can be decomposed into subcomponents achievable by individuals or subteams. The team has access to a limited set of resources with which to meet this objective.
- A global objective function quantifies the system designer’s preferences over all possible solutions.
- An individual utility function (or cost function) specified for each robot quantifies that robot’s preferences for its individual resource usage and contributions towards the team objective given its current state. Evaluating this function cannot require global or perfect information about the state of the team or team objective. Subteam preferences can also be quantified through a combination of individual utilities (or costs).
- A mapping is defined between the team objective function and individual and subteam utilities (or costs). This mapping addresses how the individual production and consumption of resources and individuals’ advancement of the team objective affect the overall solution.
- Resources and individual or subteam objectives can be redistributed using a mechanism such as an auction. This mechanism accepts as input teammates’ bids, which are computed as a function of their utilities (or costs), and determines an outcome that maximizes the mechanism-controlling agent’s utility (or minimizes the cost). In a well-designed mechanism, maximizing the mechanism-controlling agent’s utility (or minimizing cost) results in improving the team objective function value.

2.2 Auctions

Auctions are the most common mechanisms used in market-based approaches. In an auction, a set of items is put on the market by an auctioneer in an *announcement* phase, and the participants can make an offer for these items by submitting *bids* to the auctioneer. Once all bids are received or a pre-specified deadline has passed, the auction is then *cleared* in the *winner determination* phase by the auctioneer who decides which items to award and to whom. In robotic applications, the items for sale are typically tasks, roles, or resources. The bid prices reflect the robots' costs or utilities associated with completing a task, satisfying a role, or utilizing a resource.

The simplest kind of auction is a *single-item* auction in which only one item is offered. In such auctions, each participant submits a bid, and the auctioneer awards the item to the highest bidder¹. Alternatively, the auctioneer retains the item if no bid beats the auctioneer's price (called a *reserve price*). Bids are usually submitted only to the auctioneer; such *sealed-bid* auctions are in contrast to *open-cry* auctions where bidders have the benefit of overhearing the other bids as they are made. There are two common approaches to determining the sale price of the auctioned item. In a *first-price auction*, the sale price is the same as the winning bid. In a *Vickrey auction*, the sale price is the value of the second-highest bid and is intended to motivate truthful bids from the participants. Some multirobot systems have used Vickrey auctions (*e.g.* [49]), though the resulting allocations are equivalent to first-price auctions if the robots are designed to behave truthfully. Wolfstetter provides an excellent introductory survey into single-item auction theory [71].

Combinatorial auctions are more complex: multiple items are offered and each participant can bid on any combination of *bundles* (*i.e.* subsets) of these items. This allows the bidder to explicitly express the synergies between items. In the context of the Mars distributed sensing scenario, a bidder can express the positive synergy between two sites that are close together by bidding only slightly higher for the bundle containing these tasks than for either task individually. To express the negative synergy between two tasks located far from one another, the bid for the bundle would be much higher than the sum of the individual costs of the tasks. In general, there are an exponential number of bundles to consider in a combinatorial auction, which makes bid valuation, communication, and auction clearing intractable if all bundles are considered [55].

In between these two extremes are multi-item auctions in which multiple items are offered but the participants can win at most one item apiece. The maximum number of awards per auction may also be limited. Multi-item auctions can be thought of as a special case of combinatorial auctions where only bundles of cardinality one are considered; bidding and clearing become tractable, but the resulting solutions are generally much less efficient.

2.3 Costs, Utilities, and Valuation

The example scenario in Section 1 compares robots' suitability for tasks in terms of cost. That is, the auction allocates tasks to the robots with the lowest costs for perform-

¹We will assume utility maximization here; the case of cost minimization is analogous, with awards going to the lowest-cost bidders.

ing them and the overall goal is to minimize some global cost function. As suggested in Section 2.2, in some systems bids are compared based on *utilities*, in which case the highest bids win auctions and the system attempts to maximize the global utility function. Utilities often encapsulate multiple factors, some representing the benefit or expected quality of task execution and others representing cost estimates. Cost estimates can also include diverse factors such as the time taken to compute solutions and the loss of efficiency caused by transitioning between tasks. As an example of a utility calculation, Gerkey and Matarić [25] propose subtracting a robot’s *cost* of performing a task from its expected *quality* of completing the task, assuming the units of cost and quality are directly comparable. Thus, utility and cost functions that combine multiple factors often require finding a reasonable set of weights between the different components considered. In general, utility and cost factors might be combined through any arbitrary, possibly nonlinear function.

The process of estimating costs for bid valuation can also be difficult. Though participants in the market may have well-defined cost or utility functions, these functions still rely on having accurate models of the world state and may require computationally expensive operations. For example, the cost to complete the task of driving to a goal site depends on having an accurate map of the environment; however, the robots may be working in an unknown, partially known, or changing environment. When there are multiple goal locations, determining the cost to perform even one task can require solving multiple path planning problems and an instance of the traveling salesman problem (TSP), the latter being \mathcal{NP} -hard. Thus heuristics and approximation algorithms are commonly used, implying that bid prices may not always be entirely accurate. Inaccurate bids can result in tasks not being awarded to the robots best able to complete them. In this case re-auctioning tasks can often improve solution quality.

2.4 The Range of Coordination Approaches

The goal in most robotic application domains is to generate optimal solutions in a timely manner. Unfortunately, many multirobot coordination problems are \mathcal{NP} -hard. The challenges are compounded by team considerations that include operation in dynamic and uncertain environments, inconsistent information, unreliable and limited communication, interaction with humans, and various system and component failures. A spectrum of coordination approaches has emerged to negotiate these demands.

At one end of the spectrum, fully centralized approaches employ a single agent to coordinate the entire team. In theory, this agent can produce optimal solutions by gathering all relevant information and planning for the entire team. In reality, fully centralized approaches are rarely tractable for large teams (the associated planning problems typically grow exponentially with team size and mission complexity), can suffer from a single point of failure, have high communication demands, and are usually sluggish to respond to local changes. Thus, centralized approaches are most suited for applications involving small teams and static environments or easily available global information.

At the other end of the spectrum, in fully distributed systems, robots rely solely on local knowledge. Such approaches are typically very fast, flexible to change, and robust to failures but can produce highly suboptimal solutions since good local solutions may not necessarily aggregate to a good global solution. Applications where large

teams carry out relatively simple tasks with no strict requirements for efficiency are best served by fully distributed coordination schemes.

A vast majority of coordination approaches have elements that are centralized and distributed and thus reside in the middle of the spectrum. Market-based approaches fall into this hybrid category, and, in some instances, they can opportunistically adapt to dynamic conditions to produce more centralized or more distributed solutions. Market mechanisms can distribute much of the planning and execution over the team and thereby retain the benefits of distributed approaches, including robustness, flexibility, and speed [24, 17]. Auctions quickly and concisely assemble team information in a single location to make decisions about distributing resources; in some cases they provide guarantees of solution quality [55, 40]. Market-based approaches may also incorporate methods of opportunistically coordinating subteams in a centralized manner [18, 34]. Nevertheless, market-based approaches are not without their weaknesses. In domains where fully centralized approaches are feasible, market-based approaches can be more complex to implement and produce poorer solutions. In domains where fully distributed approaches suffice, market-approaches can be unnecessarily complicated in design and have greater communication and computation requirements.

The sections that follow discuss market-based multirobot coordination in greater detail along the dimensions mentioned in the introduction. Each section introduces the topic and its challenges, defines the goals and appropriate evaluation metrics, reviews the relevant literature, and identifies remaining research challenges.

3 Planning

In multirobot teams, planning can be required to coordinate robots to accomplish the team mission. Unfortunately, optimal planning problems for multirobot systems are typically \mathcal{NP} -hard [1]. The challenge then is to have tractable planning that produces efficient solutions. Market-based approaches manage this by distributing planning over the entire team to produce solutions quickly. When required or when resources permit, markets can behave in a more centralized fashion and plan over larger portions of the team to improve solution quality. Here, we consider different layers at which planning arises in a multirobot system and how these planning problems are handled by various market-based approaches.

3.1 Related Work

3.1.1 Planning and Task Allocation

Task allocation is the problem of feasibly assigning a set of tasks to a team in a way that optimizes a global objective function. Many special cases of task allocation appear frequently in the literature; here, we offer a general and formal definition that allows us to discuss and compare them.

Definition 1 Given a set of robots R , let $\mathcal{R} := 2^R$ be the set of all possible robot subteams. An **allocation** of a set T of tasks to R is a function, $A : T \rightarrow \mathcal{R}$, mapping each task to a subset of robots responsible for completing it. Equivalently, \mathcal{R}^T is the

set of all possible allocations of the tasks T to the team of robots R . Let $T_r(A)$ be the set of tasks allocated to subteam r in allocation A .

Definition 2 The Multirobot Task Allocation Problem: Given a set of tasks T a set of robots R and a cost function for each subset of robots $r \in \mathcal{R}$ specifying the cost of completing each subset of tasks, $c_r : 2^T \rightarrow \mathbb{R}^+ \cup \{\infty\}$, find the allocation $A^* \in \mathcal{R}^T$ that minimizes a global objective function $C : \mathcal{R}^T \rightarrow \mathbb{R}^+ \cup \{\infty\}$.

Gerkey and Matarić [25] provide a taxonomy for some variants of the task allocation problem, distinguishing between: single-task (ST) and multi-task (MT) robots; single-robot (SR) and multiple-robot (MR) tasks; and instantaneous (IA) and time-extended (TA) assignment. In instantaneous assignment robots do not plan for future allocations and are only concerned with the one task they are carrying out at the moment or for which they are bidding. In time-extended assignment robots have more information and can come up with longer-term plans involving task sequences or schedules. Definition 2 encompasses each of the types of task allocation in the taxonomy, but in general describes TA task allocation. IA allocation can be represented as a special case where all cost functions map to infinity for any subsets of tasks with cardinality greater than one. Further, if we allow the sets of tasks T and robots R to be time dependent (*i.e.* $T(t), R(t)$) and require the objective function be minimized at every instant of time or over the entire history, then the definition also covers online and dynamic domains where tasks and robots may be added or removed over time (see Section 6). This definition also implies that task allocation is \mathcal{NP} -hard in general, as the multi-depot traveling salesman problem is a special case [1].

Market-based approaches distribute the planning required for task allocation through the auction process: each robot or group of robots locally plans the achievement of the offered tasks, computes its costs, and encapsulates the costs in its bids. This process is illustrated in the introduction of this paper for a distributed sensing task on Mars: each robot determined its own cost of visiting different sites. Most existing market-based approaches fall into the SR-ST category in the task allocation taxonomy. Several assume instantaneous assignment (IA) [24, 39, 60, 65, 68], while others allow for time-extended assignment (TA), introducing an additional layer of planning whereby robots sequence [40, 4, 11, 28, 50, 51, 75] or schedule [26, 43, 58] a list of tasks and can therefore explicitly reason about the dependencies between multiple tasks and upcoming commitments. More recently, market-based systems have addressed the allocation of multiple-robot tasks (MR-ST) [29, 45], including human-robot tasks [33]. Definition 2 does not cover all cases of MR tasks: the cost functions for subteams in the definition are assumed to be independent and do not account for the cost dependencies associated with any subteam members participating in another subteam or performing a task on their own. This can be resolved by conditioning the local cost functions on the other allocations of the teammates. Market-based mechanisms for task allocation can also be differentiated as centralized or distributed. Centralized mechanisms have the ability to find optimal solutions (*e.g.* through combinatorial auctions [55, 4]) or provide bounds on solution quality [40], but sometimes require an exponential amount of computation and communication [55]. Distributed mechanisms [11, 28] can act as anytime algorithms and require less computation and communication resources, but

are not guaranteed to find optimal solutions and have no known approximation bounds. *TraderBots* [11] attempts to find a balance between these two approaches by opportunistically allowing “pockets” of centralized optimization to emerge within subgroups of the team when resources permit. In our distributed sensing example, for instance, the team might begin with a suboptimal allocation of sites, perhaps caused by an inaccurate map of the environment resulting in inaccurate bids. At some point during execution (perhaps when map information is more accurate), a robot might find a better distribution of sites for some subset of its teammates. The robot’s motivation for group optimization is that it can pocket the cost difference as profit by winning the tasks from the original holders and subcontracting them to the new holders. Simultaneously, this results in a better team solution. Solution quality and scalability aspects of these different approaches are discussed in more detail in Sections 4 and 5 respectively.

Related problems of allocating constrained subtasks, roles, and multiple-robot tasks can have additional planning requirements:

Allocating Constrained Subtasks In many domains, tasks are temporally constrained with respect to one another. They may be partially ordered or may need to start or finish within a common time frame. For instance, consistency may be important in our Mars distributed sensing task, so we might require that samples from particular sites be collected at the same time. In the case of partially ordered tasks, one can use a central allocator to auction only those tasks whose predecessors have been completed [5, 65]. Alternatively, during assignment, robots can incorporate the cost of meeting constraints into their bids [46]. In terms of Definition 2, a violation of constraints can be modeled as infinite values for local cost functions or the global objective function. Constraints can add another dimension to the bid valuation and auction clearing processes and may thus increase computation requirements. Often, robots must also coordinate during execution to reschedule and accommodate team and task changes that have occurred since the initial allocation [26, 43]. In these cases, robots must be able to determine when and how the rescheduling should occur.

Allocating Roles and Instantaneous Assignment In team games such as robotic soccer one usually assigns *positions* such as “primary offense” or “supporting defense” instead of tasks such as “shoot the ball” or “capture a rebound.” These positions can be classified as *roles*. More generally a role defines a collection of related actions or behaviors. Indeed, in many domains it is more natural to think of teammates playing roles than completing distinct tasks. In market-based approaches, role allocation can use the same *auction-bid-award* protocol as task allocation. However, robots can usually take on only one role at any given time (SR-ST-IA or MR-ST-IA) and generate bids by evaluating a fitness function that reflects how well its current state matches the requirements of the role. Once allocated, a robot locally plans the execution of actions and behaviors specified by its role. Market-based role allocation has been demonstrated in robot soccer [39, 68] and treasure hunt [12] domains.

Instantaneous assignment (IA) also arises in cases where the tasks being allocated are short-term partial actions that bring the team goal closer to being realized. Examples of instantaneous assignment include allocating *push* actions in a box-pushing

application [24] and assigning waypoint locations (that do not necessarily have to be reached) in an exploration scenario [60].

Allocating Multiple-Robot Tasks Assigning multiple-robot (MR) tasks is another challenging variant of the task allocation problem. In SR task allocation, a robot’s bid for a task generally depends only on its own state (including its existing task commitments), and the auctioneer can consider bids individually when making an allocation. In contrast, multiple-robot tasks depend on more than one robot’s state and require some amount of subteam planning unless the tasks can be easily decomposed. MR tasks may require robots to tightly coordinate during execution, an issue we consider separately in Section 3.1.3.

