Reinforcement Learning

Summer of Science

End-Term Report

Saksham Rathi
First-Year Computer Science Undergraduate
22B1003

Under the mentorship of **Omar Kashmar**

Contents

Certificate of Completion

This is to certify that

Saksham Rathi

has successfully completed the report on

Reinforcement Learning

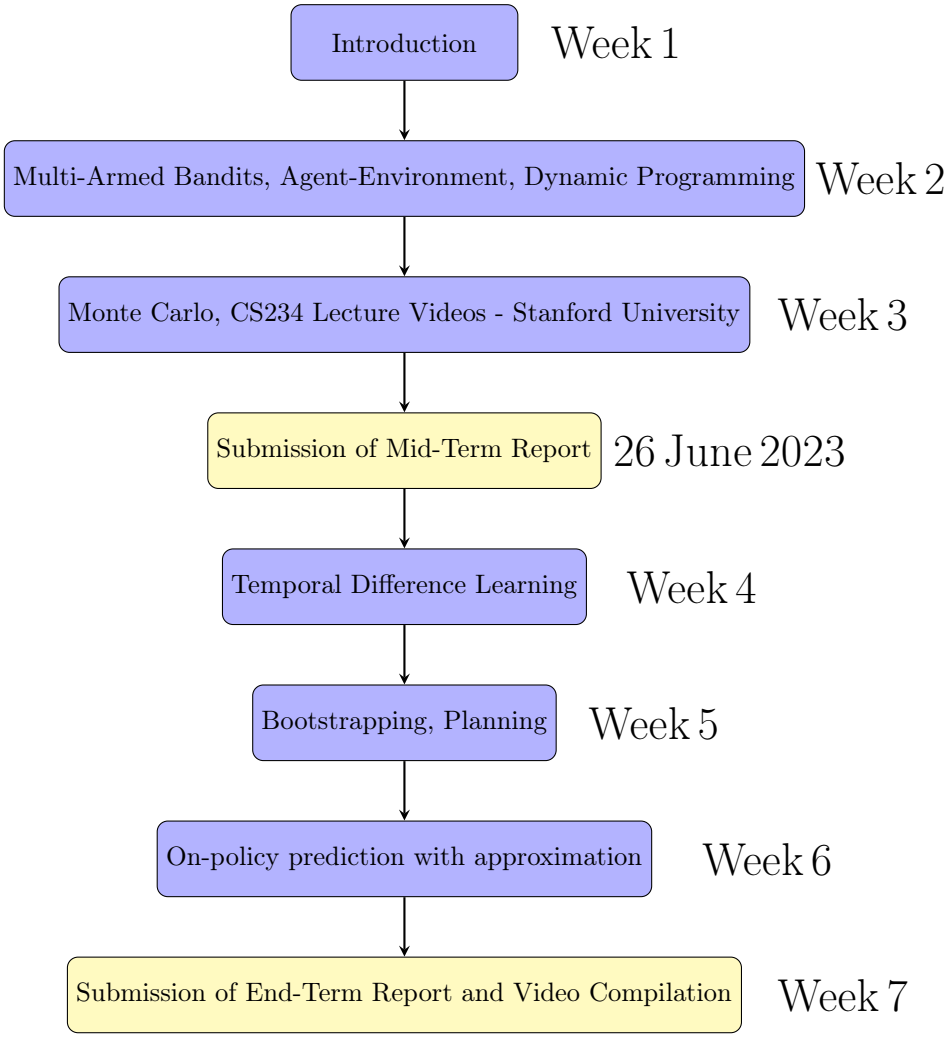
This report on Reinforcement Learning is presented in partial fulfillment of the requirements for the completion of the Summer of Science conducted by Maths and Physics Club, IIT Bombay. The report demonstrates a good understanding of the concepts and principles of reinforcement learning, and showcases the student's ability to apply them effectively.

Date: 27 July 2023

Signature:

[Saksham Rathi]

Plan of Action



A Brief Overview of the Report

The ultimate goal of this report is to delve into the realm of Machine Learning and explore the utilization of diverse algorithms to enhance the precision and effectiveness of autonomous learning systems. Through this investigation, we aim to acquire valuable insights into the development of Machine Learning methodologies. Various references have been used to prepare the report. Much of the text and pseudo-code has been taken from [2]. The overall flow of the material has been adopted from [3] and [4].

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.

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 make, but instead must discover which actions yield the most reward by trying them.

1.1 Huge Increase in Interest

Over the past few years, there has been a huge increase in the research papers published in the field of Reinforcement Learning.

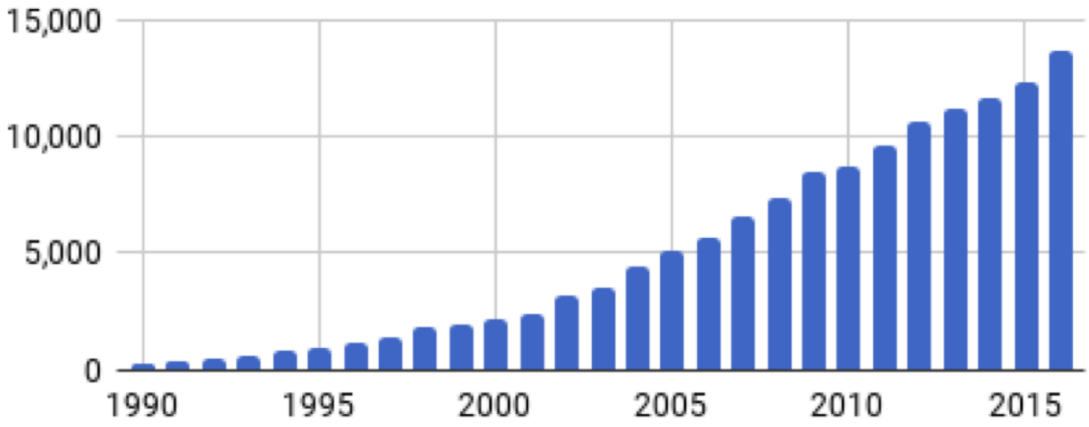


Figure 1: Research Papers Published over the years [1]

1.2 Key Features

Reinforcement learning is different from supervised learning which is learning from a training set of labeled examples provided by a knowledgeable external supervisor. 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. 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.

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.

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.

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.

1.3 Examples

Some of the possible applications where Reinforcement Learning can be applied are:

- A master chess player makes a move. The choice is informed both by planning—anticipating possible replies and counter-replies and by immediate, intuitive judgements of the desirability of particular positions and moves.
- 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.

