CS 7642 Reading Notes

Jie Wu

Summer 2022

1 Week 1 Articles

1.1 Sutton & Barto Reinforcement Learning Chapter 1

reinforcement learning: pp.1, the two most important distinguishing features of reinforcement learning are

trial-and-error search: the learning is not told which actions to take, but instead must discover which actions yield the most reward by trying them

delayed reward: actions may affect not only the immediate reward but also the next situation and through that all subsequent rewards

pp.2, we formalize the problem of reinforcement learning using ideas from dynamical systems theory, specifically as the optimal control of incompletely-known Markov decision processes
pp.2, reinforcement learning is different from

supervised learning: 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

unsupervised learning: reinforcement learning is trying to maximize a reward signal instead of trying to find hidden structure

pp.3, one of the challenges that arise in reinforcement learning and not in other kinds of learning is the trade-off between exploration and exploitation, another key feature of reinforcement learning is that it explicitly considers the whole problem, starting with a complete, interactive, goal-seeking agent
pp.4, one of the most exciting aspects of modern reinforcement learning is its substantive and fruitful interactions with other engineering and scientific disciplines, most distinctively reinforcement leanring has also interacted strongly with psychology and neuroscience, of all the forms of machine learning reinforcement learning is the closest to the kind of learning that humans and other animals do

examples: pp.5, correct choice requires taking into account indirect, delayed consequences of actions, and thus may require foresight or planning

elements of reinforcement learning: pp.6, the four main subelements of a reinforcement learning system are

policy: a policy is a mapping from perceived states of the environment to actions to be taken when in those states, the policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior, in general policies may be stochastic specifying probabilities for each action

reward signal: 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, in general reward signals may be stochastic functions of the state of the environment and the actions taken

value function: the value of a state is the total amount of reward an agent can expect to accumulate over the future starting from that state, whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states, action choices are made based on value judgments, we seek actions that bring about states of highest value not highest reward, the most important component of almost all reinforcement learning algorithms we consider is a method for efficiently estimating values

model of the environment: mimics the behavior of the environment and 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, methods for solving reinforcement learning problems that use models and planning are called model-based methods as opposed to simpler model-free methods that are explicitly trial-and-error learners, modern reinforcement learning spans the spectrum from low-level, trial-and-error learning to high-level, deliberative planning

limitations and scope: pp.7, we encourage the reader to follow the informal meaning and think of the state as whatever information is available to the agent about its environment
pp.7, most of the reinforcement learning methods we consider in this book are structured around estimating value functions, but it is not strictly necessary to do this to solve reinforcement learning problems, e.g. evolutionary methods such as genetic algorithms; pp.8, if the space of policies is sufficiently small or can be structured so that good policies are common or easy to find—or if a lot of time is available for the search—then evolutionary methods can be effective, our focus is on reinforcement learning methods that learn while interacting with the environment which evolutionary methods do not do

an extended example: tic-tac-toe: pp.11, evolutionary and value function methods both search the space of policies, but learning a value function takes advantage of information available during the course of play, how well a reinforcement learning system can work in problems with such large state sets is intimately tied to how appropriately it can generalize from past experience, it is in this role that we have the greatest need for supervised learning methods within reinforcement learning

summary: pp.13, the use of value functions distinguishes reinforcement learning methods from evolutionary methods that search directly in policy space guided by evaluations of entire policies

early history of reinforcement learning: pp.13, the early history of reinforcement learning has two main threads that were pursued independently before intertwining in modern reinforcement learning

- learning by trial and error and originated in the psychology of animal learning
- the problem of optimal control and its solution using value functions and dynamic programming, for the most part this thread did not involve learning

the two threads were mostly independent but became interrelated to some extent around a third less distinct thread concerning temporal-difference methods such as that used in the tic-tac-toe example in this chapter, all three threads came together in the late 1980s to produce the modern field of reinforcement learning pp.14, dynamic programming is widely considered the only feasible way of solving general stochastic optimal control problems, it suffers from what Bellman called "the curse of dimensionality" meaning that its computational requirements grow exponentially with the number of state variables, but it is still far more efficient and more widely applicable than any other general method
pp.17, many researchers seemed to believe that they were studying reinforcement learning when they were actually studying supervised learning, they miss the essential character of trial-and-error learning as selecting actions on the basis of evaluative feedback that does not rely on knowledge of what the correct action should be

1.2 Sutton & Barto Reinforcement Learning Chapter 3

pp.47, MDPs involve delayed reward and the need to trade off immediate and delayed reward, as in all of artificial intelligence there is a tension between breadth of applicability and mathematical tractability

the agent-environment interface: pp.48, the agent and environment interact at each of a sequence of discrete time steps $t = 0, 1, 2, \ldots$, at each time step t the agent receives some representation of the environment's *state* $S_t \in \mathcal{S}$ and on that basis selects an *action* $A_t \in \mathcal{A}(s)$, one time step later the agent receives a numerical *reward* $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and finds itself in a new state S_{t+1}, the MDP and agent together thereby give rise to a sequence or *trajectory* that begins like this: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$, in a *finite* MDP the set of states, actions, and rewards $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ all have a finite number of elements, in this case the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action

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

the function p defines the *dynamics* of the MDP, the dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is an ordinary deterministic function of four arguments; pp.49, in a *Markov* decision process the probabilities given by p completely characterize the environment's dynamics, that is the probability of each possible value for S_t and R_t depends on the immediately preceding state and action S_{t-1} and A_{t-1} and not at all on earlier states and actions, 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*
pp.50, 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, in particular we always consider the reward computation to be external to the agent because it defines the task facing the agent and thus must be beyond its ability to change arbitrarily, the agent-environment boundary represents the limit of the agent's absolute control not of its knowledge

goals and rewards: pp.53, the agent's goal is to maximize the total amount of reward it receives, this means maximizing not immediate reward but (the expected value of) cumulative reward in the long run

returns and episodes: pp.54, in some cases 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, tasks with episodes of this kind are called *episodic tasks*, 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*

pp.54, in general we seek to maximize the *expected return* where the return denoted G_t is defined as some specific function of the reward sequence; pp.55, the expected discounted return is

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

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*, as γ approaches 1, the return objective takes future rewards into account more strongly, the agent becomes more farsighted

unified notation for episodic and continuing tasks: pp.57, these two can be unified by considering episode termination to be the entering of a special *absorbing state* that transitions only to itself and that generates only rewards of zero

policies and value functions: pp.58, formally a *policy* is a mapping from states to probabilities of selecting each possible action, if the agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$, 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

$$v_\pi(s) := \mathbb{E}_\pi\left[G_t | S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

we call the function v_π the *state-value function* for policy π

pp.58, similarly we define the value of taking action a in state s under a policy π denoted $q_\pi(s,a)$ as the expected return starting from s taking the action a and thereafter following policy π

$$q_\pi(s,a) := \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

pp.59, a fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy the following recursive relationship