Approaches for MR task allocation generally require joint planning for each task followed by an assignment of robots to the subcomponents of this plan. Ideally, in a market-based approach, the agent doing the joint planning can do both the planning and assignment based on minimal information encapsulated in teammates’ bids. This ensures that the robots only communicate simple bid prices in order to parallelize local planning and individual cost calculations, as well as to reduce communication bandwidth. For instance, one way to handle MR tasks is to assign each task to a single robot that must recruit other teammates to assist it. Chaimowicz *et al.* [9] and Guerrero and Oliver [29] describe two similar approaches using instantaneous assignment in a foraging and object transportation task. In both, robots explore the environment looking for objects. Upon finding one, a robot uses an auction to recruit help in moving the object to a goal location. These helpers are then committed to assist the leader, but can be preempted for another task if the utility is higher. One disadvantage to this type of approach is that upon discovery of a task, it is automatically assigned to a robot that may not be able to find enough assistance. That robot is then committed to the task rather than being able to perform other useful work. Additionally, while this approach can be applied to most situations where robots are discovering or generating the tasks during execution, it is not clear how it would apply to tasks being introduced by external sources (*e.g.* a human operator). Jones *et al.* [33] introduce an auction mechanism that allows robots to solicit cooperation without committing to the tasks. This mechanism is capable of allocating tasks originating from both online robot discovery and external input. In order to avoid premature commitment, each auction round incorporates a second nested round in order to garner information to compute subteam costs. After the initial auction call, each bidder selects one among several alternatives from a list of “plays” that solve the task. Each play consists of several roles and the bidder chooses the lowest cost role for itself. The rest of the roles are then proposed to the other teammates in a non-binding role auction, allowing the initial bidder to subsequently estimate the full cost of the play and submit a bid price to the original auctioneer. Upon being awarded a task, the winner requests commitments from the role-bidders who accept the award unless they have agreed to perform some other task in the time since they submitted their bids. If any role-bidders reject the award, the task winner rejects the award from the initial auctioneer and the task gets re-auctioned. This ensures that no robot accepts an awarded task without a commitment for all relevant roles being filled as estimated during bid valuation. Vig and Adams [70] use a different

mechanism where a task is broken down into a list of required resources or services, then agents representing each kind of resource determine the minimum costs from the robots able to provide some amount of that resource. The resource agents then report these costs to the auctioneer, who then decides if the price is low enough to make the task profitable.

An approach by Lin *et al.* [45] requires that robots submit more complete state information to the auctioneer who then arrives at a plan. Here, the robots send in “capability vectors” describing their current resources and abilities. The auctioneer then offers a “pre-award” to the subgroup that can compete the task for the least cost. The members of the subgroup then communicate amongst themselves and can agree to form a subteam for that task given the auctioneer’s price. After notifying the auctioneer, the actual award is finally sent out. Because the approach involves communicating detailed state information rather than bid prices, this approach loses some of the advantages of market-based coordination, such as distributing planning activities and low communication requirements. Nonetheless, centralized planning may be able to better deal with problems containing complex inter-robot constraints. A different approach by MacKenzie [46] does most planning locally, but requires more information to be communicated in order to solve temporal inter-robot constraints centrally. This is done by supplementing each bid price with information on projected task start times, and including multiple such bids for each task. From the submitted bids, the auctioneer then determines which subset of teammates can coordinate in time and space to collectively complete the task most efficiently. The most notable algorithmic difference between this work and that of Lin *et al.* [45] is that although in both some joint planning is done centrally, most of the planning in Mackenzie’s approach is still done locally when robots compute their set of bids: the auctioneer is merely finding a feasible plan based on a limited set of options provided by the bidders.

3.1.2 Planning and Task Decomposition

Although many approaches to task allocation assume that a list of primitive or simple tasks is input to the system, a complex mission is often more naturally described at a higher level of abstraction. For example, scientists desiring data about Mars may only be concerned with the general regions from which data is collected and not the precise sites. As illustrated in Figure 1, a mission might be phrased as “capture images that collectively show 50% of crater regions A, B, C, and D.” In these cases, multirobot systems must also decompose a mission into subtasks, often making use of well-known planners [5, 7] or domain-specific decomposition algorithms [74].

There are two common approaches to this planning problem. In the *decompose-then-allocate* method, a single agent recursively decomposes the task into simple subtasks which are allocated to the team [7, 65]. In the distributed sensing scenario, this amounts to finding a fixed set of observation sites for all crater regions and then allocating these sites to the team. In the *allocate-then-decompose* method, complex tasks are first allocated to robots then each robot locally decomposes its awarded tasks [5]. This corresponds to assigning entire crater regions to robots and letting each robot choose the sites. It is also possible to include instances of both techniques [26, 5].

By decoupling the decomposition and allocation problems, these approaches do

not consider the complete solution space and may find highly inefficient solutions. In general, one cannot decompose a task optimally without knowing which robots will execute the subtasks, nor can one allocate tasks efficiently without knowing how they will be decomposed. One solution is to simultaneously work on both problems by generalizing tasks to *task trees* and trading these explicitly on the market [74]. In this setting, bidding occurs in two stages. In the first pass, bidders simply evaluate nodes in the offered task tree according to the auctioneer’s plan. Second, the bidders may come up with new decompositions for any abstract nodes in the tree, and use the price of this new plan within their bid if the cost is lower. By using this bidding procedure together with a specialized task tree auction clearing algorithm, both the costs of allocations and plans can be compared in a single auction mechanism. Experiments in complex task domains demonstrate that using task tree auctions can improve solution quality over the aforementioned two-stage approaches [74].

3.1.3 Planning and Task Execution

Many missions including our example distributed-sensing mission on Mars consist of tasks that can be completed independently by individual robots. Such missions often consist of SR tasks and can usually be achieved by a *loosely-coordinated team* in which robots coordinate during task decomposition and allocation but not during execution. Thus, in these domains, planning how tasks should be executed can be done at an individual level without consideration of teammates’ actions and is outside the scope of market-based coordination. (An exception is when robots unexpectedly interfere with each other during execution. Azarm and Schmidt [2] address collision avoidance of independent robots during execution using market-based techniques.)

Another class of problems, however, require *tightly-coordinated teams* in which members continuously coordinate throughout execution. This includes tasks such as the collective manipulation of objects (*e.g.* beams to construct a scaffold) and moving in formation (*e.g.* to safely travel between sites of interest). Tasks requiring tight coordination pose a number of significant challenges. Firstly, these tasks tend not to be easily decomposable (*e.g.* the object the team is transporting cannot be split into pieces and carried independently). Secondly, teammates heavily constrain each other’s choice of actions (*e.g.* they must all move in concert together to avoid dropping the object). Thirdly, systems including these types of tasks are rarely fault-tolerant since task success depends on the simultaneous success of multiple teammates. In total, teams must essentially solve a tightly-coupled multirobot planning problem, but cannot easily take advantage of the distributed planning and execution that make loose coordination tractable and robust.

In many cases, the role of market-based approaches in these missions is limited to allocating the MR tasks to a subset of the team [61]. Market-based approaches are rarely used to actually coordinate the activities of the subset of robots tightly coordinating to achieve each MR task because, in practice, this coordination can be achieved with reactive or behavior-based approaches that forgo planning and are less expensive in terms of design, computation, and communication.

Nevertheless, some domains greatly benefit from and even require advanced planning of the coordination between robots and cannot be solved with simpler emergent

approaches. For example, we may want our Mars rovers to always stay in communication contact with a base station; doing so requires that they tightly coordinate over large distances and plan paths to sites with each others' actions in mind. Reactive and behavior-based approaches cannot achieve the required planning; instead, market-based approaches have recently been developed to address these domains. The idea is to exploit small pockets of centralized planning by having robots buy and sell tightly-coupled joint execution plans over the market (as opposed to buying and selling individual tasks) [34]. This technique is related to the idea of opportunistic centralization discussed earlier. Although such approaches have higher communication and computational demands than the market-based approaches designed for loosely-coordinated teams, they have been shown to outperform competing approaches to the same problems.

3.2 Future Challenges

Multirobot systems typically must incorporate multiple types of planning for different aspects of the problems they address. Market-based approaches are currently capable of many types of planning, but several challenges remain. First, there has been limited work in domains with many complex constraints between tasks and domains requiring tight coordination. Second, efficient replanning is crucial to working in uncertain environments and relates closely to issues to be raised in Section 6. Task reallocation can be achieved by peer-to-peer trading and some progress has been made in redecomposing complex tasks in market-based systems [74], but significant work remains in replanning for tightly-coordinated teams. A third important and relevant area of research (for which some initial work has been done [12]) is understanding the formation of subteams and enabling their positive interaction using market-based methods. Finally, market-based approaches need better strategies for making use of multirobot planners and providing alternatives to combinatorial auctions for vetting complex plans in the market.

4 Quality of Solution

In this section, we look at the solution quality of existing market-based allocation algorithms. Essentially, all auction-based solutions aim to optimize some global cost or utility. Here we focus on theoretical and experimental results that demonstrate how well some of these algorithms achieve that goal. As discussed in Section 3, a fundamental optimization problem encountered in market-based multirobot systems is the task allocation problem. Since task allocation is \mathcal{NP} -hard, system designers face the challenge of choosing market mechanisms that result in the most efficient solutions within a reasonable amount of time. Various global cost objectives (C in Definition 2) appear in market-based systems depending on the application. Most common are minimizing the sum of individual robots' costs ($C(A) = \sum_{r \in \mathcal{R}} c_r(T_r(A))$) [4, 11, 28, 47, 53] or minimizing the maximum individual cost (makespan, $C(A) = \max_{r \in \mathcal{R}} c_r(T_r(A))$) [50, 43]; although others are possible (e.g. minimizing the average overall time to complete each

task [67]). In the Mars distributed sensing example, these global objectives correspond to finding the allocation of sites to robots that results in the least amount of fuel expended (sum of costs) or the task being done in the least amount of time (makespan). While it has been demonstrated that inaccuracies in cost models can affect the quality of solutions obtained [13], the results presented in this section are all developed under the assumption that the robots' cost or utility estimates are accurate.

In this section, we make use of the terms *approximation* and *competitive* ratios. Approximation algorithms find suboptimal solutions to (usually \mathcal{NP} -hard) offline problems where the factor relating the two solutions can be bounded by a known factor.

Definition 3 *An algorithm A is a ρ -approximation for a minimization problem iff for all instances of the problem, the cost of the solution, $C(A)$, is guaranteed to be at most ρ times the cost of the optimal solution, $C(\text{OPT})$, i.e. $C(A) \leq \rho * C(\text{OPT})$. Similarly, for a maximization problem, an algorithm A is a ρ -approximation iff the cost of the solution is guaranteed to be at least $\frac{1}{\rho}$ times the cost of the optimal solution.*

Online problems are those in which the input is not fully specified ahead of time; for example, the Mars scenario becomes an online problem if the site tasks are being introduced when a partial allocation has already been made or even while the robots are executing the task. For online algorithms, a common measure of the solution quality is the *competitive ratio*.

Definition 4 *An algorithm A is ρ -competitive for an online minimization problem iff for all inputs of the problem, the cost of the online solution, $C(A)$ is guaranteed to be at most ρ times the cost of the offline optimal solution, $C(\text{OPT})$, i.e. $C(A) \leq \rho * C(\text{OPT})$. Similarly, for an online maximization problem, an algorithm A is ρ -competitive iff the cost of the solution is guaranteed to be at least $\frac{1}{\rho}$ times the cost of the offline optimal solution.*

4.1 Related Work

As described in Section 3, we distinguish between instantaneous (IA) and time-extended (TA) task allocation. The IA model often arises in cases where tasks require time-indefinite exclusive commitments by robots. Positional roles in robot soccer [39, 68] fall into this category as do “persistent” tasks (*e.g.* in a target tracking scenario, robots must follow a target for as long as possible and therefore do not need to consider taking on future tasks). In both cases, robots will carry out a task or role indefinitely unless it gets reassigned to another robot. At the other extreme are instances where the tasks being assigned are short-lived partial actions that bring the team goal closer to being realized. After some of these actions are carried out, new ones are generated by planning from the observed resulting state. For example, Gerkey and Matarić [24] use an IA approach in a box-pushing domain where with high frequency an auctioneer assigns incremental push actions given the box orientation and the goal location. Similarly, in one approach for an exploration scenario, auctions are used to assign a goal point to each robot on a team, then determine a new set of target points once the first robot reaches its goal (the remaining robots are not obligated to reach their initial

goals) [60]. However, sometimes IA approaches are used for simplicity in lieu of a TA approach; they are easier to implement and do not need computationally expensive task sequencing or scheduling algorithms [25, 29, 7, 10, 36, 59, 44]. In these systems, if there are more tasks than robots, the remaining tasks can be allocated once robots complete their previous assignments. Ignoring dependencies between tasks by using an IA approach should theoretically result in inferior solution quality, but we are not aware of any explicit comparative study.

When IA allocation is appropriate, it has been demonstrated that optimal allocation is possible when there are at least as many robots as tasks, although several existing systems use a 2-approximate greedy solution [25]. Additionally, performance guarantees are not always equivalent for cost- and utility-based systems: the greedy algorithm for the metric online variant of the IA task allocation problem is 3-competitive for utility maximization [25, 37] but scales exponentially with the number of robots for cost minimization [37].

TA sequencing approaches have additional planning and scheduling requirements, but, when appropriate, model the problem more accurately and should produce better results. For example, a combinatorial auction can theoretically result in an optimal allocation if the robots compute and submit bids on all possible combinations of tasks (of which there are an exponential number) [55]. In practice, performance guarantees are sacrificed in order to reduce the computation and communication requirements by considering only a relatively small number of task bundles [4, 47, 31, 16] (details in Section 5).

A simpler centralized mechanism is one in which single tasks are iteratively allocated in multiple auctions until all tasks are assigned [28, 50, 75, 5, 7, 53]. In general, these types of auctions are not guaranteed to find the optimal solution; however, because they require less computation and communication than combinatorial auctions and are easier to implement, they are more prevalent in the literature. Additionally, if costs are considerably uncertain investing the time to produce an optimal solution may not be worthwhile since future assessments are likely to invalidate the optimality of the initial solution.

Tovey *et al.* [67] suggest a hill-climbing heuristic for generating bidding rules for single-task auctions with various global objective functions: essentially each robot bids the difference in *team* cost between the existing allocation and the allocation that would result if the robot wins the offered task. It turns out that for common objective functions robots can calculate these bids using only local information—without requiring knowledge of the states or current assignments of its teammates. This method gives some justification for the typically utilized strategies, as one can derive the commonly used bidding rules encountered in market-based systems: for minimizing total cost (called MINISUM by the authors), this rule advises that bidders should base their bids on the marginal costs of the offered tasks [4, 11, 28, 47, 53]; for makespan minimization (MINIMAX), load balancing can be better achieved if participants instead bid based on their total costs [50]. A third objective function, minimum average latency (MINIAVE) is also suggested, and the generation heuristic suggests that robots should bid the difference between the minimum sum of per target costs with and without the task under consideration. Experimentally, Tovey *et al.* found that the bidding rule derived for each of the three objective functions results in better solutions than the rules

derived for the other objectives. By modeling multirobot problems as vehicle routing problems [8, 27, 42], Lagoudakis *et al.* [40] provide a set of approximation bounds for the same rules and objective functions. They present six results for the three objective functions with two bidding rules derived for each for a total of eighteen approximation bounds. The results prove that bids based on individual *marginal costs* when applied to a sum-of-costs objective results in a 2-approximation, while bids based on individual *total costs* applied to a makespan objective yields an approximation algorithm that scales linearly with the number of robots (which is a worst-case result for *any* makespan algorithm). Examples are also given to show lower bounds for each algorithm, which in some cases show that the approximation bounds are tight.

Task reallocation is also possible by introducing peer-to-peer auctions [11, 28, 43, 53, 14]. In this case, there is some initial allocation, and any robot on the team is capable of holding auctions in order to reallocate tasks to robots that are better suited to perform them. In static environments, distributed trading can improve inefficient initial allocations resulting from the use of faster but suboptimal mechanisms. In unknown or partially known environments where costs constantly change as new observations are made, initial solutions may no longer maintain optimality guarantees or even be reasonably efficient; in such a case the use of peer-to-peer auctions can be used to repair undesirable allocations. Peer-to-peer trading can be viewed as a local search and thus is subject to local optima. Sandholm proves that by using a sufficiently expressive set of contract types (single-task, multi-task, swap, and multi-party), the global minimum can be reached in a finite (but possibly large) number of steps [54], while experiments by Andersson and Sandholm demonstrate that more practical systems that include just single- and multi-task contracts (*e.g.* [16]) find the most efficient solutions given a limited number of rounds [1]. Another interesting result by Vidal [69] shows that by not requiring agents to be purely selfish (*i.e.* some agents may be worse off after some trades) the local search algorithm can circumvent some local optima and in the long run find better solutions. Dias *et al.* [14] look at initializing the team allocation by holding central greedy multi-task auctions (multiple tasks can be awarded in each but at most one task is awarded per robot per auction) before distributed trading begins. They find that increasing the number of tasks awarded per auction can have a negative effect on the resulting solution quality but requires less time (fewer auctions are held) to find a solution. Table 1 gives a summary of the results presented in this section.