1.4 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.

1. **Policy**: A policy defines the learning agent's way of behaving at a given time. It is a mapping from perceived states of the environment to actions to be taken when in those states.
2. **Reward**: 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.

	AI Planning	SL	UL	RL	IL
Optimization	Y			Y	Y
Learns from experience		Y	Y	Y	Y
Generalization	Y	Y	Y	Y	Y
Delayed Consequences	Y			Y	Y
Exploration				Y	

Table 1: Different Machine Learning Paradigms

3. **Value Function**: A value function specifies what is good in the long run. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.
   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.

4. **Model**: 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. 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.

1.5 Different from Other Machine Learning Paradigms

Reinforcement Learning is different from other Machine Learning types. This will become clear based on the following factors as is shown in the table 1.5.

Let us go through the terms used in the table briefly:

- **Optimization:** Goal is to find an optimal way to make decisions.
- **Delayed Consequences:** Decisions can impact things later.
- **Exploration:** Learning about the things by making decisions.
- **SL:** Supervised Learning
- **UL:** Unsupervised Learning
- **IL:** Imitation learning typically assumes input demonstrations of good policies.
- **AI planning:** It assumes that we have a model of how decisions impact environment.

2 Multi-armed Bandits

In this problem, you are faced repeatedly with a choice among k different options, or actions. After each choice, you will receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your goal is to maximize the expected total reward over some time period.

We denote the action selected on time step t as A_t and the corresponding reward as R_t. The value then of an arbitary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) = E[R_t|A_t = a]$$

We denote the estimated value of action a at time step t as $Q_t(a)$.

One natural way to estimate this is by averaging the rewards actually received:

$$Q_t(a) = \frac{\texttt{sum of rewards when a taken prior to t}}{\texttt{number of times a taken prior to t}} = \frac{\sum_{i=1}^{t-1} R_i.1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}} \tag{1}$$

where $1_{predicate}$ denotes the random variable that is 1 if predicate is true and 0 if it is not.

The simplest action selection rule is to select one of the actions with the highest estimated value. We write this greedy action selection method as:

$$A_t = argmax_a Q_t(a) \tag{2}$$

where $argmax_a$ denotes the action a for which the expression that follows is maximized.

Another 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 selection rule ϵ-greedy methods.

2.1 Incremental Implementation

The general form of incremental implementation is:

```
NewEstimate <- OldEstimate + StepSize [ Target - OldEstimate ]
```

A simple bandit algorithm

Initialize, for $a = 1$ to k:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

Loop forever:
$$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$
$R \leftarrow bandit(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\big[R - Q(A)\big]$

Figure 2: A simple bandit algorithm using incremental implementation [2]

Pseudocode for a complete bandit algorithm using incrementally computed sample averages and ϵ-greedy action is shown in the figure 2.

2.2 Upper-Confidence-Bound Action Selection

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 being to being maximal and the uncertainties in those estimates. One effective way of doing this is to select actions according to:

$$A_t = argmax_a[Q_t(a) + c\sqrt{\frac{lnt}{N_t(a)}}] \tag{3}$$

2.3 Associative Search

Suppose we are facing an actual slot machine that changes the colour of its display as it changes its action values. Now we can learn a policy facing that task – for instance, if red, select arm 1; if green, select arm 2. With the right policy we can usually do much better that we 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.

3 The Agent-Environment Interface

Markov Decision Processes (MDPs) are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, through those future rewards. 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. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions. This relationship is clearly demonstrated using the following figure 3:

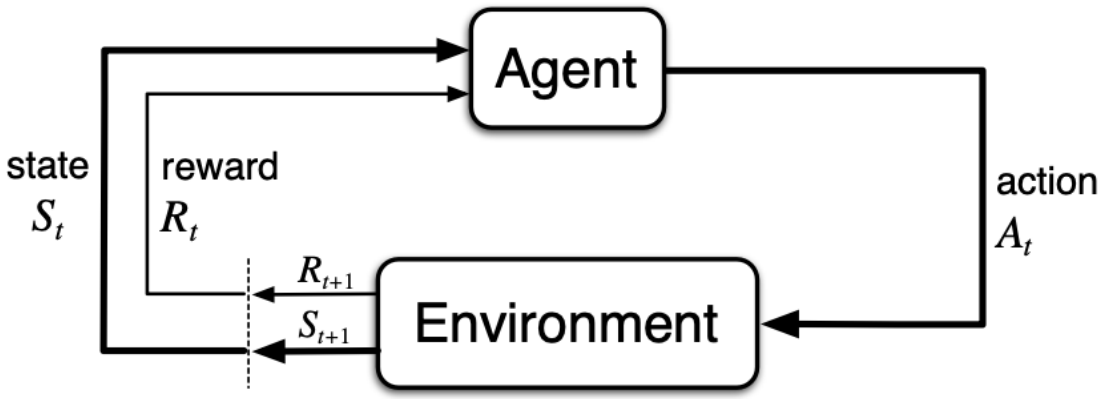


Figure 3: The agent-environment interaction in a Markov Decision Process

More specifically, the agent and environment interact at each of a sequence of discrete time steps, t = 0, 1, 2, 3, ... At each time step t, the agent receives some representation of the environment's state, $S_t \in S$, and on that basis selects an action, $A_t \in A(s)$.

For particular values of random variables, $s' \in S$ and $r \in R$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{T-1} = S, A_{t-1} = a\} \tag{4}$$

The function p defines the dynamics of the MDP. 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 can also compute the expected rewards for state-action pairs as a two-argument function $r : S \times A \to R$:

$$r(s, a) = E[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a) \tag{5}$$

and the expected rewards for state-action-next-state triples as a three-argument function $r : S \times A \times S \to R$,

$$r(s, a, s') = E[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in R} r \frac{p(s', r|s, a)}{p(s'|s, a)} \tag{6}$$

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. The agent-environment boundary represents the limit of the agent's absolute control, not of its knowledge.

3.1 Return and Episodes

In general, we seek to maximize the expected return, where the return denotes G_t, is defined as some specific function of reward sequence. In the simplest case the return is the sum of the rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{7}$$

where T is the 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. 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.

On the other hand, in many cases the agent–environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. We call these continuing tasks.

According to the discounting approach, the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{8}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate.

If $\gamma = 0$, the agent is myopic in being concerned only with maximizing immediate rewards.

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{9}$$

3.2 Policies and Value Functions

A policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$.

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.

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], forall s \in S, \tag{10}$$

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 into action a, and thereafter following policy π.