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

which is the Bellman equation for v_π, it states that the value of the start state must equal the (discounted) value of the expected next state plus the reward expected along the way, the value function v_π is the unique solution to its Bellman equation

optimal policies and optimal value functions: pp.62, a policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states, in other words $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$, there is always at least one policy that is better than or equal to all other policies, this is an *optimal policy*, although there may be more than one, we denote all the optimal policies by π_*, they share the same state-value function called the *optimal state-value function* denoted v_* and defined as

$$v_*(s) := \max_\pi v_\pi(s) \qquad \forall s \in \mathcal{S}$$

optimal policies also share the same *optimal action-value function* denoted q_* and defined as

$$q_*(s,a) := \max_\pi v_\pi(s,a) \qquad \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$$

for the state-action pair (s,a) this function gives the expected return for taking action a in state s and thereafter following an optimal policy, thus we can write q_* in terms of v_* as follows

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \middle| S_t = s, A_t = a\right]$$

pp.63, because it is the optimal value function, v_*'s consistency condition can be written in a special form without reference to any specific policy, this is the Bellman equation for v_* or the Bellman optimality equation

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \middle| S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right] \end{aligned}$$

the Bellman optimality equation for q_* is

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1},a') \middle| S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right] \end{aligned}$$

pp.64, for finite MDPs the Bellman optimality equation for v_* has a unique solution, once one has v_* it is relatively easy to determine an optimal policy, for each state s there will be one or more actions at which the maximum is obtained in the Bellman optimality equation, any policy that assigns nonzero probability only to these actions is an optimal policy, another way of saying this is that any policy that is *greedy* with respect to the optimal evaluation function v_* is an optimal policy (the term greedy is used in computer science to describe any search or decision procedure that selects alternatives based only on local or immediate considerations, without considering the possibility that such a selection may prevent future access to even better alternatives)
pp.64, having q_* makes choosing optimal actions even easier: for any state s it can simply find any action that maximizes $q_*(s,a)$, at the cost of representing a function of state-action pairs instead of just of states, the optimal action-value function allows optimal actions to be

selected without having to know anything about possible successor states and their values
pp.66, explicitly solving the Bellman optimality equation relies on at least three assumptions that are rarely true in practice

1. the dynamics of the environment are accurately known
2. computational resources are sufficient to complete the calculation
3. the states have the Markov property

thus in reinforcement learning one typically has to settle for approximate solutions, many reinforcement learning methods can be clearly understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions

optimality and approximation: pp.67, even if we have a complete and accurate model of the environment's dynamics, it is usually not possible to simply compute an optimal policy by solving the Bellman optimality equation, a critical aspect of the problem facing the agent is always the computational power available to it, the memory available is also an important constraint, in tasks with small finite state sets, it is possible to form these approximations using arrays or tables with one entry for each state (or state-action pair), this we call the *tabular* case, in many cases of practical interest there are far more states than could possibly be entries in a table
pp.68, the online nature of reinforcement learning makes it possible to approximate optimal policies in ways that put more effort into learning to make good decisions for frequently encountered states at the expense of less effort for infrequently encountered states, this is one key property that distinguishes reinforcement learning from other approaches to approximately solving MDPs

summary: pp.68, the *actions* are the choices made by the agent, the *states* are the basis for making the choices, and the *rewards* are the basis for evaluating the choices, a *policy* is a stochastic rule by which the agent selects actions as a function of states, the agent's objective is to maximize the amount of reward it receives over time
pp.68, whereas the optimal value functions for states and state-action pairs are unique for a given MDP, there can be many optimal policies, any policy that is greedy with respect to the optimal value functions must be an optimal policy
pp.69, in problems of *complete knowledge* the agent has a complete and accurate model of the environment's dynamics, if the environment is an MDP then such a model consists of the complete four-argument dynamics function p

1.3 Sutton & Barto Reinforcement Learning Chapter 16

TD-Gammon: pp.421, the learning algorithm in TD-Gammon was a straightforward combination of the TD(λ) algorithm and nonlinear function approximation using a multilayer artificial neural network (ANN) trained by backpropagating TD errors
pp.425, TD-Gammon 0.0 was constructed with essentialy zero backgammon knowledge, that it was able to do as well as Neurogammon and all other approaches is striking testimony to the potential of self-play learning methods, adding the specialized backgammon features but keeping the self-play TD learning method produced TD-Gammon 1.0, in follow-on

work Tesauro and Galperin (1997) explored trajectory sampling methods as an alternative to full-width search

Samuel's checkers player: pp.426, Samuel's program used what we now call heuristic search methods to determine how to expand the search tree and when to stop searching, in particular Samuel's program was based on Shannon's minimax procedure to find the best move from the current position, when the minimax procedure reached the search tree's root—the current position—it yielded the best move under the assumption that the opponent would be using the same evaluation criterion, shifted to its point of view
pp.427, rote learning and other aspects of Samuel's work strongly suggest the essential idea of temporal-difference learning—that the value of a state should equal the value of likely following states

Watson's Daily-Double Wagering: pp.431, the TD-Gammon method of self-play was not used, because unlike backgammon Jeopardy! is a game of imperfect information because contestants do not have access to all the information influencing their opponents' play, in particular Jeopardy! contestants do not know how much confidence their opponents have for responding to clues in the various categories, self-play would have been something like playing poker with someone who is holding the same cards that you hold
pp.431, selecting endgame DD bets in live play based on Monte-Carlo trials instead of the ANN significantly improved WATSON's performance, because errors in value estimates in endgames could seriously affect its chances of winning

optimizing memory control: pp.433, they modeled the DRAM access process as an MDP whose states are the contents of the transaction queue and whose actions are commands to the DRAM system: precharge, active, read, write, and NoOp, the reward signal is 1 whenever the action is read or write, and otherwise it is 0, state transitions were considered to be stochastic; pp.434, critical to this MDP are constraints on the actions available in each state, in this application the integrity of the DRAM system was assured by not allowing actions that would violate timing or resource constraints, these constraints explain why the MDP has a NoOp action and why the reward signal is 0 except when a read or write command is issued, NoOp is issued when it is the sole legal action in a state
pp.434, to approximate the action-value function, the algorithm used linear function approximation implemented by tile coding with hashing