4.1.1 Relationship to Operations Research

As has been noted in many publications (*e.g.* [4, 15, 25, 40, 50, 74]), multirobot task allocation problems can be modeled as well-studied problems from the field of operations research (OR), and therefore many of the existing techniques and solutions can be applied to multirobot domains. A simple example is the use of the Optimal Assignment Problem (OAP), which looks at assigning a set of jobs to a set of workers, to model SR-ST-IA task allocation [25].

As mentioned previously, another common manifestation of OR-type problems in multirobot domains is in treating multirobot routing problems as Vehicle Routing Prob-

Table 1: Summary of solution quality results.

Approach	Theoretical guarantees	Experimental results
Combinatorial auctions [4, 47]	Optimal (if all bundles are considered) [55]	Good solutions with limited number of task bundles [4, 47, 31, 16]
Central single task iterated auctions [40, 67]	Approximation bounds for 18 cases (3 objective functions, 6 bidding rules) [40]	Close to optimal results when using the appropriate bidding rules [67]
Central instantaneous assignment (IA) [24, 39]	Optimal possible; commonly used greedy algorithm is a 2-approximation; greedy algorithm for online version is 3-competitive [25]	
Peer-to-peer trading [11, 28, 50, 75, 53]	Optimal solution possible in a finite number of trades with a sufficiently expressive set of contract types [54]	In a limited number of rounds, a combination of single- and multi-task trades outperforms all other combinations of single-task, multi-task, swap, and multi-party contracts [1]; allowing non-individual rational trades can lead to better solutions [69]
Central multi-task auctions followed by peer-to-peer trading [14]		Increasing the maximum number of tasks awarded per multi-task auction results in poorer solution quality [14]

lems (VRP)² [27]. Variants of the Traveling Salesman Problem (TSP), a type of VRP, have been mentioned as problem domains in several market-based coordination publications [4, 15, 40, 50, 74]. The multirobot variants of the TSP require a team of robots to visit a set of target points (each point must be visited by at least one robot) while incurring a minimum team travel cost. In the simplest case the relevant combinatorial optimization problems are the k -TSP [21, 52] and the multi-depot TSP [8, 42]. The distinction between the two problems is that in the k -TSP the robots must all start at the same point, whereas in the multi-depot version the robots can start at different locations. Another consideration is whether the robots are required to return to their start

²The problem does not have to literally be a transportation-type problem. For example, any problem in which the cost to perform a task depends only on the state of the robot upon completing the previous task is analogous to a Traveling Salesman Problem.

points upon completion of their tasks or not. In the latter case, the problem is called the Traveling Salesman Path Problem (TSPP) [30]. Other variants of the VRP such as consideration of finite vehicle capacities, global vehicle fuel constraints, or precedence constraints between tasks (as in the existence of pickup and delivery points) may well be useful in future multirobot research.

In some cases, existing algorithms from the OR literature may give us insight on auction-based algorithms used in the multirobot community. For example, Gerkey and Matarić [25] demonstrated that a 2-approximate greedy assignment algorithm is being used in several existing multirobot systems (*e.g.* [18, 68]) despite the fact that a polynomial-time optimal algorithm already exists. In other instances, one may find an existing algorithm with desirable theoretical properties that can be readily converted to a distributed auction algorithm. For example, Cerdeira’s multi-depot TSP 2-approximation [8] can be easily converted to an auction-based algorithm [41].

4.2 Future Challenges

While some theoretical guarantees for simple auctions are known, future work should address the more complex mechanisms that are present in implemented systems which can include online, multi-task, peer-to-peer, simultaneous, and overlapping auctions as well as task and scheduling constraints. Additionally, solution quality depends on accurate cost and utility measures which may be very challenging to acquire. Although some progress has been made in methods for learning [58] and improving [13] these estimates, further work is required.

5 Scalability

Scalability is an important consideration for any multirobot coordination approach. In general, a system is scalable if it can operate effectively even as the number of inputs or the size of inputs increases arbitrarily. The scalability of a multirobot coordination approach is typically evaluated by its ability to produce efficient solutions as the team size or the task complexity increase. For example, in the Mars distributed sensing scenario, a scalable coordination approach will continue to produce efficient task allocations as the number of robots in the team and the number of sensing tasks assigned to the team increase. Scalability in some market-based approaches may be limited by the computation and communication needs that arise from increasing auction frequency, bid complexity, and planning demands. However, market-based approaches can scale well in applications where the team mission can be decomposed into tasks that can be independently carried out by small subteams.

5.1 Related Work

In Section 4 we highlighted the tradeoffs between scalability and solution quality in market-based systems. In this section, we first elaborate these tradeoffs with an analysis of the computation and communication requirements of the different types of auctions we initially introduced in Section 2. We then describe how market-based ap-

proaches can more effectively negotiate this tradeoff by dynamically responding to the changing demands of the task and adaptively utilizing communication and computation resources.

5.1.1 Computation and Communication Considerations

Single-item auctions are usually computationally feasible and light on communication, but they produce suboptimal solutions. Only one task, resource, or role description is required to be included in the auction call, to be planned for and bid on, and to be considered during auction clearing and awarding. Combinatorial auctions can produce optimal solutions, but can require an exponential amount of computation and communication. This is due to the fact that there are an exponential number of bundles for the bidders to consider making bid valuation, bid submission, and winner determination potentially intractable. Multi-item auctions are also computationally manageable, but produce inferior solutions as compared to single-item and combinatorial auctions. Although there are multiple items included in an auction in this case, bids are only considered for each item independently. Therefore, there is only a linear increase in communications and bid valuation. One advantage of multi-item auctions is that more items are awarded per auction so items can be allocated quickly since less auctions are required. When considering these tradeoffs it is also important to consider the problem domain: for highly uncertain or dynamic environments, it may not be worth spending the time to compute an optimal solution if that solution will constantly be changing as more information is gathered; or, if there are hard real-time constraints, there may not be enough time to compute an exact solution.

Table 2 summarizes the time complexities of the important phases of several auction types. For each protocol, the table lists the maximum number of bid valuations, the computation times of the best-known auction clearing algorithms, and the number of auctions required to allocate all items to the team (if the objective is to offload all items from the initial auctioneer—this may not be the aim in peer-to-peer auctions as the auctioneer may retain some items or some reserve prices may not be met). Table 3 similarly gives a summary of communication costs. Of particular interest in these tables are the exponential expressions for combinatorial auctions in the areas of bid valuation, winner determination, and bid submission. As discussed previously, in order to make combinatorial auctions scalable, the common approach is to limit the number of bundles that are considered during bid valuation. This in turn reduces the number of bids that need to be communicated and also allows the auction to be cleared quickly in practice. Indeed, Sandholm’s optimal clearing algorithm, CABOB [55], relies on a sparse bid set in order to find solutions quickly (although the resulting allocation is still likely to be suboptimal given that not all item bundles are considered). The number of bundles can be reduced by the auctioneer, the bidders, or both. The auctioneer may offer only a limited set of bundles, for example, by grouping items that may require similar resources [31], or by exploiting hierarchical problem structure to form the bundles [74]. On the bidders’ side heuristic clustering algorithms (*e.g.* nearest neighbor) are often used [4, 16]. Another strategy is to consider only those bundles that are smaller than a given size [4, 47]. Berhault *et al.* [4] compare four clustering algorithms for goal point tasks and find that one based on repeated graph cuts outperforms

a nearest-neighbor algorithm and two algorithms based on limiting cluster size.

Table 2: Comparison of time complexities of various auction types. n is the number of items, r is the number of bidders, b is the number of bids, and $m \leq r$ is the maximum number of awards per auction (for multi-item auctions). v and V represent the (domain-dependent) amount of time required to perform a valuation for a single item (v) or set of items (V).

Auction type	Bid valuation	Winner determination	Number of auctions
Single-item	v	$O(r)$	n
Multi-item (greedy)	$O(n \cdot v)$	$O(n \cdot r \cdot m)$	$\lceil n/m \rceil$
Multi-item (optimal)	$O(n \cdot v)$	$O(r \cdot n^2)$ [25]	$\lceil n/m \rceil$
Combinatorial	$O(2^n \cdot V)$	$O((b+n)^n)$ [55]	1

Table 3: Comparison of communication complexities of various auction types. Notation is the same as in Table 2. This assumes constant message space for each task description.

Auction type	Auction call	Bid submission	Award	Award (+ losers)
Single-item	$O(r)$	$O(r)$	$O(1)$	$O(r)$
Multi-item	$O(r \cdot n)$	$O(r \cdot n)$	$O(m)$	$O(r)$
Combinatorial	$O(r \cdot n)$	$O(r \cdot 2^n)$	$O(n)$	$O(r + n)$

Bid valuation itself may be computationally expensive as the process usually involves some amount of local planning. That is, the expressions v and V in Table 2 may in reality represent the running time of algorithms that must solve a difficult or even \mathcal{NP} -hard problem in order to estimate costs [4, 11, 28, 50, 53]. Additionally, expensive task decompositions may be required at the bidding stage [74]. In the Mars exploration scenario, bid valuation can be expensive since evaluating the cost of a set of target points may require solving two traveling salesman problems (TSP): one to determine the cost of the rover’s path after including the offered tasks, and one to find the cost without the new tasks. Heuristics and approximation algorithms can help deal with the \mathcal{NP} -hard problems (*e.g.* TSP approximation algorithms for task sequencing [18]), although when there are many items to consider simultaneously—either from auctions offering many items [4, 11, 28, 43, 74, 47] or from multiple robots holding auctions simultaneously [11, 74]—bidders can still be overburdened with valuation problems. As a result, system designers must ensure that bidders are able to meet auction deadlines and do not tax their processors to the point of compromising real-time requirements.

As mentioned in Section 4, bid valuation computation can be less costly for instantaneous allocation problems (IA) as there is no need for an explicit task sequencing or scheduling algorithm. However, if the problem is inherently a time-extended one (TA), ignoring scheduling in general results in poorer quality solutions.

Busquets and Simmons [6] offer an offline learning approach to improve the scalability of multi-item auctions: by keeping track of histories, the auctioneer can reduce the number of tasks offered if it believes it is unlikely to get any bids for some of them, while a bidder can reduce the number of bids submitted by removing those for tasks it believes it is unlikely to win. Reducing the number of offered tasks in a multi-item auction decreases the amount of communication in the auction call and bid submission, plus the amount of computation in bid valuation and winner determination. Reducing the number of bids decreases bid submission communications and winner determination computation. In this approach, each bidder computes its bid price on a task and then stochastically decides whether to submit its bid to the auctioneer. The probability of submission is equal to the proportion of previous bids at or below the current bid value that successfully won their auctions. Similarly, an auctioneer stochastically decides to offer a task based on the proportion of past bids that have exceeded the reserve price. In a variation of the Mars distributed sensing scenario, this learning algorithm results in significant scalability improvements by reducing the number of messages communicated and the number of task valuations performed.

A scalability comparison of market-based, behavior-based and centralized approaches is presented by Dias and Stentz [11] on a distributed sensing task in which the goal is to visit the last observation site in as little time as possible (*i.e.* a makespan objective). Simulation experiments demonstrate that the market approach can provide significantly higher-quality solutions than behavior-based approaches while using significantly less computation time than centralized approaches. Xu *et al.* [72] compare market- and token-based coordination approaches, and further introduce a hybrid approach in which auction calls (tokens) are sent only to the agents most likely to submit the best bids. This requires agents to maintain a model of the team state in order to make intelligent token-routing decisions. Their experiments show that the hybrid approach offers an interesting tradeoff, producing slightly less efficient solutions than the full auction approach, but with much reduced communications (slightly worse than the token approach).

Tilley and Williams [66] study several scalability effects in a simple parts manufacturing system. In this application, agents bid for machining tasks for which execution times do not depend on task order and a simple scheduling heuristic is used: each part is processed in the order it is allocated. One method they employ to reduce task evaluation time is to limit the number of tasks considered at any time by each machine; that is, if the number of current tasks plus outstanding bids exceeds some small constant that machine ignores all incoming auctions. Decreasing this limit decreases the average number of auction participants and increases the number of auctions that go unbid for (thus decreasing the solution quality), but also lowers the required auction deadlines (time required for all bids to be submitted). They also conclude that the time required to evaluate a task has a greater effect on required auction deadlines than does system task load, and that task valuation times should be much smaller than task inter-arrival time for efficient system performance.

5.1.2 Selective or Opportunistic Centralization

Centralized planning has the potential to produce optimal team solutions, but most centralized algorithms have complexity exponential in the number of robots and quickly become intractable for more than a few team members. Nevertheless, market-based approaches can selectively use centralized planning to scalably improve solutions over subparts of the team. For example, in the *TraderBots* architecture, a leader robot may replan the allocation of a subset of tasks contracted to some subset of the team when time and computation resources permit. If this leader discovers a reallocation that is better than the current allocation, it can purchase the tasks from their current holders and subcontract them according to its new allocation. The difference in value between the two allocations minus the required payoffs can be pocketed by the leader as profit. This is similar to the coalition formation problem addressed by Sandholm and Lesser [56] in which agents representing trucking depots could form coalitions to pool their delivery tasks together in order to find more efficient solutions for both parties.

The Hoplites framework uses a similar approach to solve tightly-coupled problems in which robots continuously constrain each other's actions (*e.g.* to remain in line-of-sight contact). Robots begin with a simple coordination strategy which is light on communication and computation and allows teammates to iteratively respond to each other's actions without directly influencing them. When this traps robots in local minima, a more powerful coordination mechanism improves solutions by enabling robots to influence each other directly by purchasing each other's participation in complex plans over the market. This consumes more resources but is also capable of producing much better solutions. Hoplites selectively injects pockets of complex centralized coordination into the system only when necessary; as a result, it provides improvements in solution quality while remaining computationally competitive with much less sophisticated approaches.

5.2 Future Challenges

While much is known theoretically about the scalability of various auction mechanisms, market-based approaches have yet to be implemented on teams of more than a few robots. Larger teams have been used in simulation however; for example, Xu *et al.* [72] consider teams of 100 agents. Further challenges exist in improving opportunistically centralized approaches' means of selecting task clusters and team members to reduce unnecessary computation. The challenge of dealing with limited computation when faced with an excess of solicited bid valuations is also largely unaddressed. The recent work of Busquets and Simmons [6] has made some progress in this area.

6 Dynamic Events and Environments

Operations in dynamic and uncertain environments pose a variety of challenges to team coordination such as ensuring graceful degradation of solution quality with failures, enabling team functionality despite imperfect and uncertain information, maintaining effective response speed to dynamic events, and accommodating evolving conditions

and constraints. Successful team operations in dynamic and uncertain environments is therefore highly dependent on achieving robustness. Benchmarks for the robustness of a coordination approach must take into account the diversity of failures the team can accommodate, the requirements for quantity, quality, and certainty in information, the team’s response speed to dynamic events, the fluidity of the team (that is, the ability of the team to accommodate the addition of new members and the loss of current members), and the overall solution quality produced by the team in the face of dynamic events. In this section we examine the different ways in which market-based multirobot coordination approaches to date deal with these conditions.