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{11}$$

We call q_π the action-value function for policy π.

$$v_\pi(s) = \sum_a \pi(a, s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{12}$$

for all $s \in S$. Equation(12) is the Bellman equation for v_π.

3.3 Optimal Policies and Optimal Value Functions

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 $s \in S$. 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, denotes v_*, and defined as:

$$v_*(s) = max_\pi v_\pi(s)$$

for all $s \in S$. Optimal policies also share the same optimal action-value function.

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). Bellman optimality equations:

$$v_*(s) = max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s)] \tag{13}$$

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma max_{a'} q_*(s', a')] \tag{14}$$

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)

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 13 as an update rule:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{15}$$

Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called iterative policy evaluation. A complete in-place version of iterative policy evaluation is shown in pseudocode in the box below4.1. The pseudocode tests the quantity $max_{s\in S}|v_{k+1}(s) - v_k(s)|$ 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 $V(terminal)$ to 0

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

4.2 Policy Improvement

Let π and π' be any pair of deterministic policies such that, for all $s \in S$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

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 $s \in S$:

$$v_{\pi'}(s) \geq v_\pi(s)$$

.

In other words, to consider the new greedy policy π', given by:

$$\pi'(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

where $argmax_a$ denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). 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. In particular, the policy improvement theorem carries through as stated for the stochastic case.

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. 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 below4.3.

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 $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow false$
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

4.4 Value Iteration

Value Iteration is an algorithm that can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$v_{k+1}(s) = max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{16}$$

The box below4.4 shows a complete algorithm with this kind of termination condition. Value

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 $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

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.

4.5 Asynchronous Dynamic Programming

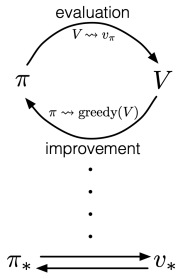
A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive.

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.

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.

4.6 Generalized Policy Iteration

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.



5 Monte Carlo Methods

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).

Monte Carlo methods sample and average returns for each state–action pair much like the bandit methods 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.

5.1 Monte Carlo Prediction

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. First-visit MC is shown in procedural form in the box5.1. 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.

5.2 Monte Carlo Estimation of Action Values

Without a model, 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

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: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow$ average$(Returns(S_t))$

goals for Monte Carlo methods is to estimate q_*.

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. 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.

5.3 Monte Carlo Control

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.

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 box5.3.

5.4 Monte Carlo Control without 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.

In on-policy control methods the policy is generally soft, meaning that $\pi(a|s) > 0$ for all $s \in S$ and all $a \in A(s)$, but gradually shifted closer and closer to a deterministic optimal policy. In our on-policy method we will move it only to an ϵ-greedy policy.5.4

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

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: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
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 $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

5.5 Off-policy Prediction via Importance Sampling

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.

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:

$$Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T | S_t, At : T-1 \sim \pi\} = \prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)$$

Thus, the relative probability of the trajectory under the target and behavior policies (the importance sampling ratio) is:

$$p_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k) p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{17}$$

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.

To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

$$V(s) = \frac{\sum_{t \in J(s)} \rho_{t:T-1} G_t}{|J(s)|} \tag{18}$$

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:

$$V(s) = \frac{\sum_{t \in J(s)} \rho_{t:T-1} G_t}{\sum_{t \in J(s)} \rho_{t:T-1}} \tag{19}$$

or zero if the denominator is zero.

5.6 Incremental Implementation

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. We wish to form the estimate

$$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}$$

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:

$$V_{n+1} = V_n + \frac{W_n}{C_n}[G_n - V_n]$$

and

$$C_{n+1} = C_n + W_{n+1}$$

The box 5.6 contains a complete episode-by-episode incremental algorithm for Monte Carlo policy evaluation.

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of π
    Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, 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

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 5.7 shows the pseudocode of this:

6 Temporal-Difference Learning

Temporal-Difference Learning (TD) is a central and novel idea to Reinforcement Learning. It is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo

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)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \arg\max_a Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

methods, TD methods can learn directly from raw experience without a model of the environment's dynamics.

6.1 TD Prediction

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:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{20}$$

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. The box 6.1below specifies TD(0) completely in procedural form.

TD error:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{21}$$

Monte Carlo error can be written as a sum of TD errors:

$$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k \tag{22}$$

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 $V(terminal) = 0$

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'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

6.2 Advantages of TD Prediction Methods

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 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.

6.3 Optimality of TD(0)

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 are computed for every time step t at which a non-terminal 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. This is called batch updating because updates are made only after processing each complete batch of training data.

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. 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.

6.4 Sarsa: On-policy TD Control

The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{23}$$

This update is done after every transition from a non-terminal 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 general form of the Sarsa control algorithm is given in the box6.4.

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 $Q(terminal, \cdot) = 0$

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)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

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, defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{24}$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. The Q-learning algorithm is shown below in procedural form.6.5

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:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma E_\pi[Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)] \tag{25}$$

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 $Q(terminal, \cdot) = 0$

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'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

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.

6.7 Maximization Bias

All the control algorithms discussed so far involve maximization in the construction of their target policies. Considering 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. This is called the maximization bias.

6.8 Double Learning

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). We could then use one estimate, say Q_1, to determine the maximizing action $A^* = argmax_a Q_1(a)$ and the other, Q_2, to provide the estimate of its value, $Q_2(A^*) = Q_2(argmax_a Q_1(a))$. This estimate will then be unbiased in the sense that $E[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(argmax_a Q_2(a))$. This is the idea of double learning.

A complete algorithm for Double Q-learning is given in the box below 6.8:

7 n-step Bootstrapping

n-Step TD methods generalize both Monte Carlo and the one-step temporal-difference 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.

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 $Q(terminal, \cdot) = 0$

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 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

n-Step methods enable bootstrapping to occur over multiple steps, freeing us from the tyranny of the single time step.

7.1 n-step TD Prediction

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. Figure 4 shows the backup diagrams of the spectrum of n-step updates with the one-step TD update on the left and up-until termination Monte Carlo update on the right.

Methods in which the temporal difference extends over n steps are called n-step TD methods. The target for an arbitrary n-step update is the n-step return:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n_1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{26}$$

for all n,t such that $n \geq 1$ and $0 \leq t < T - n$. 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 ($G_{t:t+n} = Gtift + n \geq T$).

The natural state-value learning algorithm for using n-step returns is thus:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)] \tag{27}$$

Complete pseudocode is given in the box 7.1.