human-level video game play: pp.436, most successful applications of reinforcement learning owe much to sets of features carefully handcrafted based on humen knowledge and intuition about the specific problem to be tackled, a team of researchers at Google DeepMind developed an impressive demonstration that a deep multi-layer ANN can automate the feature design process
pp.437, Mnih et al. developed a reinforcement learning agent called deep Q-network (DQN) that combined Q-learning with a deep convolutional ANN, DQN is similar to TD-Gammon in using a multi-layer ANN as the function approximation method for a semi-gradient form of a TD algorithm, with the gradients computed by the backpropagation algorithm, however instead of using TD(λ) as TD-Gammon did, DQN used the semi-gradient form of Q-learning; pp.438, being model-free and off-policy made Q-learning a natural choice
pp.438, because the full states of many of the Atari games are not completely observable from the image frames, Mnih et al. "stacked" the four most recent frames so that the inputs

to the network had dimension $84 \times 84 \times 4$, this did not eliminate partial observability for all of the games, but it was helpful in making many of them more Markovian, because no game-specific prior knowledge beyond this minimal amount was used in preprocessing the image frames, we can think of the $84 \times 84 \times 4$ input vectors as being "raw" input to DQN
pp.440, instead of S_{t+1} becoming the new S_t for the next update as it would in the usual form of Q-learning, a new unconnected experience was drawn from the replay memory to supply data for the next update, because Q-learning is an off-policy algorithm it does not need to be applied along connected trajectories, experience reply reduced the variance of the updates, because successive updates were not correlated with one another as they would be with standard Q-learning, and by removing the dependence of successive experiences on the current weights, experience replay eliminated one source of instability
pp.441, DQN advanced the state-of-the-art in machine learning by impressively demonstrating the promise of combining reinforcement learning with modern methods of deep learning

mastering the game of Go: pp.442, where in addition to reinforcement learning AlphaGo relied on supervised learning from a large database of expert human moves, AlphaGo Zero used only reinforcement learning and no human data or guidance beyond the basic rules of the game (hence the "Zero" in its name)
pp.442, in many ways both AlphaGo and AlphaGo Zero are descendants of Tesauro's TD-Gammon, itself a descendant of Samuel's checkers player, all these programs included reinforcement learning over simulated games of self-play, AlphaGo and AlphaGo Zero also built upon the progress made by DeepMind on playing Atari games with the program DQN that used deep convolutional ANNs to approximate optimal value functions
pp.443, experts agree that the major stumbling block to creating stronger-than-amateur Go programs is the difficulty of defining an adequate position evaluation function, a major step forward was the introduction of Monte Carlo tree search (MCTS) to Go programs, which is a decision-time planning procedure that does not attempt to learn and store a global evaluation function

AlphaGo: pp.444, the main innovation that made AlphaGo such a strong player is that it selected moves by a novel version of MCTS that was guided by both a policy and a value function learned by reinforcement learning with function approximation provided by deep convolutional ANNs, another key feature is that instead of reinforcement learning starting from random network weights, it started from weights that were the result of previous supervised learning from a large collection of human expert moves
pp.444, in contrast to basic MCTS which extends its current search tree by using stored action values to select an unexplored edge from a leaf node, APV-MCTS as implemented in AlphaGo expanded its tree by choosing an edge according to probabilities supplied by a 13-layer deep convolutional ANN trained previously by supervised learning, also in contrast to basic MCTS which evaluates the newly-added state node solely by the return of a rollout initiated from it, APV-MCTS evaluated the node in two ways—by this return of the rollout but also by a value function learned previously by a reinforcement learning method, throughout its execution APV-MCTS kept track of how many simulations passed through each edge of the search tree, and when its execution completed the most-visited edge from the root node was selected as the action to take, here the move AlphaGo actually made in a game
pp.446, the team actually found that AlphaGo played better against human opponents

when APV-MCTS used as the SL policy instead of the RL policy, interestingly the situation was reversed for the value function used by APV-MCTS, they found that when APV-MCTS used the value function derived from the RL policy, it performed better than if it used the value function derived from the SL policy
pp.447, the best play resulted from setting $\eta = 0.5$, indicating that combining the value network with rollouts was particularly important to AlphaGo's success—the value network evaluated the high-performance RL policy that was too slow to be used in live play, while rollouts using the weaker but much faster rollout policy were able to add precision to the value network's evaluations

AlphaGo Zero: pp.447, AlphaGo Zero implemented a form of policy iteration interleaving policy evaluation with policy improvement, a significant difference between AlphaGo Zero and AlphaGo is that AlphaGo Zero used MCTS to select moves throughout self-play reinforcement learning, whereas AlphaGo used MCTS for live play after learning, besides AlphaGo Zero's MCTS was simpler than the version used by AlphaGo in that it did not include rollouts of complete games and therefore did not need a rollout policy
pp.448, because repetition is not allowed in Go and one player is given some number of "compensation points" for not getting the first move, the current board position is not a Markov state of Go, this is why features describing past board positions and the color feature were needed; pp.449, extra noise was added to the network's output $\mathbf{p}$ to encourage exploration of all possible moves, they used the Elo rating system to evaluate the relative performances of the programs

personalized web services: pp.450, a reinforcement learning system can improve a recommendation policy by making adjustments in response to user feedback, one way to obtain user feedback is by means of website satisfaction surveys, but for acquiring feedback in real time it is common to monitor user clicks as indicators of interest in a link, a method long used in marketing called A/B testing is a simple type of reinforcement learning used to decide which of two versions A or B of a website users prefer, better results are possible by formulating personalized recommendation as a Markov decision problem (MDP) with the objective of maximizing the total number of clicks users make over repeated visits to a website vs. by not using the fact that many users repeatedly visit the same websites greedy policies do not take advantage of possibilities provided by long-term interactions with individual users
pp.451, Theocharous et al. compared the results of two algorithms for learning ad recommendation policies, the first algorithm which they called greedy optimization had the goal of maximizing only the probability of immediate clicks, the other algorithm, a reinforcement learning algorithm based on an MDP formulation, aimed at improving the number of clicks users made over multiple visits to a website, they called this latter algorithm life-time value (LTV) optimization, both algorithms faced challenging problem because the reward signal in this domain is very sparse because users usually do not click on ads; pp.453, assured by these probabilistic guarantees Adobe announced in 2016 that the new LTV algorithm would be a standard component of the Adobe marketing cloud

thermal soaring: pp.453, birds and gliders take advantage of upward air currents–thermals–to gain altitude in order to maintain flight while expending little or no energy, this behavior is called thermal soaring, Reddy et al. modeled the soaring problem as a continuing MDP with discounting, the agent interacted with a detailed model of a glider flying in turbulent air; pp.454, for the learning experiments air flow was modeled by a sophisticated physics-