6.1 Related Work

The application of market-based coordination to dynamic domains is in its early stages. However, several research groups have already made early contributions in this area. We discuss related work here in three categories. The first section on robustness and fluidity deals with a team’s ability to handle agents dynamically joining the team. We then discuss online tasks which require a team to adapt to tasks being added and removed from the mission during task execution. Finally, we explore related work that deals with a team’s ability to operate with limited information.

6.1.1 Robustness and Fluidity

Robotic systems are often complex and their physical interactions with the environment make them highly prone to failures. Thus, a successful coordination approach must gracefully degrade solution quality when failures occur. For example, in the Mars distributed sensing scenario, it is likely that one or more of the rovers will suffer some form of hardware or software failure. A rover can wander outside the range of communication from other rovers, damage a scientific instrument when performing a sensing task, or become completely disabled due to a rockslide. A successful coordination approach will allow the team to accomplish the mission despite these failures if the remaining resources of the team allows for mission completion. Three principal categories of faults that coordination approaches must consider are communication failures, partial malfunctions, and robot death [17]. A variety of strategies are employed by market-based approaches to handle these failures. These strategies are explored in the following sections.

Teams can often perform more effectively if teammates can communicate [75]. However, communication failures occur in a variety of domains and range from occasional loss of messages to loss of all communication. In the *TraderBots* approach, if the team is informed of a common task, consequent disruptions in communication are gracefully handled by using opportunistic auctioning solely for improving solution quality [17, 75]. Sheng *et al.* [59] also consider disruptions due to limited-range communication in the context of multirobot exploration. Here, utility functions incorporate a term that depends on inter-robot distance in an attempt to actively keep the robots closer together and limit the amount of resulting communication loss. As with communication disruptions, partial malfunctions limit a robot’s capability but retain the robot’s planning ability. *TraderBots* employs active reasoning about failed resources to allow

robots to reallocate tasks that they can no longer complete due to malfunctions [17]. Similarly, MURDOCH relies on monitoring progress in short-duration tasks to detect and respond to faults [24]. In the case of robot death, the affected robot cannot aid in the recovery process. However, robots can monitor progress or heartbeats of teammates and re-auction tasks previously assigned to the dead robot [24, 17]. A further improvement is to allow repair of malfunctioning robots and enable their return to the team. In order to accomplish this, a coordination approach must accommodate both the exit of malfunctioning robots and the entrance of the repaired robots. Bererton *et al.* demonstrate reasoning about assisting malfunctioning robots and towing of disabled robots to a base station for repair [3], while Dias *et al.* [17, 18] and Gerkey and Matarić [24] demonstrate both the ability to accommodate the loss of a robot and the re-entry of a repaired robot or the entry of a new robot to the team.

When introducing robustness under dynamic and uncertain conditions, coordination approaches must wrestle with several tradeoffs. For example, if detecting robot death relies on monitoring a heartbeat, a robot is presumed dead if its heartbeat is not received by its teammates within a pre-determined interval. However, if this interval is too large, the time to detect and respond to a failure is increased and the solution quality can degrade as a result. Instead, if the interval is too short solution quality can still degrade due to false positives that can arise if the robot temporarily drops out of communication range.

Another important design consideration when operating under dynamic conditions is choosing auction deadlines. A common practice is to specify a deadline at which time the auction is cleared regardless of whether or not all bids have been submitted. This is required in realistic situations where communications are imperfect, or team members may fail, enter, or leave the team. This may also be necessary in situations where bidders do not bid on all offered tasks, which can be the case for a number of reasons. First, a robot may not be able to perform an offered task due to limited capabilities. Second, a robot may not be able to perform a task for less than a reserve price or for less than a known competitor’s price; for example, another bidder’s bid in an open-cry auction. Finally, bidders may choose not to bid on tasks they are unlikely to win. The tradeoff to consider with respect to auction deadlines is that the deadline must be long enough for all interested participants to submit their bids, while not being overly long such that the slow turnaround time for an auction results in inefficiency due to an insufficient amount of task reallocation.

Determining an appropriate frequency for running auctions is another design consideration that involves a tradeoff. If auctioning happens too frequently, communication networks can get overloaded and robots may spend more time running their own auctions than effectively participating in other on-going auctions. However, if auctions are not held with sufficient frequency, the team may be slow to respond to dynamic conditions. Effectively navigating these tradeoffs and their implications remains an open research challenge.

6.1.2 Online Tasks

In many dynamic application domains, the demands on the robotic system can change during operation. Operators of a multirobot system may submit new tasks or alter

or cancel existing tasks during operation [18]. Alternatively, robots may generate new tasks during execution as they observe new information about their surroundings. In the Mars scenario, a scientist may choose to add new sites to explore or eliminate existing sites while reviewing incoming data, or the robots themselves may have the capacity to make such decisions. Market-based approaches can often seamlessly incorporate online tasks by auctioning new tasks as they are introduced by an operator [11] or as they become available due to the completion of preceding tasks [24, 60, 5, 7]. In some cases new tasks can be generated by the robots themselves and inserted into their plans to be executed or subsequently traded [18, 51, 75, 29]. The solution quality of some simple online auction-based task allocation algorithms are given in Section 4. In general, online versions of optimization problems are more difficult to solve than their offline counterparts.

6.1.3 Uncertainty

Most real-world multirobot applications require operation with only partial or changing information about the environment, the team, and the task. For instance, in the Mars example, it is likely that the rovers will not have access to a complete detailed map of their environment before they begin their mission. Fortunately, market-based approaches have few requirements for prior information and can accommodate new information through frequent auctioning of tasks and resources. The *TraderBots* approach demonstrates that robots can execute tasks with no initial map information and dynamically reallocate tasks when new map information is gathered [18, 75].

6.2 Future Challenges

While much can be done to improve the operation of market-based approaches in dynamic environments, a few key challenges are paramount. Effective information sharing among team members in market-based approaches is one necessary area of research. If a robot discovers a task is expensive because of new environmental information it has gathered, it can potentially allocate the task to another robot that does not have that information. This robot will then try to execute the task until it, too, perceives the new information; then it tries to allocate the task to another robot. This can continue until all robots have attempted to perform the task, causing tremendous inefficiency.

Characterizing the ability of both the individual robots and the team to respond quickly to dynamic conditions using market-based coordination approaches is another important challenge. The authors are not aware of any study of individual or team response speed for any market-based multirobot coordination approach. Other challenges for improving robustness are developing more sophisticated methods for cooperative handling of partial malfunctions and repairs, evaluating robustness to a variety of failures, incorporating contract breaches with appropriate penalties, and incorporating sliding autonomy into market-based approaches to allow robots to request assistance when appropriate.

Formalizing design principals to achieve guaranteed performance within specified bounds despite dynamic conditions is also an important research challenge for market-based team coordination. Specifically, we need more principled methods for navi-

gating the various tradeoffs encountered when designing a market-based coordination approach that must perform effectively under dynamic conditions.

7 Heterogeneous Teams

A team is heterogeneous if not all of its members are equally capable of performing all the tasks (*e.g.* because of hardware or software differences) or if its members play different roles (*e.g.* in team games where robots play different positions). In contrast, the members of a homogeneous team have identical skills or are generalists that can perform all necessary tasks. Heterogeneity is highly advantageous for several reasons. First, complex missions often have many different functional requirements and can be achieved more effectively by a team of specialists rather than by a team of generalists that perhaps cannot perform any single function very well. For example, in our Mars scenario we may want high-resolution images taken, rock samples collected, and core samples taken from the ground. A complex mission such as this is often better achieved with robots that specialize in particular tasks: some robots can take samples from both rocks and the ground while others only capture images. Second, it is often more practical to design robots that specialize in only a small set of skills than to design robots that are capable of all skills. Indeed, in many domains, it may be infeasible to construct robots that can do everything, for example because of limitations in budget, form factor, or on-board power. Third, by being able to coordinate heterogeneous teams, we can reuse robots across multiple applications. Ultimately, a truly heterogeneous team is not limited to robots in its membership. Human-robot-agent teams must operate seemlessly and efficiently in several application domains. Although heterogeneous team coordination is challenging, a successful approach should accommodate any team composition.

7.1 Related Work

In a heterogeneous team, robots have different abilities to perform different tasks. Task and role allocation in heterogeneous teams becomes challenging because it requires reasoning about and comparing different robots' capabilities. Market-based approaches are well suited to meet this challenge because auctions can simplify the problem of reasoning about team skills. When a task or role is auctioned, each robot's bid encapsulates its ability to complete the task in terms of resource usage, the estimated solution quality afforded by these resources, or even the opportunity cost of forgoing other tasks [58]. Additionally, robots can abstain from bidding on tasks for which they do not have sufficient resources, thereby reducing the computational burden for both bidder and auctioneer [24, 31]. The bids can also encode the solution quality afforded by each member's resources, and even the opportunity cost of forgoing other tasks [58]. The auctioneer can award roles or tasks to team members according to the best bid, without requiring knowledge of individual capabilities. Thus, market-based approaches only require each team member to recognize its own skills and resources but not necessarily those of teammates. However, auctions introduce a new difficulty: it is not always clear how to compute and compare the cost of performing a task between different types of

robots who may perform the task in very different ways. One idea is to allow robots to swap tasks directly whenever such a trade results in a mutually beneficial outcome [28]. This circumvents the pricing problem but also severely restricts the number of possible solutions. Thus, the pricing problem remains an open research challenge.

Market-based coordination of heterogeneous teams has been demonstrated on physical robots in a few applications: in automated assembly using three robots with very different physical configurations and capabilities [61]; in box-pushing, where a resource-addressed messaging protocol allows robots to determine in which auctions they should participate [24]; and in treasure hunt, where a human, two Pioneer I IDX robots with laser scanners, and a Segway RMP robot with vision sensing cooperate to seek objects of interest in an unknown environment [33]. In simulation, market-coordinated robots with different science instruments characterize multiple rock types in a space application similar to our Mars scenario [11, 58], and market-based role allocation mechanisms have appeared in heterogeneous robot soccer teams [39, 23]. In work by Lin *et al.* [45] each robot submits a list of its capabilities to the auctioneer who then comes up with a joint plan and sends an offer to the best subteam found who can then accept or reject the task at that price.

7.2 Future Challenges

Ultimately, coordination approaches must accommodate three levels of heterogeneity: heterogeneous robot teams, human-robot teams, and highly heterogeneous teams of humans, robots, and other agents. Future research challenges include modeling human preferences using appropriate reward functions, developing techniques for consistently computing different robots' costs for completing tasks, enabling *pickup teams*, *i.e.* dynamically formed heterogeneous teams where little may be known *a priori* about the task, the robots, or the environments [12], and addressing the challenges of human-robot teams where tasks are understandable to humans and robots and both participate in task allocation and execution [12].

8 Learning and Adaptation

While a generalized system is useful, its application to specific domains usually requires some adaptation. Online opportunistic adaptation is therefore a highly relevant and useful feature. However, in dynamic environments where teams can fluidly change in size, where interaction strategies can be continuously modified, and where external conditions can be unexpectedly changed, *a priori* definitions of best trading and coordination strategies can be very difficult, and sometimes impossible. Consequently, the robots require not only the ability to quickly adapt their behaviour in response to dynamic events and to changes in the other agents' behaviour, but also the ability to determine when and how this adaptation should take place. Hence, the integration of learning techniques can be a very powerful feature.

8.1 Related Work

The application of learning techniques in market-based coordination is currently at a very early stage. One big debate is whether learning should be applied at the team level or at the individual level or some combination of the two. Another important question to be answered is how to deal with team interactions – should other agents be dealt with as environmental factors or should they be dealt with in a special way? Oliveira *et al.* [48] present a detailed discussion of the issues relevant to the application of learning in dynamic markets. The role that learning can play in market-based multirobot coordination is also discussed briefly by Stentz and Dias [64].

The authors are unaware of any learning techniques implemented on a team of physical robots coordinated using a market-based approach. However, publications are starting to emerge in the application of learning techniques for market-based coordination of simulated robot teams. Notably, learning techniques are applied to learn bidding strategies in dynamic markets [48], opportunity costs in a simulated distributed sensing task [58], and role assignment [39] and bidding strategies [22] in simulated robot soccer.

8.2 Future Challenges

The application of relevant learning techniques to market-based coordination of robot teams is a wide open research area with tremendous potential for improving team performance in dynamic environments, reducing the requirement for accuracy in cost estimation and *a priori* knowledge, and enabling easy portability to different domains and environments.

9 Practical Considerations

Many practical considerations affect the overall impact of coordination approaches. Approaches that are general and applicable to a variety of domains are more useful in the real world. A general approach will be flexible across application domains and extensible to enable portability and easy enhancement of functionality. Other important considerations include implementation guidelines for different domains and useful comparisons of different approaches to guide the selection of the most effective coordination scheme for a given application.

9.1 Related Work

Here, we discuss several groups of work that are related to the idea of generality and practical impact.

9.1.1 Flexibility

Since different applications will have different requirements, a widely applicable coordination approach will need to be easily configurable for the different problems it proposes to solve. Instructions and advice on how to reconfigure the mechanism for

different applications will also be useful. Identifying important parameters that need to be changed based on the application requirements, instructions on how to change them, identifying components of the mechanism that need to be added/changed based on application requirements, and instructions on how to make these alterations are all important elements of a successful coordination mechanism. A further bonus will be well-designed user interfaces and tools that allow plug-and-play alterations to the coordination mechanism and automated methods for parameter tuning. The authors are aware of only three market-based approaches, MURDOCH [24], Hoplites [34, 35] and *TraderBots* [11, 75, 73], that have been demonstrated in more than one application. However, there is much that still needs to be done in terms of providing a flexible market-based multirobot coordination approach.

9.1.2 Extensibility

The ability to easily add and remove functionality is a key characteristic to building a generalized system that can evolve with the needs of the different applications. A common approach to incorporating extensibility is to build the system in a modular fashion so that different modules can be altered or replaced relatively easily according to the requirements of the specific application. In market-based approaches, it is best to modularize and isolate cost and reward functions as much as possible from task and role specifications, communication protocols, and task executives.

9.1.3 Implementation

As with any claim, a proven implementation is most convincing. Moreover, successful implementation of a coordination mechanism on a robotic system requires discovering and solving many details that are not always apparent in theory, simulation and software systems. Finally, implementation of an approach on many different platforms in a variety of application domains provides valuable insights and guidelines on how to design and implement different components of the approach in a extensible and flexible manner. Although several have been implemented on physical robot teams (*e.g.* [11, 24, 75, 73, 34, 61, 60]), market-based approaches have yet to be proven in a wide variety of domains.

9.1.4 Comparisons

Comparisons are important to provide guidelines on how to evaluate different coordination approaches when deciding which approach is best for a given application. However, comparing different coordination approaches is a highly challenging endeavor since many considerations need to be addressed.

Some of the challenges in providing a comparative framework for coordination approaches are explored by Gerkey and Matarić [25] who provide an initial framework for evaluating task allocation schemes in terms of complexity and optimality, showing that market-based methods perform favorably in terms of computation and communication requirements. Dias and Stentz [11] compare a centralized optimal approach,

a distributed behavioral approach, and a market-based approach, evaluated in a distributed sensing scenario. Simulation results compare the three approaches in scalability and heterogeneity, and show that all three approaches perform well with heterogeneous teams, and the market method performs best overall in scalability. Rabideau *et al.* [50] also conduct a similar comparative study between a centralized planner that does not guarantee optimality, a distributed planner, and a single-task auction approach. They conclude that the auction approach performs best but takes up the most CPU cycles.