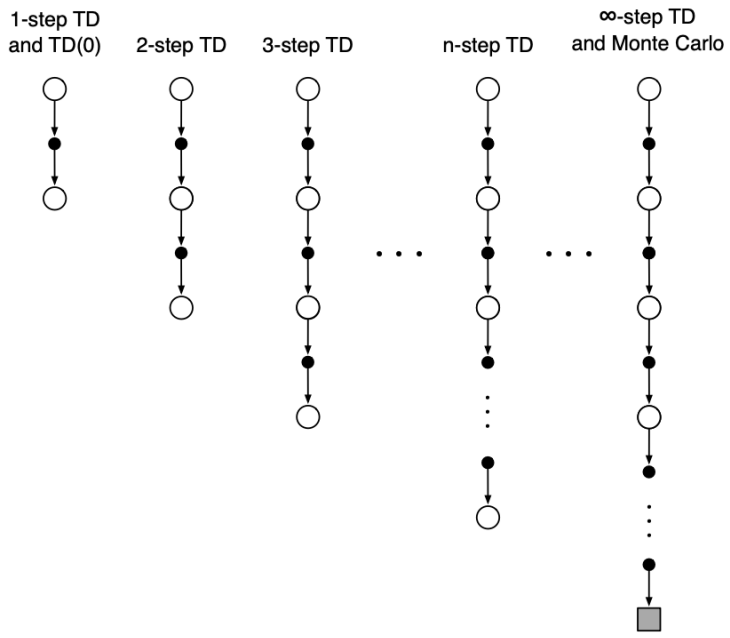


Figure 4: The backup diagrams of n-step methods. These methods form a spectrum ranging from one-step TD methods to Monte Carlo methods.

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}

$$\max_s |E_\pi[G_{t:t+n}|S_t = s] - v_\pi(s)| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)| \tag{28}$$

This is called the error reduction property of n-step returns.

7.2 n-step Sarsa

We redefine n-step returns (update targets) in terms of estimated action values:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1(S_{t+n}, A_{t+n}} \tag{29}$$

The natural algorithm is then: 7.2

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{30}$$

7.3 n-step Off-policy Learning

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1}[G_{t:t+n} - V_{t+n-1}(S_t)] \tag{31}$$

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 $t = 0, 1, 2, \ldots$:
  | 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 $\tau = T - 1$

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}

$$\rho_{t:h} = \prod_{k=t}^{min(h,T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{32}$$

Pseudocode for the full algorithm is shown in the box. 7.3

7.4 Per-decision Methods with Control Variates

Suppose the action at time t would never be selected by π so that p_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:

$$G_{t:h} = \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1 - \rho_t)V_{h-1}(S_t) \tag{33}$$

where $G_{h:h} = V_{h-1}(S_h)$. The second term in equation 33 is called a control variate.

7.5 The n-step Tree Backup Algorithm

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 $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | 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 $A_{t+1} \sim \pi(\cdot|S_{t+1})$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose 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 $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\left[G - Q(S_\tau, A_\tau)\right]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T - 1$

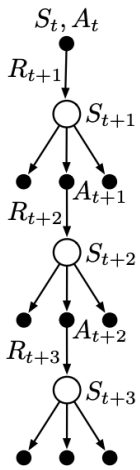
So far we have always updated the estimated value of the node at the top of the diagram toward a target combining the rewards along the way (appropriately discounted) and the estimated values of the nodes at the bottom. In the tree-backup update, the target includes all these things plus the estimated 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.

The one-step return(target) is same as that of Expected Sarsa:

$$G_{t:t+1} = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a) \tag{34}$$

The general recursive definition of the tree-backup diagram n-step return:

$$G_{t:t+n} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n} \tag{35}$$



the 3-step tree-backup update

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 π 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 $t = 0, 1, 2, \ldots$:
  | 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 $A_{t+1} \sim b(\cdot|S_{t+1})$
  | $\tau \leftarrow t-n+1$ (τ is the time whose estimate is being updated)
  | If $\tau \geq 0$:
  | $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n,T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$ $(\rho_{\tau+1:\tau+n})$
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | 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 π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T-1$

This target is then used with the usual action-value update rule from n-step Sarsa:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{36}$$

Pseudocode is shown in the box 7.5:

7.6 Unification

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.

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

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 π 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 $t = 0, 1, 2, \ldots$:
  | 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}
  | $\tau \leftarrow t+1-n$ (τ 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 $\tau + 1$:
  | $G \leftarrow R_k + \gamma \sum_{a \neq A_k} \pi(a|S_k) Q(S_k, a) + \gamma \pi(A_k|S_k) G$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha \left[G - Q(S_\tau, A_\tau) \right]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T-1$

full sampling and $\sigma = 0$ denoting a pure expectation with no sampling. This proposed new algorithm is called n-step $Q(\sigma)$.

n-Step return:

$$G_{t:h} = R_{t+1} + \gamma(\sigma_{t+!}\rho_{t+1} + (1-\sigma_{t+1})\pi(A_{t+1}|S_{t+!}))(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1}) + \gamma \bar{V}_{h-1}(S_{t+1}) \quad (37)$$

The recursion ends with $G_{h:h} = Q_{h-1}(S_h, A_h)$ if h<T, or with $G_{T-1:T} = R_T$ if h=T. A complete algorithm is given in the box 7.6:

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 $t = 0, 1, 2, \ldots$:
  | 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:
  | $\bar{V} \leftarrow \sum_a \pi(a|S_k) Q(S_k, a)$
  | $G \leftarrow R_k + \gamma\big(\sigma_k \rho_k + (1-\sigma_k)\pi(A_k|S_k)\big)\big(G - Q(S_k, A_k)\big) + \gamma \bar{V}$
  | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
  | If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt Q
  Until $\tau = T - 1$

8 Planning and Learning

Model-based methods require a model of the environment, such as dynamic programming and heuristic search. They rely on planning as their primary component. Model-free methods can be used without a model, such as Monte Carlo and temporal-difference methods. They primarily rely on learning.

8.1 Models and Planning

Model of the environment means anything that can be used by the agent 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.

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.

Planning refers to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment.

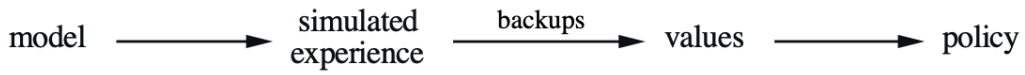
$$model \xrightarrow{planning} policy$$

State-space planning is a search through the state space for an optimal policy or an optimal path to a goal. In plan-space planning, planning is instead a search through the space of plans.