based set of partial differential equations, introducing small random perturbations into the numerical simulation caused the model to produce analogs of thermal updrafts and accompanying turbulence
pp.455, the overall objective of thermal soaring is to gain as much altitude as possible from each rising column of air, Reddy et al. tried a straightforward reward signal that rewarded the agent at the end of each episode based on the altitude gained over the episode, they found that learning was not successful with this reward signal, by experimenting with various reward signals they found that learning was best with a reward signal that at each time step linearly combined the vertical wind velocity and vertical wind acceleration observed on the previous time step, learning was by one-step Sarsa with actions selected according to a soft-max distribution based on normalized action values
pp.456, due to the fact that soaring in different levels of turbulence requires different policies, training was done in conditions ranging from weak to strong turbulence, different levels of turbulence led to policy differences, policies learned in strong turbulence were more conservative in that they preferred small bank angles, whereas in weak turbulence the best action was to turn as much as possible by banking sharply, systematic study of the bank angles preferred by the policies learned under the different conditions led the authors to suggest that by detecting when vertical wind acceleration crosses a certain threshold the controller can adjust its policy to cope with different turbulence regimes; pp.457, they found that the altitude gained in an episode increased as γ increased reaching a maximum for $\gamma = 0.99$ suggesting that effective thermal soaring requires taking into account long-term effects of control decisions
pp.457, learning policies having access to different sets of environmental cues and control actions contributes to both the engineering objective of designing autonomous gliders and the scientific objective of improving understanding of the soaring skills of birds

1.4 Littman PhD Dissertation Chapter 1

sequential decision making: pp.5, a more accurate perspective would consider the environment to be non-stationary, such environment are of interest because they make it possible to consider a broader range of sequential decision-making problems, however they are more complex mathematically and few formal models have been proposed
pp.7, stationary policies have several important properties that make them extremely important

- in highly unpredictable environments nearly any state can follow any other state, so any partial list of contingencies would be inadequate
- it is hard to imagine finding optimal behavior without reasoning about action choices in all possible states

pp.8, my purpose in this thesis is to examine methods for producing policies that maximize a measure of the long-run reward to an agent following it in a specific environment, these policies can be produced under two different problem scenarios that differ in the information available for constructing the policy

planning: a complete model of the environment is known in advance, this makes it possible to separate the decision-making problem into two components—the planner and the agent, the planner is responsible for taking a description of the environment and generating a policy, this policy is then downloaded into the agent for execution in the environment

reinforcement learning: for a planner to function it must have a complete description of the environment's states, actions, rewards, and transitions, reinforcement learning can be used when a model of the environment is unknown or difficult to work with directly, the only access a reinforcement-learning agent has to information about its environment is via perception and action, reinforcement learning removes the distinction between planner and agent—a reinforcement-learning agent is responsible for gathering information about the environment, organizing it to determine a policy, and behaving according to the policy, from this perspective reinforcement learning and planning are closely related—a reinforcement learner carries out its planning in the context of direct interactions with the environment

pp.9, although reinforcement learning is defined by the absence of an a priori model of the environment vs. planning by the presence of such a model, in

model-based reinforcement learning: an agent uses its experience with the environment to construct an approximate model

simulated reinforcement learning: a reinforcement-learning agent is introduced into an environment with a known structure, but is forced to behave as if the structure is not known, which can actually be a good idea if the environment is complex and building a complete universal plan is infeasible, using a reinforcement-learning algorithm in such an environment can help the agent find appropriate behavior for the most common and important states

formal models: pp.10–11,

finite vs. continuous: I will focus on finite state spaces, be mainly concerned with the finite case

episodic vs. sequential: in an episodic environment the agent faces the same problem over and over again, I am more concerned with sequential environments in which the agent makes a sequence of interrelated decisions without necessarily being reset to a starting state

accessible vs. inaccessible: if the observations it makes are sufficient to reveal the entire state of the environment, the environment is accessible or completely observable, otherwise it is inaccessible or partially observable

Markovian vs. non-Markovian: in a Markovian environment the future evolution of the system can be predicted on the basis of the environment's state, in non-Markovian environments it is often important to remember something about earlier states to predict the future accurately

fixed vs. dynamic: I focus exclusively on the simpler fixed environments

deterministic vs. stochastic: deterministic means that there is exactly one next state for each combination of state and action

synchronous vs. asynchronous: synchronous means precisely one state transition in the environment occurs for each action the agent takes, in asynchronous models the environment does not wait for the agent to take an action but instead changes continually, the actions serve as synchronizing events during which the agent and environment interact, it is possible to approximate an asynchronous environment by a synchronous one by discretizing time

single vs. multiple agent:

pp.12, in summary the environments covered in this thesis share the properties of being finite state, finite action, sequential, Markovian, fixed, and synchronous, can be completely or partially observable, stochastic or deterministic, and can contain one or two agents

Markov decision process (MDP): fixed, stochastic environments in which a single agent issues actions given knowledge of the current state

Markov game: generalize MDP to allow a pair of agents to control state transitions

partially observable Markov decision process (POMDP): consists of a single agent that must make decisions given only partial knowledge of its current state

incomplete-information game: multiple agents control the transitions in the environment, and the agents have incomplete and perhaps differing knowledge of the environment's state

evaluation criteria: pp.15–17,

transition values: three types

- rewards—each transition is replaced by its reward value
- steps—each transition is assigned a constant value
- goals—all transitions are assigned a value of zero unless the transition is to a goal state

horizon truncation: infinite—no cut off vs. finite—cut off at a prescribed length vs. goal—cut off when it first enters a goal

sequence summary: discounted—a sum in which later terms are scaled down according to the discount factor β vs. undiscounted—simple sum vs. average

value summary: expected—probability-weighted average of sequence values vs. worst case vs. best case—largest sequence value

other agents: none vs. cooperative—other agents act so as to maximize the objective function vs. competitive

pp.15, the algorithms in this thesis find policies that maximize or minimaximize expected discounted reward over the infinite horizon, all the other objective functions can be viewed as special cases of the infinite-horizon expected discounted reward objective function
pp.18, the complexity of a planning algorithm is its worst-case average run time expressed as a function of the size of a description of the environment, primarily the number of transition probabilities and rewards
pp.19, the problems in RP, P, and NC are the only ones known to be solvable in polynomial time, for the other classes the best known algorithms take exponential time, however the range of difficulty among these classes is large