More recently, Xu *et al.* [72] compare a market-based, a token-based, and a market-token-hybrid approach. They find that a market-based solution finds more efficient solutions than a token-based one, but requires more communication. The hybrid approach falls somewhere in between the pure market and token approaches in both areas. Kalra and Martinoli [36] compare the performance of a market-based and a threshold-based approach to IA task allocation along several dimensions. They find that the accuracy of task and robot state information can play an important role in determining the relative effectiveness of these approaches. In sum, market-based approaches have performed well in comparative studies; nevertheless, these studies are fairly limited and broader studies are in high demand.

9.2 Future Challenges

Market-based multirobot coordination approaches have only been implemented and tested in a few application domains to date. Thus, understanding and implementing generality in market-based approaches still requires significant work. However, the growing popularity of market-based methods for coordinating robot teams will be a large contributing factor to inspiring generality in this research area.

10 Conclusions and Future Directions

The vision that drives research in multirobot systems is that teams of robots will inevitably be an integral part of our future. To realize this vision, robots must be capable of executing complex tasks as a team. While many multirobot coordination approaches have been proposed by the research community, market-based approaches in particular have been proven effective; their resulting increase in popularity over the past few years warrants a survey of the field. Many multirobot coordination approaches have been proposed by the research community, market-based approaches in particular have been proven effective and as a result have grown in popularity over the past few years, consequently warranting a survey of the field. We address this need by providing the first survey of the state of the art in market-based multirobot coordination approaches with three contributions to the multirobot literature: a tutorial on market-based multirobot coordination approaches, a review and analysis of the relevant literature, and a discussion of remaining challenges in this research area.

The existing work in market-based multirobot coordination ranges from theoretical formulations to conceptual design frameworks to implementations in simulation and on physical robot teams. The chosen application domains span a wide range and include distributed sensing, mapping, exploration, surveillance, perimeter sweeping, assembly,

box-pushing, reconnaissance, soccer, and treasure hunt. However, this is still a relatively new area of research, and hence many research challenges still remain. Here, we discuss some of the overall challenges in the field.

A first important need is a principled formalization of market-based coordination approaches. Much research is needed to further our understanding of how components such as cost and reward functions, bidding strategies, and auction clearing mechanisms can be designed, implemented, and used effectively in different multirobot application domains. Understanding the tradeoff between solution quality and scalability when designing and implementing coordination mechanisms is also important.

Additionally, much work still remains in rigorously comparing different coordination approaches to enable users to select an appropriate approach given a particular multirobot scenario. First, a relevant set of benchmarks must be defined for effective comparison of different coordination approaches. Second, comparisons must occur between a wider range of approaches, with a greater variety of tasks and team compositions, and across a broader set of metrics. Third, approaches must be compared on physical robot teams as well as in simulation to validate their performance under real-world conditions.

A final challenge is to demonstrate long-term, reliable, and robust operation of larger robot teams in the real world. This will require simultaneous use of many of the techniques discussed in this paper, including learning and adaptation, scalability, fault detection and tolerance, handling uncertainty, and enabling heterogeneity. While many of these features have been demonstrated on a small scale, holistic implementations covering the spectrum of these features are now required.

Despite the many challenges ahead, market-based techniques are proving to be versatile and powerful coordination schemes for groups of robots executing complex tasks as part of a team. Different application requirements and tradeoffs in implementation make it difficult to construct a single market-based approach that can be successful in all domains. Nevertheless, a well-designed market-based approach with sufficient plug-and-play options for manually or automatically altering different tradeoffs can be successful in a wide range of applications. And, with further research, market-based approaches promise to significantly further our vision of robots playing an integral role in human life.

Acknowledgments

This work is sponsored in part by the Boeing Company Grant CMU-BA-GTA-1, in part by the U.S. Army Research Laboratory, under contract **Robotics Collaborative Technology Alliance** (contract number DAAD19-01-2-0012), and in part by the Qatar Foundation for Education, Science and Community Development. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Boeing Company, the Army Research Laboratory, the U.S. Government, or the Qatar Foundation.

References

- [1] M. Andersson and T. Sandholm. Contract type sequencing for reallocation negotiation. In *International Conference on Distributed Computing Systems*, 2000.
- [2] K. Azarm and G. Schmidt. A decentralized approach for the conflict free motion of multiple mobile robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1996.
- [3] C. Bererton, G. Gordon, S. Thrun, and P. Khosla. Auction mechanism design for multi-robot coordination. In *Advances in Neural Information Processing Systems*, 2003.
- [4] M. Berhault, H. Huang, P. Keskinocak, S. Koenig, W. Elmaghraby, P. Griffin, and A. Kleywegt. Robot exploration with combinatorial auctions. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2003.
- [5] S. S. C. Botelho and R. Alami. M+: A scheme for multi-robot cooperation through negotiated task allocation and achievement. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1999.
- [6] D. Busquets and R. Simmons. Learning when to auction and when to bid. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS)*, 2006.
- [7] P. Caloud, W. Choi, J. C. Latombe, C. L. Pape, and M. Yim. Indoor automation with many mobile robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 1990.
- [8] J. O. Cerdeira. The multi-depot traveling salesman problem. *Investigaçāo Operacional*, 12(2), 1992.
- [9] L. Chaimowicz, M. F. M. Campos, and V. Kumar. Dynamic role assignment for cooperative robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.
- [10] P. Chandler and M. Pachter. Hierarchical control for autonomous teams. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, 2001.
- [11] M. B. Dias. *TraderBots: A New Paradigm for Robust and Efficient Multirobot Coordination in Dynamic Environments*. PhD thesis, Robotics Institute, Carnegie Mellon University, January 2004.
- [12] M. B. Dias, B. Browning, M. M. Veloso, and A. Stentz. Dynamic heterogenous robot teams engaged in adversarial tasks. Technical Report CMU-RI-TR-05-14, Robotics Institute, Carnegie Mellon University, 2005.
- [13] M. B. Dias, B. Ghanem, and A. Stentz. Improving cost estimation in market-based coordination of a distributed sensing task. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005.

- [14] M. B. Dias, D. Goldberg, and A. Stentz. Market-based multirobot coordination for complex space applications. In *the 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, 2003.
- [15] M. B. Dias and A. Stentz. A free market architecture for distributed control of a multirobot system. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*, 2000.
- [16] M. B. Dias and A. Stentz. Opportunistic optimization for market-based multirobot control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.
- [17] M. B. Dias, M. Zinck, R. Zlot, and A. Stentz. Robust multirobot coordination in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2004.
- [18] M. B. Dias, R. Zlot, M. Zinck, J. P. Gonzalez, and A. Stentz. A versatile implementation of the *TraderBots* approach to multirobot coordination. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*, 2004.
- [19] M. B. Dias, R. M. Zlot, N. Kalra, and A. T. Stentz. Market-based multirobot coordination: A survey and analysis. Technical Report CMU-RI-TR-05-13, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, April 2005.
- [20] D. J. Farber and K. C. Larson. The structure of a distributed computing system – software. In *Proceedings of the Symposium on Computer-Communications Networks and Teletraffic*, 1972.
- [21] G. N. Fredrickson, M. S. Hecht, and C. E. Kim. Approximation algorithms for some vehicle routing problems. *SIAM Journal on Computing*, 7(2), 1978.
- [22] V. Frias-Martinez and E. Sklar. A team-based co-evolutionary approach to multi agent learning. In *In Proceedings of the 2004 AAMAS Workshop on Learning and Evolution in Agent Based Systems*, 2004.
- [23] V. Frias-Martinez, E. Sklar, and S. Parsons. Exploring auction mechanisms for role assignment in teams of autonomous robots. In *Proceedings of the RoboCup Symposium*, 2004.
- [24] B. P. Gerkey and M. J. Matarić. Sold!: Auction methods for multi-robot control. *IEEE Transactions on Robotics and Automation Special Issue on Multi-Robot Systems*, 18(5), 2002.
- [25] B. P. Gerkey and M. J. Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*, 23(9), 2004.
- [26] D. Goldberg, V. Cicirello, M. B. Dias, R. Simmons, S. Smith, and A. Stentz. Market-based multi-robot planning in a distributed layered architecture. In A. Schultz, L. Parker, and F. Schneider, editors, *Multi-Robot Systems: From Swarms to Intelligent Automata: Proceedings from the 2003 International Workshop on Multi-Robot Systems*, volume 2. Kluwer Academic Publishers, 2003.

- [27] B. L. Golden and A. A. Assad, editors. *Vehicle Routing: Methods and Studies*, volume 16 of *Studies in Management Science and Systems*. Elsevier Science Publishers, Amsterdam, 1988.
- [28] M. Golfarelli, D. Maio, and S. Rizzi. A task-swap negotiation protocol based on the contract net paradigm. Technical Report 005-97, CSITE (Research Center for Informatics and Telecommunication Systems), University of Bologna, 1997.
- [29] J. Guerrero and G. Oliver. Multi-robot task allocation strategies using auction-like mechanisms. In *Sixth Congress of the Catalan Association for Artificial Intelligence (CCIA)*, 2003.
- [30] J. A. Hoogeveen. Analysis of christofides' heuristic: Some paths are more difficult than cycles. *Operations Research Letters*, 10, 1991.
- [31] L. Hunsberger and B. J. Grosz. A combinatorial auction for collaborative planning. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS)*, 2000.
- [32] M. Jia, G. Zhou, and Z. Chen. Arena – a centralized framework for multi-robot exploration, 2004. submitted to Robotics and Autonomous Systems.
- [33] E. G. Jones, B. Browning, M. B. Dias, B. Argall, M. Veloso, and A. Stentz. Dynamically formed heterogeneous robot teams performing tightly-coupled tasks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2006.
- [34] N. Kalra, D. Ferguson, and A. Stentz. Hoplites: A market-based framework for complex tight coordination in multi-robot teams. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2005.
- [35] N. Kalra, D. Ferguson, and A. T. Stentz. Constrained exploration for studies in multirobot coordination. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2006.
- [36] N. Kalra and A. Martinoli. A comparative study of market-based and threshold-based task allocation. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems (DARS)*, 2006.
- [37] B. Kalyanasundaram and K. Pruhs. Online weighted matching. *Journal of Algorithms*, 14(3), 1993.
- [38] S. Koenig, C. Tovey, and W. Halliburton. Greedy mapping of terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2001.
- [39] H. Köse, U. Tatlıdere, Çetin Meriçli, K. Kaplan, and H. L. Akin. Q-learning based market-driven multi-agent collaboration in robot soccer. In *The Turkish Symposium on Artificial Intelligence and Neural Networks*, 2004.

- [40] M. Lagoudakis, E. Markakis, D. Kempe, P. Keskinocak, A. Kleywegt, S. Koenig, C. Tovey, A. Meyerson, and S. Jain. Auction-based multi-robot routing. In *Robotics: Science and Systems*, 2005.
- [41] M. G. Lagoudakis, M. Berhault, S. Koenig, P. Keskinocak, and A. J. Kleywegt. Simple auctions with performance guarantees for multi-robot task allocation. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.
- [42] G. Laporte, Y. Nobert, and H. Mercure. The multi-depot travelling salesman problem. *Methods of Operations Research*, 40, 1981.
- [43] T. Lemaire, R. Alami, and S. Lacroix. A distributed tasks allocation scheme in multi-UAV context. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2004.
- [44] L. Lin, W. Lei, Z. Zheng, and Z. Sun. A learning market based layered multi-robot architecture. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2004.
- [45] L. Lin and Z. Zheng. Combinatorial bids based multi-robot task allocation method. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2005.
- [46] D. C. MacKenzie. Collaborative tasking of tightly constrained multi-robot missions. In *Multi-Robot Systems: From Swarms to Intelligent Automata: Proceedings of the 2003 International Workshop on Multi-Robot Systems*, volume 2. Kluwer Academic Publishers, 2003.
- [47] R. Nair, T. Ito, M. Tambe, and S. Marsella. Task allocation in the rescue simulation domain: A short note. In *RoboCup-2001: The Fifth Robot World Cup Games and Conferences*. Springer-Verlag, 2002.
- [48] E. Oliveira, J. M. Fonseca, and N. R. Jennings. Learning to be competitive in the market. In *Proceedings of the AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities*, 1999.
- [49] A. Pongpunwattana, R. Rysdyk, J. Vagners, and D. Rathbun. Market-based co-evolution planning for multiple autonomous vehicles. In *Proceedings of the 2nd AIAA Unmanned Unlimited Systems, Technologies and Operations Conference*, 2003.
- [50] G. Rabideau, T. Estlin, S. Chien, and A. Barrett. A comparison of coordinated planning methods for cooperating rovers. In *Proceedings of the AIAA 1999 Space Technology Conference*, 1999.
- [51] I. M. Rekleitis, A. P. New, and H. Choset. Distributed coverage of unknown/unstructured environments by mobile sensor networks. In *3rd International NRL Workshop on Multi-Robot Systems*, 2005.

- [52] D. J. Rosenkrantz, R. E. Stearns, and P. M. L. II. Approximate algorithms for the traveling salesman problem. In *Proceedings of the 15th Symposium on Switching and Automata Theory*, 1974.
- [53] T. Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *Proceedings of the 12th International Workshop on Distributed Artificial Intelligence*, 1993.
- [54] T. Sandholm. Contract types for satisficing task allocation: I theoretical results. In *AAAI Spring Symposium: Satisficing Models*, 1998.
- [55] T. Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135(1), 2002.
- [56] T. Sandholm and V. Lesser. Coalitions among computationally bounded agents. *Artificial Intelligence, Special Issue on Economic Principles of Multiagent Systems*, 94(1), 1997.
- [57] S. Sariel and T. Balch. Efficient bids on task allocation for multi-robot exploration. In *The 19th International FLAIRS Conference*. Florida Artificial Intelligence Research Society, 2006.
- [58] J. Schneider, D. Apfelbaum, D. Bagnell, and R. Simmons. Learning opportunity costs in multi-robot market based planners. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2005.
- [59] W. Sheng, Q. Yang, S. Ci, and N. Xi. Multi-robot area exploration with limited-range communications. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.
- [60] R. Simmons, D. Apfelbaum, W. Burgard, D. Fox, M. Moors, S. Thrun, and H. Younes. Coordination for multi-robot exploration and mapping. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2000.
- [61] R. Simmons, S. Singh, D. Hershberger, J. Ramos, and T. Smith. First results in the coordination of heterogeneous robots for large-scale assembly. In *Proceedings of the International Symposium on Experimental Robotics (ISER)*, December 2000.
- [62] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12), 1980.
- [63] A. Solanas and M. A. Garcia. Coordinated multi-robot exploration through unsupervised clustering of unknown space. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.
- [64] A. Stentz and M. B. Dias. A free market architecture for coordinating multiple robots. Technical Report CMU-RI-TR-99-42, Robotics Institute, Carnegie Mellon University, December 1999.

- [65] G. Thomas, A. M. Howard, A. B. Williams, and A. Moore-Alston. Multi-robot task allocation in lunar mission construction scenarios. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, 2005.
- [66] K. J. Tilley and D. J. Williams. Modeling of communications and control in an auction-based manufacturing control system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1992.
- [67] C. Tovey, M. G. Lagoudakis, S. Jain, and S. Koenig. The generation of bidding rules for auction-based robot coordination. In *Proceedings of the 3rd International Multi-Robot Systems Workshop, Naval Research Laboratory*, 2005.
- [68] D. Vail and M. Veloso. Multi-robot dynamic role assignment and coordination through shared potential fields. In A. Schultz, L. Parker, and F. Schneider, editors, *Multi-Robot Systems: From Swarms to Intelligent Automata: Proceedings from the 2003 International Workshop on Multi-Robot Systems*, volume 2. Kluwer Academic Publishers, 2003.
- [69] J. M. Vidal. The effects of cooperation on multiagent search in task-oriented domains. In *Proceedings of the Autonomous Agents and Multi-Agent Systems Conference*, 2002.
- [70] L. Vig and J. A. Adams. Market-based multi-robot coalition formation. In *Distributed Autonomous Robotic Systems (DARS)*, 2006.
- [71] E. Wolfstetter. Auctions: An introduction. *Journal of Economic Surveys*, 10(4), 1996.
- [72] Y. Xu, P. Scerri, K. Sycara, and M. Lewis. Comparing market and token-based coordination. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 2006.
- [73] R. Zlot and A. Stentz. Complex task allocation for multiple robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2005.
- [74] R. Zlot and A. Stentz. Market-based multirobot coordination for complex tasks. *International Journal of Robotics Research Special Issue on the 5th International Conference on Field and Service Robotics*, 25(1), January 2006.
- [75] R. Zlot, A. Stentz, M. B. Dias, and S. Thayer. Multi-robot exploration controlled by a market economy. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2002.