Two basic ideas:

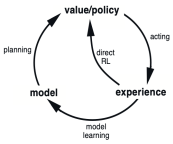
1. All state-space planning methods involve computing value functions as a key intermediate step toward improving the policy
2. They compute value functions by updates or backup operations applied to simulated experience

The common structure can be diagrammed as follows 8.1:



Planning uses simulated experience generated by a model, learning methods use real experience generated by the environment.

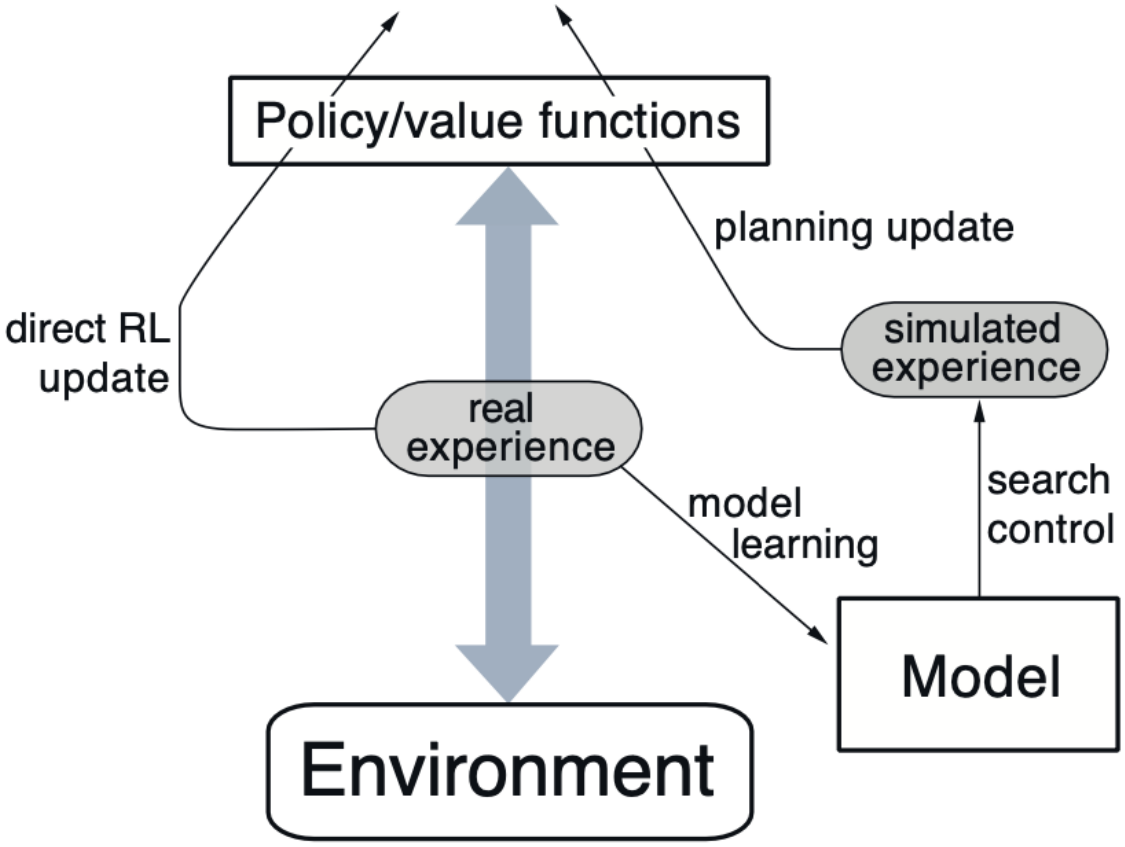
8.2 Dyna



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. The former is called 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 8.2 to the right. Each arrow shows a relationship of influence and presumed improvement.

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.

Dyna-Q includes all of the processes shown in the diagram above 8.2 - planning, acting, model-learning and direct RL - all occurring continually.



The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 8.2. 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. The pseudocode of the algorithm is shown in the box below 8.2:

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) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (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
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

The Dyna-Q+ 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. To encourage behaviour that tests long-untried actions, a special "bonus reward" is given on simulated experiences involving theses 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.

8.3 Prioritized Sweeping

Suppose that the values are initially correct given the model. 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.

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. The full algorithm for the case of deterministic environments is given in the box 8.3.

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:
$S,A \leftarrow first(PQueue)$
$R,S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
Loop for all $\bar{S},\bar{A}$ predicted to lead to S:
$\bar{R} \leftarrow$ predicted reward for $\bar{S},\bar{A},S$
$P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.
if $P > \theta$ then insert $\bar{S},\bar{A}$ into $PQueue$ with priority P

8.4 Trajectory Sampling

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.

8.5 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). RTDP is an example of an asynchronous DP algorithm. 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.

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.

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).

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.

8.6 Planning at Decision Time

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) is called 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. Unlike the first use of planning, here planning focuses on a particular state. We call this decision-time planning.

Decision-time planning is most useful in applications in which fast responses are not required. 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.

8.7 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. 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.

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.

8.8 Rollout Algorithms

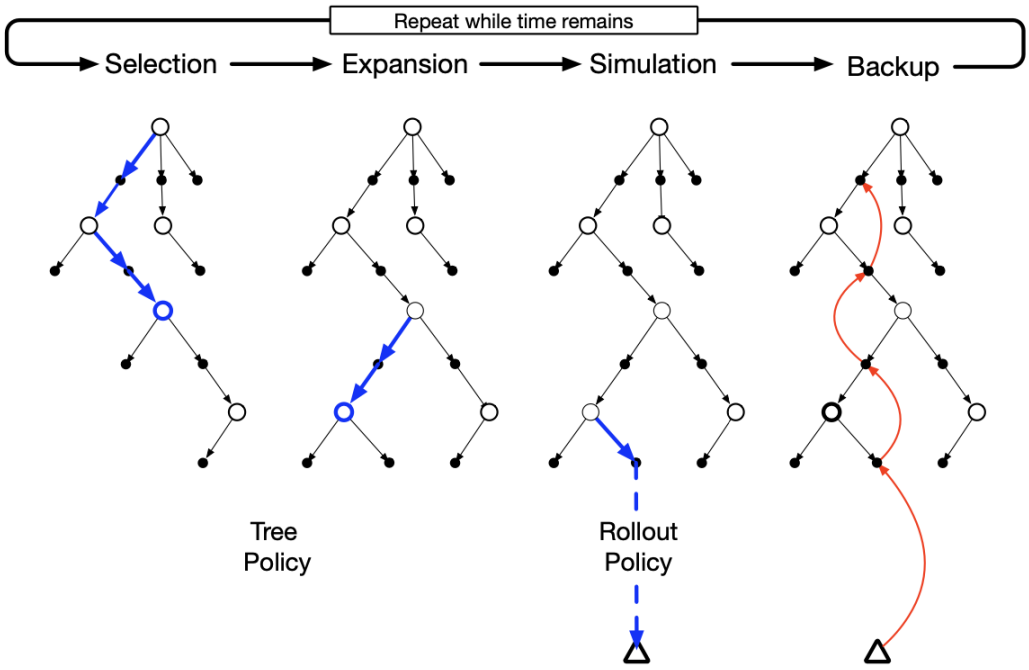
Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible 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.