NP $\cap$ co-NP: solvable in polynomial time by the best algorithm

NP-complete: can often be approximated in polynomial time

PSPACE: often quite difficult to approximate

EXPTIME-complete: not to be solvable in polynomial time in the worst case

much of the uncertainty here derives from the famous open problem—does P equal NP?
pp.20, in a reinforcement-learning scenario the agent must solve the same basic problem faced in planning, but must do so without a correct description of the environment, as a result the range of choices for evaluating reinforcement-learning algorithms are quite large

2 Week 2 Articles

2.1 Sutton & Barto Reinforcement Learning Chapter 4

pp.73, classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they provide an essential foundation for the understanding of the methods presented in the rest of this book
pp.73, starting with this chapter we usually assume that the environment is a finite MDP, that is we assume that its state, action, and reward sets $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ are finite, in this chapter we show how DP can be used to compute the value functions defined in Chapter 3

policy evaluation (prediction): pp.74, the sequence $\{v_k\}$ (where $\pi(a|s)$ is the probability of taking action a in state s under policy π)

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

which replaces the old value of s with a new value obtained from the old values of the successor states of s, 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

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

this algorithm is called *iterative policy evaluation*
pp.75, to write a sequential computer program to implement iterative policy evaluation, you could use one array and update the values "in place", that is, with each new value immediately overwriting the old one, for the in-place algorithm the order in which states have their values updated during the sweep has a significant influence on the rate of convergence, we usually have the in-place version in mind when we think of DP algorithms

policy improvement: pp.78, *policy improvement theorem* states that, let π and π' be any pair of deterministic policies such that for all $s \in \mathcal{S}$

$$q_\pi\left(s, \pi'(s)\right) \geq v_\pi(s) \qquad q_\pi(s,a) := \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$$

then the policy π' must be as good as or better than π, that is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$: $v_{\pi'}(s) \geq v_\pi(s)$, moreover the strict inequality

of the former implies the strict inequality of the latter
pp.79, consider the new greedy policy π' given by

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

by construction the greedy policy meets the conditions of the policy improvement theorem, so we know that it is as good as or better than the original policy, the process of making a new policy that improves on an original policy by making it greedy with respect to the value function of the original policy is called *policy improvement*
suppose the new greedy policy π' is as good as but not better than the old policy π, then $v_\pi = v_{\pi'}$, it follows that for all $s \in \mathcal{S}$

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

which is exactly the Bellman optimality equation, policy improvement thus must give us a strictly better policy except when the original policy is already optimal
pp.79, in the general case a stochastic policy π specifies probabilities $\pi(a|s)$ for taking each action a in each state s, all the ideas of this section extend easily to stochastic policies, in particular the policy improvement theorem carries through as stated for the stochastic case, in addition any apportioning scheme is allowed as long as all submaximal actions are given zero probability

policy iteration: pp.80, because a finite MDP has only a finite number of deterministic policies, this policy improvement 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*, note that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy, this typically results in a great increase in the speed of convergence of policy evaluation (presumably because the value function changes little from one policy to the next); pp.82, policy iteration often converges in surprisingly few iterations

value iteration: pp.82, one drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set, in fact the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration, one important special case is when policy evaluation is stopped after just one sweep (one update of each state), this algorithm is called *value iteration*, it can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps

$$v_{k+1}(s) := \max_a \sum_{s',r} p(s',r|s,a)\,[r + \gamma v_k(s')] \qquad \forall s \in \mathcal{S}$$

pp.83, note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule, also note how the value iteration update is identical to the policy evaluation update except that it requires the maximum to be taken over all actions, like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_*, in practice we stop once the value function changes by only a small amount in a sweep; pp.83, value iteration effectively combines in each of its sweeps one sweep of policy evaluation and one sweep of policy improvement, faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep, in general the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates

asynchronous dynamic programming: pp.85, 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, 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, however it can't ignore any state after some point in the computation, asynchronous DP algorithms allow great flexibility in selecting states to update

pp.85, 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, e.g. we can apply updates to states as the agent visits them, this makes it possible to focus the DP algorithm's updates onto parts of the state set that are most relevant to the agent, this kind of focusing is a repeated theme in reinforcement learning

generalized policy iteration: pp.86, 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, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function, this implies that the Bellman optimality equation holds and thus that the policy and the value function are optimal

efficiency of dynamic programming: pp.87, DP may not be practical for very large problems, but compared with other methods for solving MDPs, DP methods are actually quite efficient, in the worst case the time that DP methods take to find an optimal policy is polynomial in the number of states and actions, in this sense DP is exponentially faster than any direct search in policy space could be, linear programming methods can also be used to solve MDPs, and in some cases their worst-case convergence guarantees are better than those of DP methods, but linear programming methods become impractical at a much smaller number of states than do DP methods (by a factor of about 100), for the largest problems only DP methods are feasible

pp.88, on problems with large state spaces asynchronous DP methods are often preferred, for some problems even one sweep is impractical, yet the problem is still potentially solvable because relatively few states occur along optimal solution trajectories, asynchronous methods and other variations of GPI can be applied in such cases, and may find good or optimal policies much faster than synchronous methods can

summary: pp.88, *policy evaluation* refers to the (typically) iterative computation of the value

function for a given policy, *policy improvement* refers to the computation of an improved policy given the value function for that policy, putting these two computations together, we obtain *policy iteration* and *value iteration*, the two most popular DP methods
pp.88, it is not necessary to perform DP methods in complete sweeps through the state set, asynchronous DP methods are in-place iterative methods that update states in an arbitrary order, perhaps stochastically determined and using out-of-date information
pp.89, all of them update estimates of the value of states based on estimates of the values of successor states, that is they update estimates on the basis of other estimates, we call this general idea *boostrapping*

2.2 Sutton & Barto Reinforcement Learning Chapter 5

pp.91, unlike the previous chapter here we do not assume complete knowledge of the environment, Monte Carlo method require only experience—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment, 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)
pp.91, Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns, to ensure that well-defined returns are available, here we define Monte Carlo methods only for episodic tasks—only on the completion of an episode are value estimates and policies changed, Monte Carlo methods can thus be incremental in an episode-by-episode sense, but not in a step-by-step (online) sense, Monte Carlo methods sample and average returns for each state-action pair
pp.91, the return after taking an action in one state depends on the action taken in later states in the same episode, because all the action selections are undergoing learning, the problem becomes nonstationary from the point of view of the earlier state, to handle the nonstationarity we adapt the idea of general policy iteration (GPI) developed in Chapter 4 for DP, whereas there we computed value functions from knowledge of the MDP, here we learn value functions from sample returns with the MDP