A Example Problems and Case Studies

In this section, we present a couple of real-world example scenarios and demonstrate how they can be addressed using market-based techniques. First, in Section A.1, we present an aggregation problem in which a team of robots must collect a set of spatially distributed items and transport them to a common location. We concretely illustrate how a market-based approach can be formulated for this domain, and we consider problem variations that highlight alternatives and extensions to the basic solution. Second, in Section A.2, we present a case study in market-based multirobot exploration, a problem that has been addressed in a variety of ways in the literature. Thus it allows us to highlight a range of approaches to a single problem, to analyze and compare some of the popular existing methods, and to suggest areas for future work.

A.1 Basic Aggregation

Consider a scenario in which we have a team R of robotic forklifts that must move all of the wooden pallets T scattered in a warehouse to a loading bay for transportation to another site. Their goal is to bring all the pallets into the loading bay as quickly as possible so that the shipment can be made on time. Assume that each forklift can carry up to P pallets at a time; once a forklift has P pallets, it must go to the loading bay to unload. Here, each pallet maps to a single task in the mission and the team's objective of collecting all the pallets as quickly as possible can be described by the global cost function C :

$$C(A) = \max_{r \in R} c_r(T_r(A)) \quad (1)$$

where A is a particular allocation of tasks to robots, $T_r(A)$ is the set of pallets allocated to r for transport in allocation A , and $c_r(X)$ is the fastest time that r can transport all pallets in X to the bay. Our goal is to find an allocation A^* that minimizes this global cost function.

One approach to task allocation is to hold sequential, single-task auctions (*e.g.* one pallet is allocated in each auction and only one auction occurs at a time). We could use the following bidding function B :

$$B(r, p_j) = c_r(T_r(A_{curr}) \cup \{p_j\}) \quad (2)$$

where A_{curr} is the current allocation and p_j is the pallet being auctioned. Essentially, the robot bids its total time for moving its currently allocated set of pallets and the pallet being auctioned. To find the optimal time, the robot would have to solve an instance of a multi-depot capacitated vehicle routing problem (MD-CVRP). In reality, this is intractable for more than a few tasks, so we might use some heuristics or limit our search space to find a feasible or approximate solution. Naturally, this is not guaranteed to result in the best possible solution. As we discussed in Section 4, Tovey *et al.* [67] demonstrate for $P = |T|$ that such a bidding rule results in better performance than using other bidding rules such as bidding the marginal increase in time.

A.1.1 Aggregation of Multiple-Robot Tasks

To extend example A.1, suppose that we have pallets of different sizes and that several forklifts might have to work together to jointly carry a pallet depending on its size. For example, small or S -type pallets might require only a single forklift, while medium-sized or M -type pallets require the efforts of two forklifts, and that large or L type pallets require the efforts of three forklifts. Then, the M and L type pallets are multiple-robot tasks while the S type pallets are single-robot tasks. Further suppose that a forklift can carry up to P of the S -type pallets at a time but only one M - or L - type pallet. Now, our objective of collecting the pallets as quickly as possible remains the same, so our global cost function C remains the same as in Equation 1. However, note that now the function c_r will often depend not just on the tasks allocated, but also on the activities of other teammates. Several types of auctions and bidding approaches are possible. We present three such approaches and hypothesize about the benefits and the drawbacks of each.

Instantaneous allocation. A first approach might be to use an instantaneous allocation as we discussed in Section 3.1.1 to simplify the problem of MR task allocation. That is, an auctioneer might hold sequential, single-task auctions and each unassigned robot bids its cost to complete that task:

$$B(r, p_j) = c_r(p_j)$$

where $c_r(p_j)$ is simply the time it would take for robot r to move pallet p_j . If the pallet being auctioned is an S pallet, the auctioneer awards the task to the robot with the smallest bid value. If the pallet being auctioned is an M or L pallet, then the auctioneer would award the task to the set of robots with the smallest collective set of bid values. The advantage to this approach is that it is very simple and will keep most robots working most of the time, thereby reducing the total time taken to move the pallets. However, without sequencing, each robot is limited to moving a single pallet at a time even though it has the capability to move up to P pallets at a time, thus creating potentially significant inefficiencies.

Mission decomposition. A second approach that would allow some sequencing might be to treat the problem as three different missions: the first mission is to move all the S -type tasks, the second is to move all the M -type tasks, and the third is to move all the L -type tasks. Thus, the first portion of the mission would be identical to Example A.1. In the second portion of the mission, the auctioneer could hold sequential single-item auctions and collect bids for each robot that indicate the earliest each robot could complete that item and the corresponding cost. It would award the item to the cheapest pair of bids.

For example, suppose that for a robot r , $T_r(A_{curr}) = (p_{M2}, p_{M1}, p_{M4})$, and this is also the order in which it will complete those tasks. Now, r is *obligated* to work on those tasks in that order because other robots have coordinated their schedule with r through the bidding. So, when a task p_{M6} is auctioned, r can only consider working on it after it completes p_{M4} , say at time t , and so submits a bid for time t with the cost

of that new schedule. The auctioneer awards p_{M6} to the pair of robots that have similar times of completion and that collectively have the least-costly completion time. Once the M -type tasks are all allocated and completed, an analogous approach is taken for the L -type tasks.

The benefit here is that we have significantly simplified the problem into three shifts and can easily harness the benefits of task sequencing. The obvious drawbacks are that some robots may be idle when there are fewer tasks than robots even though there are other types of tasks to be completed, and the solution space is still limited to scheduling of tasks of the same type.

Full sequencing. A third, more complex approach is to have the auctioneer hold sequential, single-item auctions for any type of task at any time. We might use two types of bidding depending on the type of pallet being auctioned. If the item currently up for auction is an S -type pallet, then all robots submit a single bid as in Equation 2. However, when the task is an M -type or a L -type pallet, robots would submit a set of bids which indicate not just the cost, but also the time that a robot could help move that pallet. The auctioneer would award the task to the strongest set of two or three bids, respectively. This approach of submitting multiple bids is similar to that used by MacKenzie [46] which we discussed in Section 3.1.1.

For example, suppose that for a robot r , $T_r(A_{curr}) = (p_{S2}, p_{S1}, p_{S4}, p_{S10})$ where S , M , and L indicate the type of pallets. Further suppose that this is the order in which r expects to complete the tasks. Then, when a pallet p_{L11} is auctioned, r can consider completing it after any of the tasks in its current schedule. It bids a set of pairs $\{t_k, c_r(T_r(A_{curr}) \cup p_j, t_k)\}$ where t_k is some time that r could arrive at p_{L11} to complete it and $c_r(T_r(A_{curr}) \cup p_j, t_k)$ would be the resulting cost of that schedule. The auctioneer would receive many bids for completing this task at different times and the corresponding costs. Its goal is to find the set of three bids that have the smallest total cost and that have the same or very similar arrival times. If such a set exists, p_{L11} is awarded to those robots under the agreement that they will be ready to move P_{L11} at that time. So, r 's schedule $T_r(A_{curr})$ might now be $(p_{S2}, p_{S1}, p_{S4}, p_{L11}, p_{S10})$. The next time a pallet is auctioned, r can only submit bids that will still enable it to be ready to complete p_{L11} at the time agreed upon.

The benefit here is that we have greatly enlarged our solution space and will probably find a better solution, but the drawback is the increased complexity of the approach: robots may have to perform complex operations to submit a set of bids, and the auctioneer's clearing algorithm may computationally intensive given the large number of bids per auction. Note that there are other ways in which a market-based solution can be constructed for this problem. For example, hierarchical auctions can be used to allocate both abstract tasks and the decomposed primitive tasks. The detailed solutions explore a few options for applying market-based techniques to the aggregation problem.

A.2 Market-based Exploration

Exploration is a fundamental multirobot problem, and makes for an interesting case study since there are a few different existing market-based solutions [60, 59, 75]. Ex-

ploration requires the team to acquire as much information about the environment as possible (*e.g.* by building a map) within the shortest amount of time or distance traveled³. Additionally, limiting repeated coverage demonstrates an efficient team solution. Typically, these requirements are manifested in the utility function, which tries to balance information gain (revenue) versus travel distance, time, or other cost factors. Here we describe three market-based approaches, specifically by looking at the auction mechanism and utility functions in each.

One approach described by Simmons *et al.* [60] uses a central greedy instantaneous assignment (IA) algorithm. The idea is that each robot bids on a list of goal point tasks, and each is assigned exactly one task by the auctioneer. The map representation used is an occupancy grid, in which each cell contains a probability of containing an obstacle and can be classified as free, occupied, or unknown by a simple thresholding. The goal points selected are frontier cells, which are known free cells adjacent to unknown cells. Each bid contains an estimated cost and information gain associated with each frontier point. (The reason for including both values separately will become apparent shortly.) The costs are based on the distance of the optimal path from the current robot position to the goal. The information gain is based on the number of unknown cells expected to be visible from the goal location. The utility for each task t is calculated by the auctioneer during auction clearing as: $U_i(t) = I_i(t) - d_i(t)$, where $I_i(t)$ is robot i 's expected information gain from task t , and $d_i(t)$ is the distance between the robot and the task. (Although not used in this approach, a scaling factor may be required for this type of utility function since the units of information gain and cost are different. Such a weight would specify the point at which it is no longer profitable to incur cost to gain more information.)

The auction clearing algorithm proceeds as follows. First, the goal with the maximum utility is allocated to the highest bidder. Then, the auctioneer discounts all remaining information gains according to the expected overlap around the assigned frontier region and the outstanding goals before making the next allocation. This is continued until all robots are assigned a goal or no goals remain. During execution, the robots are retasked whenever there is a map change; thus robots are not always required to reach their goals. One drawback to this approach is that it requires communication with a central agent, as well as considerable information sharing in order to ensure all robots maintain the same map. Additionally, the use of an IA allocation algorithm means that the robots behave myopically, although it is not clear if that is an undesirable trait for multirobot exploration [38].

Sheng *et al.* [59] essentially suggest several modifications and enhancements to the approach of Simmons *et al.* One difference is that they use a distributed auction protocol that does not require a central auctioneer or map. Robots in their system attempt to reach their goals, and upon task completion a robot determines the highest-utility goal from a list of frontier cells and broadcasts a single-task auction for that task. After a predetermined amount of time, if no other robot has submitted a higher bid for the task then it declares itself the winner and starts execution. Because each auction is only for a single task, a robot may have to call multiple auctions if it is continuously

³“The multirobot exploration problem” has also been used to describe the multi-depot traveling salesman problem in some publications [4, 41, 57], which is different from the problem we describe here.

outbid. Using multi-task auction may be an effective way to speed up the allocation process by reducing the number of auctions required in such cases. Utility functions are also modified to include a “nearness factor”, describing how close the robot is to other teammates: $U_i(t) = \omega_1 I_i(t) - \omega_2 D_i(t) + \omega_3 \lambda_i$. The intention is that this additional cost factor will tend to keep robots further apart to emphasize exploration breadth over depth. Based on simulation results, it appears that including the nearness factor may improve exploration time slightly. Since there is no central auctioneer, the robots try to maintain consistent map information in order to reduce repeated coverage. They do this by broadcasting their map updates upon reaching target points and use an information sharing protocol to reduce repeated data transmission during encounters between previously communication-isolated subgroups. If a robot receives a map update while its auction is still pending, it recalculates its utilities and starts a new auction.

Zlot *et al.* [75] introduce a distributed approach that uses peer-to-peer auctions, time-extended allocation (TA), and require no central agent. In this approach, each robot applies one or more of a set of goal point generation heuristics to produce a list of target points to visit. Although frontier points are possible as a goal generation heuristic, three simpler ones are used: choose a random point in an unknown region of the map; choose the center of the nearest unexplored region; and one based on spatial decomposition using quadtrees. At startup and subsequently upon completion of a task, a robot generates new goal points using one of these three algorithms and inserts them into its schedule until it reaches a predefined maximum length. For each heuristic, if a resulting goal is within a minimum distance of an existing goal point in the robot’s schedule or list of done tasks, it is thrown away. Robots periodically call single-award multi-task auctions to rid themselves of any goals that they may have generated that can be better-handled by another teammate. Bids are calculated as expected information gain minus expected weighted cost ($B_i(t) = I_i(t) - \omega c_i(t)$, where $c_i(t)$ is the marginal distance increase from inserting task t into robot i ’s schedule and ω is a scaling factor). In addition to submitting bids, participants can inform the auctioneer if they are already in possession of a similar task, in which case the auctioneer can eliminate that goal entirely. A limited amount of information sharing is also implemented in order to further reduce repeated coverage. In contrast to the system of Simmons *et al.* and Sheng *et al.*, this approach does not rely on maintaining consistent maps or having a central agent for task allocation, and thus can run in more communication-limited situations. In addition, the inclusion of task sequencing eliminates the myopic behavior of the robots in the other approaches above, although no direct comparisons have been made between the approaches to measure any impact this difference might have.

Unfortunately, since the above approaches have many differences, it is difficult to say which one is the best for a given problem instance without implementing each and running a thorough experiment. However, Table 4 summarizes several isolated features, allowing us to compare some aspects of the algorithms. Listed are the type of allocation agent, the category of task allocation, the auction type, and the utility function. The type of allocator reflects on the robustness of the system: as discussed in Sections 2 and 6 distributed allocation mechanisms can be more resilient to individual robot or communication failures. The allocation type refers to the taxonomy of Gerkey and Matarić [25] as discussed in Section 3. The fact that there are both instantaneous and time-extended solutions for multirobot exploration systems simply tells us that the

problem can be modeled in different ways—determining which choice is better would require empirical or theoretical results that are not currently available. Sections 4 and 5 tell us about the tradeoffs in solution quality and scalability of the auction types used in these approaches which are listed next in the table. As mentioned above, using a multi-item auction instead of a single-item auction in the approach of Sheng *et al.* may speed up the allocation stage with minimal added computational and communications overhead. Finally, Table 4 lists the utility functions used by the three approaches. A fieldable multirobot exploration system would likely require a carefully tuned (by hand or by learning) combination of multiple factors.

Table 4: Summary of market-based exploration approaches. Notation used for utility functions are described in the text of Appendix A.2.

Approach	Allocator	Allocation type	Auction type
Simmons <i>et al.</i> [60]	centralized	ST-SR-IA	multi-item
Sheng <i>et al.</i> [59]	distributed	ST-SR-IA	single-item
Zlot <i>et al.</i> [75]	distributed	ST-SR-TA	multi-item
Approach	Utility Function		
Simmons <i>et al.</i> [60]	$U_i(t) = I_i(t) - d_i(t)$		
Sheng <i>et al.</i> [59]	$U_i(t) = \omega_1 I_i(t) - \omega_2 D_i(t) + \omega_3 \lambda_i$		
Zlot <i>et al.</i> [75]	$U_i(t) = I_i(t) - \omega c_i(t)$		

Other systems that are not explicitly market-based but use similar approaches are also worth mentioning here as the utilized utility functions can be used in any of the above solutions. Solanas *et al.* [63] try to explicitly decompose the environment among the team using a k -means clustering algorithm periodically. Robots then add fixed penalty terms to their cost functions if a target point is outside their assigned region, or a smaller variable penalty term based on distance to the region centroid if the point is within the region. This encourages the robots to explore their region and discourages repeated coverage. Jia *et al.* [32] use a more complicated utility which is calculated as the ratio between the information gain and the cost (travel time plus observation time) multiplied by a term representative of local map topology.