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.

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.

8.9 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.



In more detail, each iteration of a basic version of MCTS consists of the following four steps as

illustrated in Figure 8.9:

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.

9 On-policy Prediction with Approximation

The approximate value function is represented not as a table but as a parametrized functional form with weight vector $\mathbf{w} \in R^d$. We will write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s under a given weight vector $\mathbf{w}$. Typically, the number of weights (the dimensionality of $\mathbf{w}$) is much less than the number of states ($d << |S|$), 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.

9.1 Value-function Approximation

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 non-stationary target functions (target functions that change over time).

9.2 The Prediction Objective

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 $\overline{VE}$:

$$\overline{VE}(\mathbf{w}) = \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2 \tag{38}$$

9.3 Stochastic-gradient and Semi-gradient Methods

In gradient-descent methods, the weight vector is a column vector with a fixed number of real valued components, $\mathbf{w}=(w_1, w_2, \ldots, w_d)_T$, and the approximate value function $\hat{v}(s, \mathbf{w})$ is a differentiable function of $\mathbf{w}$ for all $s \in S$.

Stochastic gradient-descent (SGD) methods minimize the error on the observed examples by adjusting the wight vector after each example by a small amount in the direction that would most reduce the error on that example:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\nabla\hat{v}(S_t, \mathbf{W}_t) \tag{39}$$

Gradient descent methods are called "stochastic" when the update is done, as here, on only a single example, which might have been selected stochastically. The gradient-descent version of Monte Carlo state-value prediction is guaranteed to fins a locally optimal solution. Pseudocode for a complete algorithm is shown in the box 9.3.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
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 $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Boot-strapping methods are not instances of true gradient descent. They take into account the effect of changing the weight vector 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. The do not converge as robustly as gradient methods, but they offer advantages that make them often clearly preferred. They typically enable significantly faster learning. Another is that they enable learning to be continual and online, without waiting for the end of an episode. Complete pseudocode for semi-gradient TD(0) is shown in the box 9.3.

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.

9.4 Linear Methods

Corresponding to every state s, there is a real-valued vector $\mathbf{x}(\text{s}) = (x_1(s), x_2(s), \ldots, x_d(s))^T$, with the same number of components as $\mathbf{w}$. Linear methods approximate the state-value function by the inner product between $\mathbf{w}$ and $\mathbf{x}(\text{s})$:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T\mathbf{x}(s) = \sum_{i=1}^{d} e_i x_i(s) \tag{40}$$

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ 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'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

The vector $\mathbf{x}$(s) is called a feature vector representing state s. The gradient of the approximate value function with respect to $\mathbf{w}$ in this case is:

$$\nabla\hat{v}(s,\mathbf{w}) = \mathbf{x}(s)$$

The semi-gradient TD(0) algorithm presented also converges under linear function approximation. The update at each time t is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T\mathbf{w}_t) \tag{41}$$

$$E[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t) \tag{42}$$

where $\mathbf{b} = E[R_{t+1}\mathbf{x}_t] \in R^d$ and $A = E[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T] \in R^{d\times d}$ If the system converges, it must converge to the weight vector $\mathbf{w}_{TD}$ at which

$$\mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b} \tag{43}$$

This quantity is called the TD fixed point.

The semi-gradient n-step TD algorithm is the natural extension of the tabular n-step TD algorithm. Pseudocode is given in the box 9.4:

9.5 Feature Construction for Linear Methods

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.

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer n
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | 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: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[G - \hat{v}(S_\tau,\mathbf{w})]\,\nabla\hat{v}(S_\tau,\mathbf{w})$
  Until $\tau = T - 1$

Polynomials

Polynomials make up one of the simplest families of features used for interpolation and regression.

Suppose each state s corresponds to k numbers, $s_1, s_2, \ldots, s_k$ with each $s_i \in R$. For this k-dimensional state space, each order-n polynomial-basis feature x_i can be written as:

$$x_i(s) = \prod_{j=1}^{k} s_j^{c_{i,j}} \tag{44}$$

where each $c_{i,j}$ is an integer in the set 0,1,...,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.

Fourier Basis

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 $1/\tau$).

Suppose each state s corresponds to a vector of k numbers, $\mathbf{s} = (s_1, s_2, \ldots, s_k)^T$, with each $s_i \in [0, 1]$. The ith feature in the order-n Fourier cosine basis can then be written:

$$x_i(s) = cos(\pi \mathbf{s}^T \mathbf{c}^i) \tag{45}$$

where $\mathbf{c}^i = (c_1^i, \ldots, c_k^i)^T$, with $c_j^i \in \{0, \ldots, n\}$ for j = 1, ..., k and i = 1, ..., $(n+1)^k$.

If α is the basic step-size parameter, then the step-size parameter setting suggested for feature x_i to $\alpha_i = \frac{\alpha}{\sqrt{(c_1^i)^2 + \cdots + (c_k^i)^2}}$ (except when each $c_j^i = 0$, in which case $\alpha_i = \alpha$).

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 in 9.5. 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. Representing a state with features that overlap in this way(although they need not be circles or binary) is known as coarse coding.