Monte Carlo prediction: pp.92, the *first-visit MC method* estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas the *every-visit MC method* averages the returns following all visits to s, these two Monte Carlo (MC) methods are very similar but have slightly different theoretical properties, first-visit MC has been most widely studied and is the one we focus on in this chapter, every-visit MC extends more naturally to function approximation and eligibility traces as discussed in Chapters 9 and 12
pp.93, both first-visit MC and every-visit MC converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to infinity, this is easy to see for the case of first-visit MC—each return is an independent identically distributed estimate of $v_\pi(s)$ with finite variance, by the law of large numbers the sequence of averages of these estimates converges to their expected value, every-visit MC is less straightforward, but its estimates also converge quadratically to $v_\pi(s)$
pp.94–95, the three advantages of Monte Carlo methods over DP methods are

- generating the sample games required by Monte Carlo methods is easy, this is the case surprisingly often, the ability of Monte Carlo methods to work with sample episodes alone can be a significant advantage even when one has complete knowledge of the environment's dynamics

- an important fact about Monte Carlo methods is that the estimates for each state are independent, the estimate for one state does not build upon the estimate of any other state, as is the case in DP, in other words Monte Carlo methods do not bootstrap as we defined it in the previous chapter
- the computational expense of estimating the value of a single state is independent of the number of states, this can make Monte Carlo methods particularly attractive when one requires the value of only one or a subset of states—one can generate many sample episodes starting from the states of interest, averaging returns from only these states ignoring all others, this is a third advantage Monte Carlo methods can have over DP methods (after the ability to learn from actual experience and from simulated experience)

Monte Carlo estimation of action values: pp.96, if a model is not available then it is particularly useful to estimate action values (the values of state-action pairs) rather than state values, because 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 goals for Monte Carlo methods is to estimate q_*
pp.96, the policy evaluation problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π, a state-action pair s, a is said to be visited in an episode if ever the state s is visited and action a is taken in it, the first-visit MC method averages the returns following the first time in each episode that the state was visited and the action was selected, these methods converge quadratically
pp.96, 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, this is the general problem of *maintaining exploration*, 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*, it is sometimes useful, but of course it cannot be relied upon in general, particularly when learning directly from actual interaction with an environment, the most common alternative approach to assuring that all state-action pairs are encountered is to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state

Monte Carlo control: pp.97, in a Monte Carlo version of classical policy iteration, we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy and optimal action-value function

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} q_{\pi_*}$$

where $\xrightarrow{E}$ denotes a complete policy evaluation and $\xrightarrow{I}$ a complete policy improvement

policy evaluation: many episodes are experienced with the approximate action-value function approaching the true function asymptotically

policy improvement: we have an action-value function and therefore no model is needed to construct the greedy policy—for any action-value function q the corresponding greedy policy is the one that for each $s \in \mathcal{S}$ deterministically chooses an action with maximal action-value

$$\pi(s) := \arg\max_a q(s, a)$$

policy improvement theorem: $q_{\pi_k}(s, \pi_{k+1}(s)) \geq v_{\pi_k}(s)$ applies for all $s \in \mathcal{S}$

in this way Monte Carlo methods can be used to find optimal policies given only sample episodes and no other knowledge of the environment's dynamics
pp.98, one approach to avoid the infinite number of episodes is that measurements and assumptions are made to obtain bounds on the magnitude and probability of error in the estimates, and then sufficient steps are taken during each policy evaluation to assure that these bounds are sufficiently small, however it is also likely to require far too many episode to be useful in practice on any but the smallest problems
a second approach is to give up trying to complete policy evaluation before returning to policy improvement, for Monte Carlo policy iteration it is natural to alternate between evaluation and improvement on an episode-by-episode basis, after each episode the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode
pp.99, stability is achieved only when both the policy and the value function are optimal, convergence to this optimal fixed point seems inevitable as the changes to the action-value function decrease over time, but has not yet been formally proved, in our opinion this is one of the most fundamental open theoretical questions in reinforcement learning

Monte Carlo control without exploring starts: pp.100, there are two approaches to ensuring this, resulting in what we call on-policy methods and off-policy methods

on-policy: evaluate or improve the policy that is used to make decisions

off-policy: evaluate or improve a policy different from that used to generate the data

the Monte Carlo ES (exploring-start) method developed above is an example of an on-policy model, in this section we show how an on-policy Monte Carlo control method can be designed that does not use the unrealistic assumption of exploring starts, off-policy methods are considered in the next section
pp.100, in on-policy control methods the policy is generally soft, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}$ but gradually shifted closer and closer to a deterministic optimal policy, the on-policy method we present in this section uses ϵ-greedy policies, that is all nongreedy actions are given the minimal probability of selection $\epsilon/|\mathcal{A}(s)|$ and the remaining bulk of the probability $1 - \epsilon + \epsilon/|\mathcal{A}(s)|$ is given to the greedy action
pp.101, the overall idea of on-policy Monte Carlo control is still that of GPI, as in Monte Carlo ES we use first-visit MC methods to estimate the action-value function for the current policy, however without the assumption of exploring starts we cannot simply improve the policy by making it greedy with respect to the current value function, in our on-policy method we will move it only to an ϵ-greedy policy, for any ϵ-soft policy π, any ϵ-greedy policy with respect to q_π is guaranteed to be better than or equal to π, that any ϵ-greedy policy with respect to q_π is an improvement over any ϵ-soft policy π is assured by the

policy improvement theorem—let π' be the ϵ-greedy policy, the conditions of the policy improvement theorem apply because for any $s \in \mathcal{S}$ we have

$$q_\pi\left(s, \pi'(s)\right) \geq \frac{\epsilon}{|\mathcal{A}(s)|}\sum_a q_\pi(s,a) + (1-\epsilon)\sum_a \frac{\pi(a|s) - \frac{\epsilon}{|\mathcal{A}(s)|}}{1-\epsilon} q_\pi(s,a) = v_\pi(s)$$

equality can hold only when both π' and π are optimal among the ϵ-soft policies, that is when they are better than or equal to all other ϵ-soft policies
pp.102, policies being ϵ-soft can be moved inside the environment—the new environment has the same action and state set as the original, and if in state s and taking action a with probability $1-\epsilon$ the new environment behaves exactly like the old environment, with probability ϵ it repicks the action at random with equal probabilities and then behaves like the old environment with the new random action

summary: pp.115, the Monte Carlo methods presented in this chapter learn value functions and optimal policies from experience in the form of sample episodes, this gives them at least three kinds of advantages over DP methods