The multirobot exploration problem demonstrates that there is not always one particular way to design a market-based approach to solve a problem: specific requirements of the domain and abilities of the robots may play a significant role in the design process. Learning the most effective utility function might provide a more solid grounding over using one of the several existing heuristic utility functions.

Constraint Optimization Coordination Architecture for Search and Rescue Robotics

Mary Koes, Illah Nourbakhsh, and Katia Sycara

Carnegie Mellon Robotics Institute

Pittsburgh, PA

{mberna, illah, katia}@cs.cmu.edu

Abstract—The dangerous and time sensitive nature of a disaster area makes it an ideal application for robotic exploration. Our long term goal is to enable humans, software agents, and autonomous robots to work together to save lives. Existing work in coordination for search and rescue does not address the variety of constraints that apply to the problem. This paper provides an expressive language for specifying system constraints. We also describe a coordination architecture capable of quickly finding an optimal or near optimal solution to the combined problems of task allocation, scheduling, and path planning subject to system constraints. We address a perceived lack of benchmarks for this research area by establishing a repository open to the research community which includes a set of benchmarks we designed to illustrate some of the complexities of the problem space. Finally, we evaluate various algorithms on these benchmarks.

Following a natural or man made disaster, rescue workers risk their lives searching for survivors in a race against time. The search and rescue domain has the potential to benefit greatly from robotic technology. Our vision is that robots could use a rough map of the environment, perhaps obtained from a blueprint or city map, and search areas specified by human rescue workers.

This problem not only has significant humanitarian benefits but poses a difficult research challenge. Robot teams in challenging environments such as disaster sites will necessarily be heterogeneous as cost limitations, power consumption, and size constraints require tradeoffs between mobility and capabilities. Large scale disasters need to include robots with diverse modalities (e.g. ground, air, water). Multiple robots are often required to work together on a *joint goal*. Coordination on a joint goal involves simultaneously solving two \mathcal{NP} -hard problems, task allocation and scheduling, as well as the path planning problem [1].

Since each passing minute reduces the chance of successfully rescuing victims, the quality of the solution is very important. Furthermore, the system must accommodate a wide variety of system constraints on goals, robots, and resources. A problem formulation that does not allow for these constraints is, for example, unable to handle the simple constraint, *turn off the circuit breakers before completing any other goals*.

These four aspects of the problem—heterogeneity, joint tasks, emphasis on optimality, and additional system constraints—distinguish the problem from other multirobot planning problems. Research in market-based algorithms for coordination [2] and token based coordination algorithms [3] cannot efficiently reason about joint goals.

The contributions of this paper are fourfold. In section

I, we establish the objective for the team planning problem and present a first order logic constraint language for the search and rescue domain. In section II, we present COCOA, a Constraint Optimization Coordination Architecture, which implements the team objective and constraint language while enabling us to employ state-of-the-art optimization engines like CPLEX [4] to solve the team planning problem. Since the problem is \mathcal{NP} -hard, finding an optimal solution may take too long, particularly as robots must interleave planning and execution due to the uncertainty in a disaster environment. In section III, we present a progressive algorithm that finds close to optimal solutions very quickly and elegantly degrades solution quality with available time. Our fourth contribution is a set of benchmarks designed to illustrate some of the complexities of the problem space which we make available to the research community. We discuss our conclusions and future work in section V.

I. TEAM PLANNING PROBLEM FOR SEARCH AND RESCUE

The coordination problem for search and rescue is part of the broader class of multi-agent planning problems. Within multi-agent planning, the search and rescue problem falls into the class of *team planning* problems since robots do not have independent goals but share the team's goals. The coordination itself is *tightly coupled* in that robots must frequently work together on a *joint goal* as no single robot working on the goal is capable of achieving the goal by itself.

A search and rescue problem consists of an environment, a set of robots, \mathcal{R} , a set of goals, \mathcal{G} , and a set of additional system constraints, \mathcal{C} . We assume that robots have some initial representation of the environment, such as a blueprint of a collapsed building, but that there may be considerable uncertainty in the model.

The set of robots on the team is defined as $\mathcal{R} := \{R^1, R^2, \dots, R^N\}$ where N is the number of robots on the team. The set of goals is $\mathcal{G} := \{G^1, G^2, \dots, G^M\}$, where M is the number of goals in the system. Each goal, G^m , has an associated location in the environment, amount of time it takes to accomplish the goal or *duration*, d^m , and reward, $Q^m(t)$.

Internal to this problem representation is the assumption that there are a number of relevant capabilities; capabilities might include the ability to map the environment, check for heat signatures, manipulate objects, or extinguish fires. Each robot has a binary capability vector indicating whether or not

robot n has a relevant capability and each goal has a binary requirement vector indicating which capabilities are needed.

The last problem component is the set of system constraints, \mathcal{C} . The expressiveness of the constraint language for the system constraints determines the scope of problems that can be solved. Goal, robot, and resource constraints form the building blocks for this constraint language.

A. Goal constraints

Allen's 13 temporal relationships [5] (*before*, *equal*, *meets*, *overlaps*, *during*, *starts*, *finishes* and their inverses) form the basis for temporal goal constraints. Goal constraints are defined to hold if and only if both goals are scheduled, allowing for a trivial solution of not scheduling either goal. In addition to the temporal relationships, the *do* constraint is necessary for cases where one or more goals must be accomplished. *Do* G^m forces the goal to be scheduled at some time but does not restrict when it is scheduled.

B. Robot constraints

It is frequently convenient to specify additional constraints on the actions of individual robots. Rescue workers may wish to specify which robots should work on a goal or exclude a certain set of robots from high risk goals. This can all be effected with the *participantIn* constraint which operates on a robot and a goal. The robot constraint R^n *participantIn* G^m forces robot n to participate in goal m if the goal is scheduled.

Another important constraint is that a robot should return to base when finished with other goals. We model this with the *endAt* constraint: R^n *endAt* L means that robot n must return to location L at the end of its mission.

C. Resource constraints

Consumable resources are an important part of the search and rescue domain. Therefore, resource constraints are a necessary part of the problem formulation. We do not consider renewable resources for this domain. In general, we assume that resource constraints apply to some subteam of robots, $\mathcal{R}_s \subseteq \mathcal{R}$ and that the resource consumption for each robot is independent.

- \mathcal{R}_s *limitFuelTo* F indicates that the robots in \mathcal{R}_s may collectively travel no farther than F which is the amount of fuel that they possess. By setting $\mathcal{R}_s = R^n$, we can limit the amount of fuel an individual robot consumes.
- \mathcal{R}_s *limitResourcesOf* $\langle f(R^n, G^m), \rho \rangle$ indicates that the robots in \mathcal{R}_s may collectively consume no more than ρ of a particular resource. The *limitResourcesOf* constraint applies to resources that are only consumed when a robot works on a goal. The function, $f(R^n, G^m)$, represents the amount of the resource robot, R^n , would consume working on goal, G^m . It must be defined for all goals in the problem, $G^m \in \mathcal{G}$, and for all robots in the subteam, $R^n \in \mathcal{R}_s$.

D. Expressive constraint language

In order to cover the full range of possible system constraints needed for search and rescue, we must be able to combine goal, robot, and resource constraints. We establish a constraint language that implements first order logic for the system constraints in \mathcal{C} .

DEFINITION 1.1. A **system constraint** is an element in \mathcal{C} and is denoted as c_i . A system constraint may either be a simple constraint or a complex constraint.

DEFINITION 1.2. A **simple constraint** is either a goal constraint (*before*, *equal*, *meets*, *overlaps*, *during*, *starts*, *finishes*, *do*), robot constraint (*participantIn*, *endAt*), or resource constraint (*limitFuelTo*, *limitResourcesOf*).

DEFINITION 1.3. A **complex constraint** consists of the composition of simple and complex constraints according to the operators listed below.

Before defining the logical operators and quantifiers, we first need to explain how constraints, which we are generally defined to always be true, can be terms in boolean logic. At the top level, the general definition of a constraint still holds; the outer most clause of a complex constraint expression must always be true. However, each nested constraint is conditional upon the value of the clause above it. If this clause is true, the nested constraint must be satisfied. If this clause is false, the nested constraint must be false. The exceptions to this are resource constraints and the *endAt* constraint which we define such that if the clause is false, the nested constraint is not required to be true. Logical operators and quantifiers include the logical not, conditional statement, if and only if, or, and, xor, forall, and exists.

II. COCOA

We have developed COCOA, a Constraint Optimization Coordination Architecture, to handle the coordination challenges posed by the search and rescue problem. The key idea behind COCOA is to preserve the semantics of the original problem by formulating it as a constraint optimization problem. COCOA uses a goal oriented representation (Figure 1) to generate a schedule that can be executed by a robot with some level of abstraction. The schedule is broken down into the necessary components of achieving a goal: traveling to the goal location, waiting for teammates who are assisting on the goal, and actually doing the goal. The maximum number of goals planned for any robot is the planning horizon, ω . Though all robots in the example in figure 1 have the same number of goals, this is not true in general; one or more robots may be idle for some or all of their planning horizon.

By modeling goal rewards as decreasing linearly with time, the problem can be modeled as a mixed integer linear programming problem (MILP) as described in [1].

A. Background MILP Problem formulation

There is the potential for confusion when discussing constraints in the MILP problem formulation. *System constraints* are defined in section I and refer to semantically meaningful

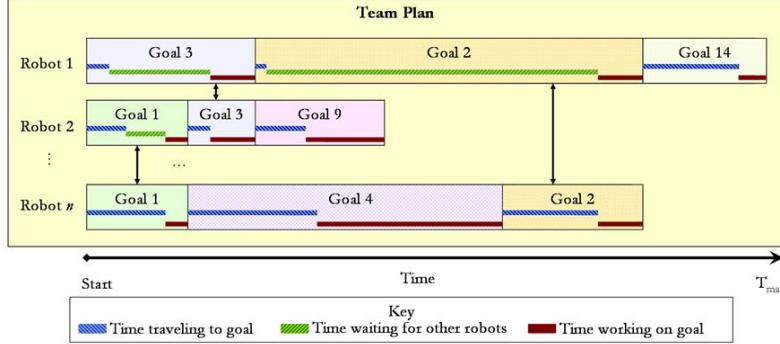


Fig. 1. COCOA uses a goal oriented representation that captures the important components of achieving a goal—traveling to the goal location, waiting for teammates who are assisting on the goal, and actually doing the goal. The maximum number of goals planned for any robot is the planning horizon, ω , ($\omega = 3$ in this example)

constraints. *Problem constraints* refer to constraints in the MILP problem formulation that implement system constraints or make the problem formulation legal.

The details of a basic problem formulation which uses both binary and real valued variables are discussed in [1]. The MILP problem formulation includes goal variables, schedule variables, and linking variables. The goal variables are of primary interest since they are used to implement the constraint language in section I.

Goals and robots are denoted with superscripts while subscripts denote a variable in the MILP problem formulation that points to the goal or robot. Therefore, G_m is the binary variable used to denote whether or not goal G^m is scheduled and $R_n G_m$ is the binary variable that indicates whether or not robot R^n works on goal G^m . $G_m.start$ is the real valued variable corresponding to the time at which goal G^m starts and $G_m.start = 0$ if G_m is 0. The MILP problem formulation also has antivariables, $G_m.start^* = G_m.start$, if $G_m = 1$, or $G_m.start^* = T_{\max}$ if $G_m = 0$.

The real valued schedule variable, $R_n O_i.travel$, is also important as it is used to implement the resource constraint *limitFuelTo*. It represents the time robot spends traveling on its way to its i th goal (blue line in figure 1, “Time traveling to goal”).

B. Modeling constraints in COCOA

In general, goal constraints are defined to hold only if the goals are scheduled, allowing a trivial solution of not scheduling either goal. This is achieved by carefully combining goal variables and antivariables in the goal constraints. With the exception of the *before* constraint, goal constraints are also defined so that scheduling one of the constrained goals forces the other constrained goal to be likewise scheduled. We assume that the duration, d^m , of goal G^m is greater than 0 and less than T_{\max} for all goals. The simple constraints can be directly translated into problem shown in table I.

While simple constraints can be modeled with variables in the problem formulation, complex constraints require the addition of new binary constraint variables, denoted C_x .

Complex constraints can be modeled by applying the following process beginning with the outer most expression and continuing in to the nested expressions.

GOAL CONSTRAINTS	
$G^x \text{before } G^y$	$G_x.start + d^x G_x < G_y.start^*$ $G_x.start^* \leq G_y.start^*$
$G^x \text{equal } G^y$	$G_x.start = G_y.start$ $G_x.d^x = G_y.d^y$
$G^x \text{meets } G^y$	$G_x.start + d^x G_x = G_y.start$
$G^x \text{overlaps } G^y$	$G_x.start < G_y.start^*$ $G_x.start + d^x G_x < G_y.start^* + d^y G_y$ $G_x.start^* + d^x > G_y.start$ $G_x = G_y$
$G^x \text{during } G^y$	$G_x.start^* > G_y.start$ $G_x.start + d^x G_x < G_y.start^* + d^y G_y$ $G_x = G_y$
$G^x \text{starts } G^y$	$G_x.start = G_y.start$ $G_x.start + d^x G_x < G_y.start^* + d^y G_y$
$G^x \text{finishes } G^y$	$G_x.start + d^x G_x = G_y.start + d^y G_y$ $G_x.start^* > G_y.start$ $G_x = G_y$
$\text{do } G^x$	$G_x = 1$

ROBOT CONSTRAINTS	
$R^n \text{ participantIn } G^m$	$R_n G_m = G_m$
$R^n \text{ endAt } L$	We create a new goal G_{end_L} $R_n G_{end_i} = 1$

RESOURCE CONSTRAINTS	
$\mathcal{R}_s \text{ limitFuelTo } F$	$\sum_{n \text{ s.t. } R^n \in \mathcal{R}_s} \sum_{i < \omega} R_n O_i.travel \leq F$
$\mathcal{R}_s \text{ limitResourcesOf } \langle f(R^n, G^m), \rho \rangle$	$\sum_{R^n \in \mathcal{R}_s} \sum_{G^m \in \mathcal{G}} f(R^n, G^m) R_n G_m \leq \rho$

TABLE I
COCOA IMPLEMENTATION OF SIMPLE CONSTRAINTS

- If the constraint is a complex constraint introduce new binary constraint variables, denoted C_x , and constraints according to the rules for each complex expression:

- Logical not: $\neg c_1: C_{not} = 1 - C_1$
- Conditional statement: $c_1 \rightarrow c_2: C_{if} \leq C_2 - C_1 + 1; C_{if} \geq \frac{C_2 - C_1 + 1}{2}$
- IF AND ONLY IF: $c_1 \leftrightarrow c_2: C_{iff} \leq C_2 - C_1 + 1; C_{iff} \geq \frac{C_2 - C_1 + 1}{2}; C_{iff} \leq C_1 - C_2 + 1; C_{iff} \geq \frac{C_1 - C_2 + 1}{2}$
- OR expression: $c_1 \vee c_2: C_{or} \leq C_1 + C_2; C_{or} \geq C_1; C_{or} \geq C_2$
- AND expression: $c_1 \wedge c_2: C_{and} \geq \frac{C_1 + C_2 - 1}{2}; C_{and} \leq C_1; C_{and} \leq C_2$
- XOR expression: $c_1 \oplus c_2: C_{xor} \geq C_1 + C_2; C_{xor} \geq C_1; C_{xor} \geq C_2$

- FORALL quantifier: This quantifier operates with robots or goals as variables.