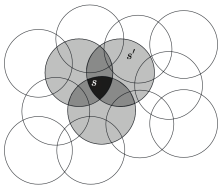
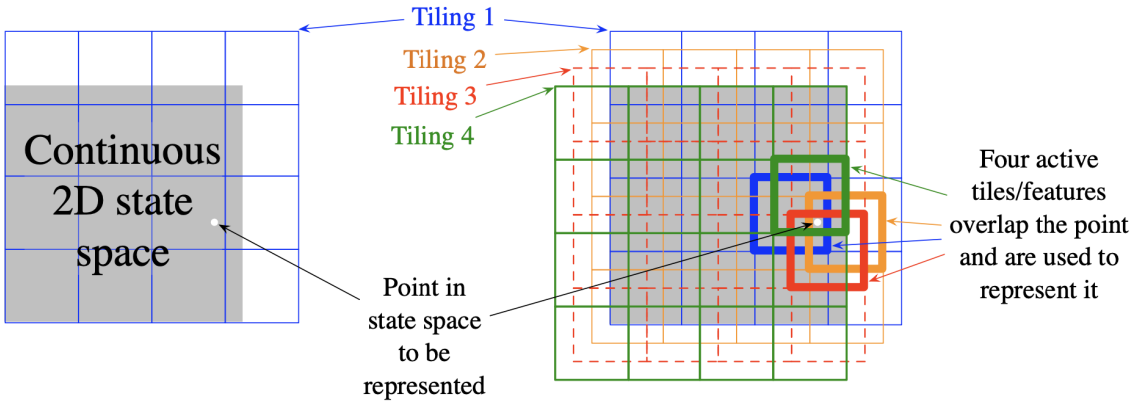


Figure 5: "Coarse Coding"

Tile Coding

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. 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 9.5.



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.

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

sqaures $\frac{w}{n}$ on a side, all states activate the same tiles, have the same feature representation, and the same approximated value.

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. The shape of the tiles will determine the nature of generalization. Tiles that are elongated along one dimension, such as the stripe tilings in Figure 9.5 (middle), will promote generalization along that dimension. The tilings in Figure 9.5 (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 9.5 (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 9.5 (left) are also possible, though rare in practice.

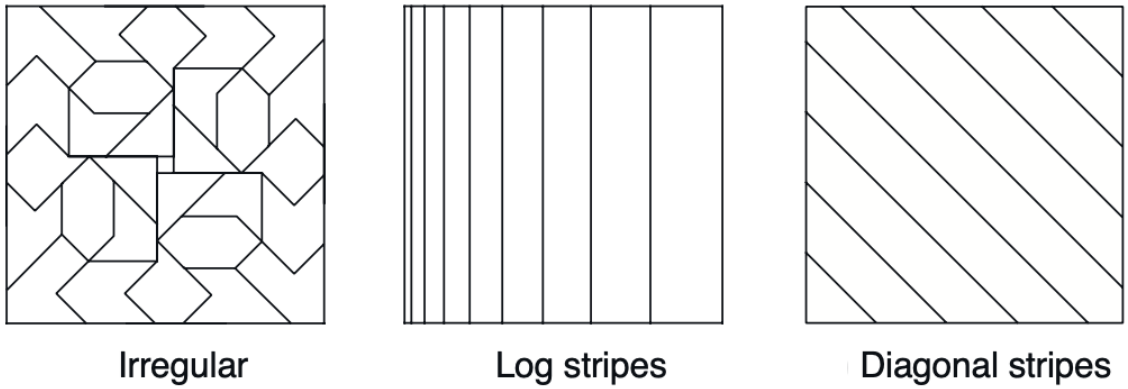


Figure 6: Tilings need not be grids. They can be arbitrarily shaped and non-uniform.

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 non-contiguous, disjoint regions randomly spread throughout the state space, but still form an exhaustive partition. 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.

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 state, s, and the feature's prototypical or centre state c_i, and relative to the feature width σ_i:

$$x_i(s) = exp(-\frac{||s - c_i||^2}{2\sigma_i^2}$$

.

9.6 Selecting Step-Size Parameters Manually

Suppose we want 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:

$$\alpha = (\tau E[\mathbf{x}^T\mathbf{x}])^{-1} \tag{46}$$

where $\mathbf{x}$ is a random feature vector chosen from the same distribution as input vectors will be in the SGD.

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. Figure 9.7 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 units (the circles in Figure 9.7) are typically semi-linear units, meaning that they compute a weighted sum of their input signals and then apply to the result of 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. 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.

Training the hidden layers of an ANN is a way to automatically create features appropriate for a given problem so that hierarchial representations can be produced without relying exclusively on hand-crafted features. ANNs typically learn by a stochastic gradient method. 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 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.

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.

Overfitting is a problem for any function approximation method that adjusts functions with many degrees of freedom on the basis of limited training data. Many methods have been developed for reducing overfitting. These include stopping training when performance begins to decrease on

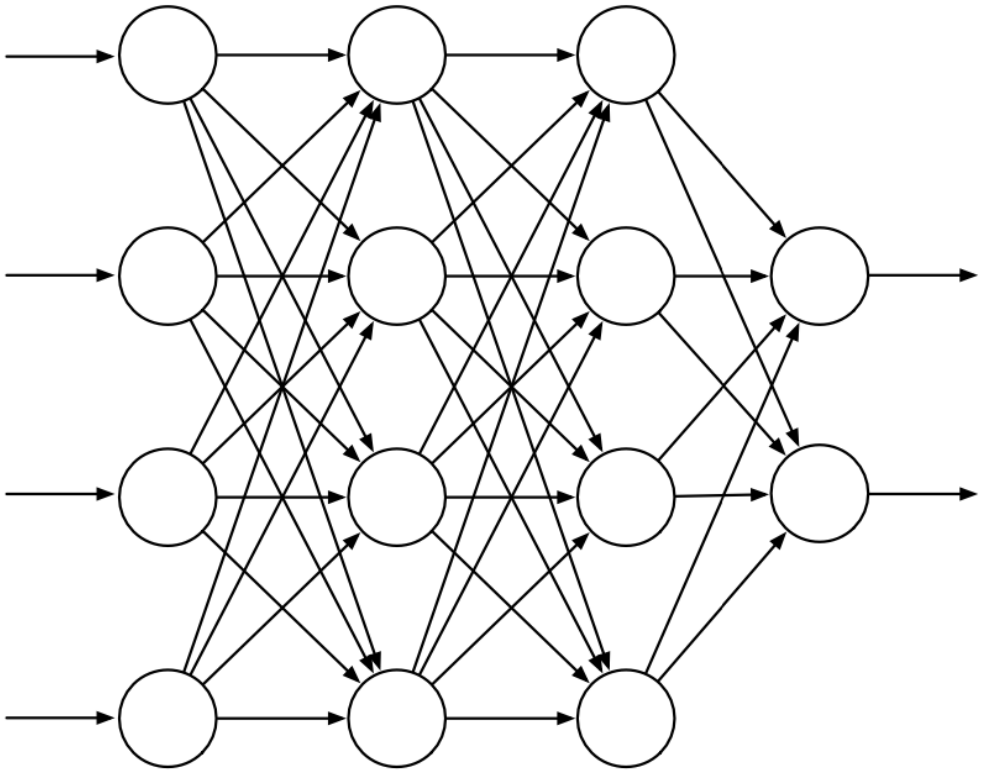


Figure 7: A generic feedforward ANN with four input units, two output units, and two hidden layers.