1. they can be used to learn optimal behavior directly from interaction with the environment, with no model of the environment's dynamics
2. they can be used with simulation or sample models, for surprisingly many applications it is easy to simulate sample episodes even though it is difficult to construct the kind of explicit model of transition probabilities required by DP methods
3. it is easy and efficient to focus Monte Carlo methods on a small subset of the states, a region of special interest can be accurately evaluated without going to the expense of accurately evaluating the rest of the state set

a fourth advantage of Monte Carlo methods, which we discuss later in the book, is that they may be less harmed by violations of the Markov property, this is because they do not update their value estimates on the basis of the value estimates of successor states, in other words it is because they do not bootstrap
pp.116, maintaining sufficient exploration is an issue in Monte Carlo control methods, there are several approaches to address it

exploring starts: assume that episodes begin with state-action pairs randomly selected to cover all possibilities, can sometimes be arranged in applications with simulated episodes but are unlikely in learning from real experience

on-policy: the agent commits to always exploring and tries to find the best policy that still explores

off-policy: the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed

pp.116, the Monte Carlo methods treated in this chapter differ from the DP methods treated in the previous chapter in two major ways

- they operate on sample experience, and thus can be used for direct learning without a model

- they do not bootstrap, that is, they do not update their value estimates on the basis of other value estimates

these two differences are not tightly linked and can be separated

2.3 Sutton & Barto Reinforcement Learning Chapter 6

pp.119, temporal-difference (TD) learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas

- like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics
- like DP, TD methods update estimates based in part on other learned estimates without waiting for a final outcome (they bootstrap)

TD prediction: pp.120, whereas Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is the return 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 (where $\delta(t)$ is called the TD error)

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] = V(S_t) + \delta(t)$$

in effect the target for the Monte Carlo update is G_t whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$, this TD method is called TD(0) or one-step TD, because it is a special case of the TD(λ) and n-step TD methods developed in Chapter 12 and Chapter 7, because TD(0) bases its update in part on an existing estimate, we say that it is a bootstrapping method like DP, the TD target is an estimate for both reasons: it samples the expected value and it uses the current estimate V instead of the true v_π, thus TD methods combine the sampling of Monte Carlo with the bootstrapping of DP
pp.121, we refer to TD and Monte Carlo updates as sample updates because they involve looking ahead to a sample successor state (or state-action pair), sample updates differ from the expected updates of DP methods in that they are based on a single sample successor rather than on a complete distribution of all possible successors
pp.121, note that if the array V does not change during the episode (as it does not in Monte Carlo methods), then the Monte Carlo error can be written as a sum of TD errors

$$G_t - V(S_t) = R_{t+1} + \gamma G_{t+1} - V(S_t) + (\gamma V(S_{t+1}) - \gamma V(S_{t+1})) = \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k$$

this identity is not exact if V is updated during the episode (as it is in TD(0)), but if the step size is small then it may still hold approximately

advantages of TD prediction methods: pp.124,

- TD methods have an advantage over DP methods in that they do not require a model of the environment, of its reward and next-state probability distributions

- the next most obvious advantage of TD methods over Monte Carlo methods is that they are naturally implemented in an online fully incremental fashion, some applications have very long episodes so that delaying all learning until the end of the episode is too slow
- some Monte Carlo methods must ignore or discount episodes on which experimental actions are taken which can greatly slow learning, TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken

pp.124, for any fixed policy π, TD(0) has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

at the current time this is an open question in the sense that no one has been able to prove mathematically that one method (TD vs. Monte Carlo) converges faster than the other, however in practice TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks

optimality of TD(0): pp.126, suppose there is available only a finite amount of experience, in this case a common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer, given an approximate value function V, the increments $\alpha\,[G_t - \gamma V(S_t)]$ are computed for every time step t at which a nonterminal state is visited, but the value function is changed only once—by the sum of all the increments, then all the available experience is processed again with the new value function to produce a new overall increment and so on, until the value function converges, we call this *batch updating* because updates are made only after processing each complete batch of training data
pp.128, batch Monte Carlo methods always find the estimates that minimize mean square error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process, in general the maximum-likelihood estimate of a parameter is the parameter value whose probability of generating the data is greatest, in general batch TD(0) converges to the certainty-equivalence estimate—equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated, this helps explain why TD methods converge more quickly than Monte Carlo methods—in batch form TD(0) is faster than Monte Carlo methods because it computes the true certainty-equivalence estimate, the relationship to the certainty-equivalence estimate may also explain in part the speed advantage of nonbatch TD(0)—although the nonbatch methods do not achieve either the certainty-equivalence or the minimum squared error estimates, they can be understood as moving roughly in these directions, at the current time nothing more definite can be said about the relative efficiency of online TD and Monte Carlo methods
pp.128, although the certainty-equivalence estimate is in some sense an optimal solution, it is almost never feasible to compute it directly, if $n = |\mathcal{S}|$ is the number of states, then just forming the maximum-likelihood estimate of the process may require on the order of n^2 memory, and computing the corresponding value function requires on the order of n^3

computational steps if done conventionally, on tasks with large state spaces TD methods may be the only feasible way of approximating the certainty-equivalence solution

Sarsa: on-policy TD control: pp.129, the first step is to learn an action-value function rather than a state-value function, in particular for an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a, now we consider transitions from state-action pair to state-action pair and learn the values of state-action pairs, the theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values

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

this update is done after every transition from a nonterminal state S_t, if S_{t+1} is terminal then $Q(S_{t+1}, A_{t+1})$ is defined as zero, this rule uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ that make up a transition from one state-action pair to the next, this quintuple gives rise to the name *Sarsa* for the algorithm
pp.129, as in all on-policy methods, we continually estimate q_π for the behavior policy π and at the same time change π toward greediness with respect to q_π, the convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q, for example one could use ϵ-greedy or ϵ-soft policies, Sarsa converges with probability 1 to an optimal policy and action-value function under the usual conditions on the step sizes

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy
pp.130, if a policy was ever found that caused the agent to stay in the same state, then the next episode would never end, online learning methods such as Sarsa do not have this problem, because they quickly learn during the episode that such policies are poor and switch to something else

Q-learning: off-policy TD control: pp.131, it is defined by

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

in this case the learned action-value function Q directly approximates q_*, the optimal action-value function, independent of the policy being followed, this dramatically simplifies the analysis of the algorithm and enabled early convergence proofs, the policy still has an effect in that it determines which state-action pairs are visited and updated, however all that is required for correct convergence is that all pairs continue to be updated, under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, Q has been shown to converge with probability 1 to q_*

expected Sarsa: pp.133, 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, but that otherwise follows the schema of Q-learning