We create a binary constraint variable, C_{\forall_i} , for each system constraint, \mathbf{c}_{R^n} or \mathbf{c}_{G^m} . These system constraints must be formulated using this process like any other complex constraint. Additionally, we have the following problem constraint where n is

$$\text{the number of robots or goals: } C_{\forall} \geq \frac{1 - n + \sum_i C_{\forall_i}}{n}$$

- EXISTS quantifier: This quantifier also operates with robots as variables.

For each system constraint, \mathbf{c}_{R^n} or \mathbf{c}_{G^m} , we create a binary constraint variable, C_{\exists_i} , and require $C_{\exists} \geq C_{\exists_i}$, so that the *exists* constraint evaluates as true if any of the individual system constraints are true. Additionally, we add the following problem constraint so that the *exists* constraint does not evaluate to true unless one or more of the individual constraints is true: $C_{\exists} \leq \sum_i C_{\exists_i}$.

- 2) For each constraint clause, \mathbf{c}_i , continue to apply the rules for modeling complex expressions.
- 3) Simple constraints that are nested inside a complex constraint cannot be directly incorporated into the MILP as non-nested simple constraints but must be formulated as dependent upon the constraint variable, C_x , of the expression in which they are nested. Algorithm 1 describes how to generate the problem constraints so that $C_x \rightarrow \mathbf{c}_i$. Note that LHS and RHS refer to the Left (or Right) Hand Side of the inequality.

Algorithm 1 Generate problem constraints for $C_x \rightarrow \mathbf{c}_i$

Use the constraints from table I that correspond to \mathbf{c}_i . Turn all equalities, $x = y$, into inequalities: $x \leq y$ and $x \geq y$. Rearrange each inequality so that the RHS = 0

```

if LHS ≤ 0 or LHS < 0 then
    RHS = (1 - Cx)Tmax
else
    RHS = (Cx - 1)Tmax
end if

```

- 4) Force $\neg C_x \rightarrow \neg \mathbf{c}_i$ by applying DeMorgan's law ($\neg(A \wedge B) = \neg A \vee \neg B$) and following algorithm 2.
- 5) Set the top level constraint variable $C_x = 1$.

We can quickly verify that the process for establishing $C_x \rightarrow \mathbf{c}_i$ is valid: if $C_x = 0$, we have the left hand side of the expression, LHS $\{>, \geq\} - T_{max}$ or LHS $\{<, \leq\} T_{max}$. Since all times are bounded by T_{max} , this is a trivial constraint. On the other hand, if $C_x = 1$, the right hand side is 0 which is simply a restatement of the original constraint. The check for $\neg C_x \rightarrow \neg \mathbf{c}_i$ is similar, though slightly longer.

C. Solving the MILP with CPLEX

By modeling the problem as an MILP as described in section I, we can leverage a vast amount of work in solving these problems including the commercially available solver, CPLEX [4]. The standard approach for solving an MILP is to relax the constraint that certain variables be integer and

Algorithm 2 Generate problem constraints for $\neg C_x \rightarrow \neg \mathbf{c}_i$

Use the constraints from table I that correspond to \mathbf{c}_i .

Turn all equalities, $x = y$, into inequalities: $x \leq y$ and $x \geq y$.

Rearrange each inequality so that the RHS = 0

for all resulting inequality problem constraints **do**

Create a new binary variable C_{x_i} .

if LHS ≤ 0 or LHS < 0 **then**

RHS = $(1 - C_{x_i})T_{max}$

else

RHS = $(C_{x_i} - 1)T_{max}$

end if

end for

Add problem constraint: $\sum_i C_{x_i} + \sum_{G_m \in \mathbf{c}_i} (1 - G_m) \geq (1 - C_x)$

solve the resulting linear program (LP) using LP algorithms such as the simplex or dual simplex algorithm. This noninteger solution is known as the *relaxed solution* and provides an upper bound on system performance. In the case that the relaxed solution is integral, this optimal solution is returned.

Since the problem is \mathcal{NP} -hard, generally the system must employ the branch and bound algorithm to search for the optimal integer solution. Without additional information, the search for the integer solution begins near the relaxed solution. Depending on the structure of the search space, finding a feasible solution can take a long time (hours to find a feasible solution to a large search and rescue problem and days or weeks to find the optimal solution). This is unacceptable for the search and rescue domain since planning delays the onset of execution.

III. REAL TIME ALGORITHMS

Particularly for problems with few or no constraints, it is possible to use heuristics to find solutions in polynomial time. In fact, the search and rescue problem has elements, specifically rewards that decrease over time, that suggest that greedy heuristics would perform reasonably well. Conveniently, we can combine heuristics and MILP solution techniques resulting in an algorithm superior to either individual approach.

As described in the previous section, solving an MILP involves branch and bound search. By default, this search begins around the relaxed solution. It is possible to generate an initial solution using a heuristic and use this as the starting point for the search. This enables the system to find a feasible solution quickly and then improve the solution with available time. Furthermore, the bounds in the branch and bound algorithm provide error bounds on a given solution so that it is possible to stop when the solution is within some bounds of the optimal or estimate distance of an arbitrary solution from optimal. We exploit this symbiosis between heuristic and optimal solver in our anytime algorithm (Algorithm 3).

A. Anytime Algorithm

The key idea in the anytime algorithm is to make the interaction between the optimization algorithm (CPLEX) and heuristic (any number of heuristics can be used—we describe two below) an iterative process by starting with a small subproblem and expanding this until it includes the entire

Algorithm 3 Anytime Scheduling using MILP Solver

```
1: oldSoln = {}
2: for  $\omega = 1$  to # Goals do
3:   milp  $\leftarrow$  CreateMILP( $\omega$ )
4:   start  $\leftarrow$  HeuristicScheduler(oldSoln,  $\omega$ )
5:   optimalSolnForPH  $\leftarrow$  Optimize(milp, start)
6:   oldSoln = optimalSolnForPH
7: end for
8: return optimalSolnForPH
```

problem. At each iteration, a heuristic is used to generate a solution for the subproblem. This heuristic solution is used as a starting point for search in the optimization step. Since the resulting optimized solution to this subproblem is at least as good as the heuristic solution to the same subproblem, the heuristic uses this optimized solution as its starting point and only finds a solution for the expanded portion of the problem.

B. Heuristics

Since goal rewards decrease over time, it is reasonable to expect greedy heuristics to perform well. Two greedy heuristics arise naturally from the problem variables—robots and goals. In the first heuristic, which we call the *myopic* heuristic, robots recursively schedule goals over an increasing planning horizon. The heuristic attempts to find the best solution for a planning horizon of 1. We use our MILP problem formulation and CPLEX to find this solution as shown in algorithm 4. Empirically, this is very fast—well under a second—even for large problems. This is because problem size grows exponentially with the length of the planning horizon so this subproblem is much smaller than the original problem.

The second heuristic, or *greedy goal* heuristic, is inspired by market based task allocation algorithms which have been shown to work well in a variety of domains [2]. Since the goals are known in advance, they are sorted and auctioned off in decreasing order of reward. For each required capability of each goal, robots bid their costs to provide that capability based on their current schedules. The lowest bidder is assigned to that goal capability. The greedy goal heuristic is less sophisticated than many market based systems as it does not allow robots to reauction their goals.

C. Myopic heuristic limitations

Each of these heuristics has its limitations and can lead to inefficient behavior if not combined with an optimization algorithm. The myopic heuristic performs poorly if it is best but not required for one robot to assume more goals than its teammates. For example, if there are two goals with the same requirements in the same room and two identical robots, one already in the room and one in a different building, the myopic heuristic will assign one goal to each robot although the optimal solution is clearly to assign both goals to the close robot.

D. Greedy goal heuristic limitations

The greedy goal heuristic but may perform poorly if the problem requires careful resource management. Consider a

Algorithm 4 Myopic heuristic

```
1:  $\omega = 1$ ; unsatGoals = Goals; currSchedule = {}
2: while ( $\omega \leq$  # Goals)  $\&\&$  !isEmpty(unsatGoals) do
3:   prob  $\leftarrow$  CreateMILP(1, unsatGoals)
4:   partialSoln  $\leftarrow$  Optimize(prob)
5:   currSchedule  $\leftarrow$  Schedule(partialSoln)
6:   unsatGoals  $\leftarrow$  unsatGoals—satGoals(currSchedule)
7:    $\omega = \omega + 1$ 
8: end while
9: return currSchedule
```

disaster site with two flooded rooms and a dry room and a team of two robots. However, only robot 1 is waterproof and able to explore the flooded rooms. If robot 1 starts slightly closer to the dry room and the goal of exploring the dry room is allocated first, the greedy goal heuristic will assign all three goals to robot 1 although it would be more efficient to assign the dry room to robot 2 to explore.

IV. BENCHMARKS AND RESULTS

The lack of benchmarks for heterogeneous multirobot coordination with joint tasks is an obstacle to rigorously comparing algorithms and heuristics. We have developed a collection of benchmarks designed to illustrate some of the challenges of a complex domain such as search and rescue including the limitations of the heuristics as described in the previous section. As a service to the research community, we have set up a repository to allow researchers to access our benchmarks and contribute their own: <http://www.cs.cmu.edu/~mberna/research/>.

Although the system constraints from section I add an important degree of expressiveness to the system, a baseline analysis without constraints must first be performed to better understand the problem space. Therefore, although we are able to automatically generate benchmarks with constraints which are included in the repository, this analysis limits itself to problems without additional system constraints. This also allows us to compare algorithms and heuristics that lack the expressiveness to represent and reason about these constraints.

A. Benchmarks

We implemented a program to automatically generate random environments according to certain specification which we used to generate 5 environments with 3 and 15 robots and 5 and 15 goals for a total of 20 problems in each of the following classes of benchmarks. All the benchmarks have three relevant capabilities but the profiles of robot capabilities and goal requirements distinguish the various benchmarks. The benchmark classes are described in table II. In order to limit the variations between benchmarks, all environments are randomly generated 10x10 grids with randomly assigned 4 point connectivity between nodes.

B. Results

We compared five solution techniques on these benchmarks: the myopic heuristic (Algorithm 4), the greedy goal heuristic described in section III, the anytime algorithm (Algorithm 3) with the myopic heuristic as the heuristic solver on line 4

Benchmark	Robot Capabilities	Goal Requirements	Goal Location
Homogeneous	All	All	Random
Tight	Single	All	Random
Easy Clustered	All	All	Clustered
Difficult Clustered	Single	All	Clustered
Precious Resources	Super and weak robots	Easy and hard goals	Random
Random	Random	Random	Random

TABLE II

BENCHMARK CHARACTERISTICS (SUPER ROBOTS CAN DO ANY GOAL

WHILE WEAK ROBOTS ARE ONLY ABLE TO DO EASY GOALS)

of Algorithm 3, the anytime algorithm with the greedy goal heuristic as the heuristic solver, and the anytime algorithm with the “best” heuristic in which the heuristic solver compares the myopic and greedy heuristics and returns the solution with the highest utility. The optimization step (line 5 of Algorithm 3) was limited to 90 seconds. We compared the results for three different levels of time pressure, $T_{max} = 100, 1000$, and 2000. The results for $T_{max} = 100$ are shown in figure 2.

C. Analysis

In general, we observed that the greedy goal heuristic performed well on the homogeneous and clustered benchmarks. This matches what we know about these benchmarks and the workings of the greedy goal heuristic. The homogeneous and easy clustered benchmarks have no joint goals so the problem reduces to the optimal assignment problem on which greedy market task allocation has been demonstrated to perform well [2]. The greedy goal heuristic performs the worst on the precious resource benchmarks which matches our analysis of the limitations of the greedy goal heuristic in section III-D.

The myopic heuristic performed at least as well as the greedy goal heuristic on all benchmarks except the easy clustered benchmarks on which it performed slightly worse. This matches our analysis of the limitations of the myopic heuristic (section III-C). The performance is only slightly worse because the tasks are clustered together so that even if the tasks are allocated suboptimally for a planning horizon of 1, this is outweighed by the goals accomplished over the remainder of the planning horizon. In this sense, the clustered goal benchmarks were less successful than we had hoped in illustrating the limitations of the myopic heuristic.

The anytime algorithm was able to significantly improve performance on a number of benchmarks with a maximum improvement of over 50%. This indicates that the heuristics benefit significantly from the relatively small amount of time spent in optimization. Without using the heuristic starting solution, CPLEX is often unable to find a feasible solution in 90 seconds which illustrates the benefits of the heuristic starting solution to the MILP solver.

We found that the results from the other values of T_{max} are nearly identical to figure 2 (and so are not included here) except in scale. The utility gains when $T_{max} = 2000$ are half the utility gains for $T_{max} = 1000$. Since utility is defined as a linear function of T_{max} , we would expect a linear relationship between T_{max} and utility gain. However, the utility gains when $T_{max} = 100$ are 15 times the utility gains of $T_{max} = 1000$.

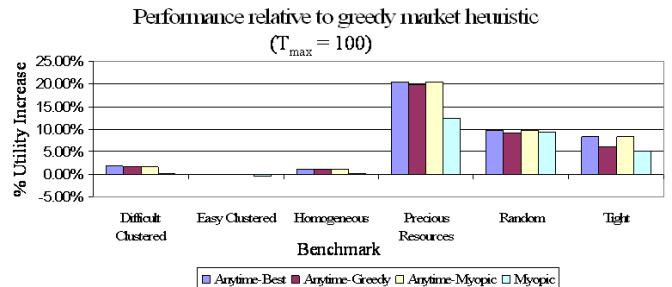


Fig. 2. Benchmark results using greedy goal heuristic as baseline averaged over 3 and 15 robots and 5 and 15 goals.

The reason for this is that as T_{max} decreases, suboptimal solutions are not able to accomplish as many goals which magnifies differences in solution quality. This illustrates the importance of optimality in the search and rescue domain where there is often significant time pressure.

V. CONCLUSIONS AND FUTURE WORK

In this paper we presented a language for modeling constraints in first order logic, an architecture based on well known mixed integer linear programming work capable of finding optimal team plans subject to these constraints, and an anytime algorithm and various heuristics to solve these problems quickly. We evaluated these solution techniques on a set of multirobot team planning benchmarks. The benchmarks analyzed here are only a beginning. We hope other researchers will contribute test cases and will continue to add benchmarks to the repository ourselves including test cases with semantically meaningful constraints. Finding optimal or near optimal team plans in the presence of system constraints is an open problem that requires the development of new algorithms. Although this paper was motivated by the search and rescue domain, the general problem formulation and solution techniques discussed are applicable to a wide range of domains such as integrated manufacturing, reconnaissance, or space exploration.

ACKNOWLEDGEMENTS

This work is supported by NSF Award IIS-0205526. The authors would also like to thank the ILOG corporation for their generous support.

REFERENCES

- [1] M. Koes, I. R. Nourbakhsh, and K. P. Sycara, “Heterogeneous multirobot coordination with spatial and temporal constraints,” in *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*. AAAI Press / AAAI Press / The MIT Press, 2005, pp. 1292–1297.
- [2] M. B. Dias, R. M. Zlot, N. Kalra, and A. T. Stentz, “Market-based multirobot coordination: A survey and analysis,” Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-05-13, April 2005.
- [3] Y. Xu, P. Scerri, B. Yu, S. Okamoto, M. Lewis, and K. Sycara, “An integrated token-based algorithm for scalable coordination,” in *AAMAS ’05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM Press, 2005, pp. 407–414.
- [4] *ILOG CPLEX 9.0 User’s Manual*, ILOG, 2004.
- [5] J. F. Allen, “Maintaining knowledge about temporal intervals.” *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983.