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. 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.

Batch normalization 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.

Another technique useful for training deep ANNs is deep residual learning. 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.

Deep Convolutional Network is specialized for processing high-dimensional data arranged in special arrays, such as images. It was inspired by how early visual processing works in the brain.

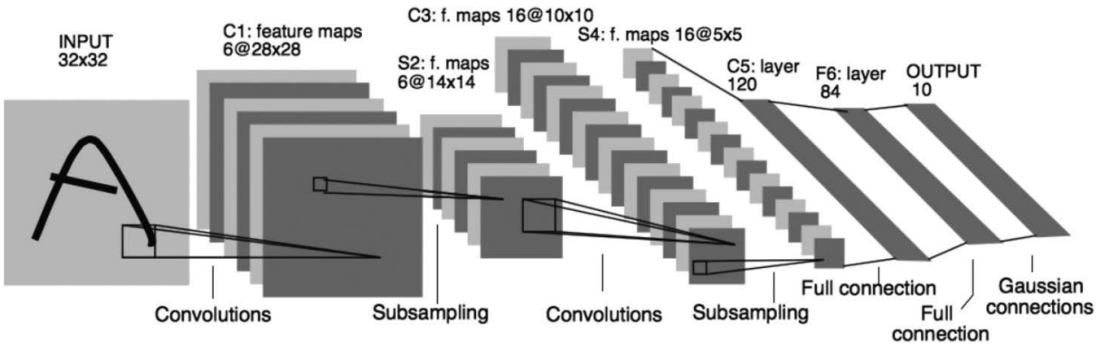


Figure 8: A generic feedforward ANN with four input units, two output units, and two hidden layers.

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 9.7 illustrates the architecture of a deep convolutional network. 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.

9.8 Least-Squares TD

The Least-Squares TD algorithm commonly known as LSTD, forms the natural estimates:

$$\hat{\mathbf{A}}_t = \sum_{k=0}^{t-1} \mathbf{x}_k (\mathbf{x}_k - \gamma \mathbf{x}_{k+1})^T + \epsilon \mathbf{I} \tag{47}$$

$$\hat{\mathbf{b}}_t = \sum_{k=0}^{t-1} R_{k+1} \mathbf{x}_k \tag{48}$$

$$\mathbf{w}_t = \hat{\mathbf{A}_t}^{-1} \hat{\mathbf{b}_t} \tag{49}$$

Fortunately, an inverse of a matrix of our special form—a sum of outer products—can also be updated incrementally with only O(d2) computations, as:

$$\hat{\mathbf{A}_t}^{-1} = \hat{\mathbf{A}_{t-1}}^{-1} - \frac{\hat{\mathbf{A}_{t-1}}^{-1} \mathbf{x}_{t-1} (\mathbf{x}_{t-1} - \gamma \mathbf{x}_t)^T \hat{\mathbf{A}_{t-1}}^{-1}}{1 + (\mathbf{x}_{t-1} - \gamma \mathbf{x}_t)^T \hat{\mathbf{A}_{t-1}}^{-1} \mathbf{x}_{t-1}} \tag{50}$$

The identity 50 is known as the Sherman-Morrison formula. The complete algorithm is shown in the box 9.8.

LSTD for estimating $\hat{v} = \mathbf{w}^\top \mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

Input: feature representation $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal) = \mathbf{0}$
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'; $\mathbf{x}' \leftarrow \mathbf{x}(S')$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top (\mathbf{x} - \gamma\mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - \big(\widehat{\mathbf{A}^{-1}}\mathbf{x}\big)\mathbf{v}^\top / \big(1 + \mathbf{v}^\top \mathbf{x}\big)$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\mathbf{w} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

9.9 Memory-based Function Approximation

Memory-based function approximation methods 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.

Local-learning methods 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. 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, 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 non-parametric, 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. 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.

Using parallel computers or special purpose hardware, we can accelerate the nearest neighbor search. 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.

9.10 Kernel-based Function Approximation

Memory-based methods such as the weighted average and locally weighted regression methods depend on assigning weights to examples in the database depending on the distance between s' and a query state s. The function that assigns these weights is called a kernel function. Viewed slightly differently, k(s,s0) is a measure of the strength of generalization from s0 to s. Kernel functions numerically express how relevant knowledge about any state is to any other state.

Kernel regression is the memory-based method that computes a kernel weighted average of all the targets of all examples stored in memory, assigning the result to the query state. If D is the set of all 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 D, as:

$$\hat{v}(s, D) = \sum_{s' \in D} k(s, s') g(s') \tag{51}$$

A common kernel is the Gaussian radial basis function (RBF) used in RBF approximation. Kernel regression with an RBF kernel is memory based: the RBFs are centered on the states of the stored examples. moreover, it is non-parametric: there are no parameters to learn.

Linear parametric regression with states represented by feature vectors $\mathbf{x}(\mathrm{s}) = (x_1(s), x_2(s), \ldots, x_d(s))^T$, can be recast as kernel regression:

$$k(s, s') = \mathbf{x}(s)^T \mathbf{x}(s') \tag{52}$$

9.11 Interest and Emphasis

A non-negative scalar measure, a random variable I_t called interest, indicates the degree to which we are interested in accurately valuing the state at time t. Another non-negative scalar random variable, the emphasis M_t multiplies the learning update and thus emphasizes or de-emphasizes the learning done at time t. The general n-step learning rule is:

$$\mathbf{w}_{t+n} = \mathbf{w}_{t+n-1} + \alpha M_t [G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1} \tag{53}$$

$$M_t = I_t + \gamma^n M_{t-n} \tag{54}$$

Bibliography

[1] Reinforcement learning statistics. https://arxiv.org/pdf/1709.06560.pdf.

[2] Sutton Barto. *Reinforcement Learning*.

[3] Emma Brunskill. CS234 stanford - Reinforcement Learning Lecture Slides. Lecture slides, 2019. Course: CS234 Reinforcement Learning.

[4] Shivaram Kalyanakrishnan. CS747 IIT Bombay - Foundations of Intelligent and Learning Agents. Lecture slides, 2022. Course: CS747 Foundations of Intelligent and Learning Agents.