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) | S_{t+1}] - Q(S_t, A_t)\right] \\ &= Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\right] \end{aligned}$$

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
pp.133, expected Sarsa is more complex computationally than Sarsa, but in return it eliminates the variance due to the random selection of A_{t+1}, given the same amount of experience we might expect it to perform slightly better than Sarsa, and indeed it generally does, in cliff walking the state transitions are all deterministic and all randomness comes from the policy, in such cases expected Sarsa can safely set $\alpha = 1$ without suffering any degradation of asymptotic performance, whereas Sarsa can only perform well in the long run at a small value of α at which short-term performance is poor
pp.134, in these cliff walking results expected Sarsa was used on-policy, but in general it might use a policy different from the target policy π to generate behavior, in which case it becomes an off-policy algorithm, for example suppose π is the greedy policy while behavior is more exploratory, then expected Sarsa is exactly Q-learning, in this sense expected Sarsa subsumes and generalizes Q-learning while reliably improving over Sarsa, except for the small additional computational cost, expected Sarsa may completely dominate both of the other more-well-known TD control algorithms

maximization bias and double learning: pp.134, all the control algorithms that we have discussed so far involve maximization in the construction of their target policies, in these algorithms a maximum over estimated values is used implicitly as an estimate of the maximum value which can lead to a significant positive bias, we call this *maximization bias*
pp.135, one way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value, to cope with it we can divide the plays in two sets and used them to learn two independent estimates called them $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$ for all a $\in \mathcal{A}$, we could then use one estimate say Q_1 to determine the maximizing action $A^* = \arg\max_a Q_1(a)$, and the other Q_2 to provide the estimate of its value $Q_2(A^*) = Q_2\left(\arg\max_a Q_1(a)\right)$, this estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$, we can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1\left(\arg\max_a Q_2(a)\right)$, this is the idea of *double learning*, note that although we learn two estimates, only one estimate is updated on each play
pp.135, the idea of double learning extends naturally to algorithms for full MDPs, e.g. the double learning algorithm analogous to Q-learning called double Q-learning, divides the time steps in two perhaps by flipping a coin on each step, if the coin comes up heads the update is

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2\left(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\right) - Q_1(S_t, A_t)\right]$$

if the coin comes up tails, then the same update is done with Q_1 and Q_2 switched so that Q_2 is updated

$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_1\left(S_{t+1}, \arg\max_a Q_2(S_{t+1}, a)\right) - Q_2(S_t, A_t)\right]$$

the behavior policy can use both action-value estimates, e.g. an ϵ-greedy policy for double Q-learning could be based on the average (or sum) of the two action-value estimates, of course there are also double versions of Sarsa and expected Sarsa

summary: pp.138, one of the two processes making up GPI drives the value function to accurately predict returns for the current policy—this is the prediction problem, the other

process drives the policy to improve locally with respect to the current value function—this is the control problem, we then first process is based on experience, a complication arises concerning maintaining sufficient exploration, we can classify TD control methods according to whether they deal with this complication by using an on-policy or off-policy approach, Sarsa is an on-policy method and Q-learning is an off-policy method, expected Sarsa is also an off-policy method as we present it here
pp.138, the methods presented in this chapter are today the most widely used reinforcement learning methods, this is probably due to their great simplicity

- they can be applied online to experience generated from interaction with an environment with a minimal amount of computation
- they can be expressed nearly completely by single equations that can be implemented with small computer programs

pp.138, the special cases of TD methods introduced in the present chapter should rightly be called one-step, tabular, model-free TD methods, in the next two chapters and then in the second part of the book we extend them to

- n-step forms (a link to Monte Carlo methods)
- forms that include a model of the environment (a link to planning and dynamic programming)
- forms of function approximation rather than tables (a link to deep learning and artificial neural networks)

2.4 Sutton & Barto Reinforcement Learning Chapter 8

pp.159, in this chapter we develop a unified view of reinforcement learning methods that require a model of the environment such as dynamic programming and heuristic search, and methods that can be used without a model such as Monte Carlo and temporal-difference methods, these are respectively called model-based (rely on planning as their primary component) and model-free reinforcement learning methods (rely on learning)

models and planning: pp.159, by a model of the environment we mean anything that an agent can use to predict how the environment will respond to its actions, 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*, the kind of model assumed in dynamic programming—estimates of the MDP's dynamics $p(s', r|s, a)$ is a distribution model, distribution models are stronger than sample models in that they can always be used to produce samples, however in many applications it is much easier to obtain sample models than distribution models
pp.160, models can be used to mimic or simulate experience, 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
pp.160, in artificial intelligence there are two distinct approaches to planning according to our definition

state-space planning: a search through the state space for an optimal policy or an optimal path to a goal, which includes the approach we take in this book

plan-space planning: a search through the space of plans, operators transform one plan into another, and value functions are defined over the space of plans, plan-space planning includes evolutionary methods and partial-order planning—a common kind of planning in artificial intelligence in which the ordering of steps is not completely determined at all stages of planning, plan-space methods are difficult to apply efficiently to the stochastic sequential decision problems that are the focus in reinforcement learning, and we do not consider them further

pp.160, the unified view we present in this chapter is that all state-space planning methods share a common structure—all state-space planning methods

- involve computing value functions as a key intermediate step toward improving the policy
- compute value functions by updates or backup operations applied to simulated experience

$$\text{model} \longrightarrow \text{simulated experience} \xrightarrow{\text{backups}} \text{values} \longrightarrow \text{policy}$$

dynamic programming methods clearly fit this structure, in this chapter we argue that various other state-space planning methods also fit this structure, with individual methods differing only in (1) the kinds of updates they do (2) the order in which they do them (3) in how long the backed-up information is retained
pp.161, the heart of both learning and planning methods is the estimation of value functions by backing-up update operations, the difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment, the common structure means that many ideas and algorithms can be transferred between planning and learning
in addition to the unified view of planning and learning methods, a second theme in this chapter is the benefits of planning in small incremental steps, this enables planning to be interrupted or redirected at any time with little wasted computation, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model

Dyna: integrated planning, acting, and learning: pp.162, if decision making and model learning are both computation-intensive processes, then the available computational resources may need to be divided between them, to begin exploring these issues, in this section we present dyna-Q, a simple architecture integrating the major functions needed in an online planning agent
pp.162, within a planning agent there are at least two roles for real experience

model learning: or indirect reinforcement learning, improve the model (to make it more accurately match the real environment)

direct reinforcement learning: directly improve the value function and policy using the kinds of reinforcement learning methods

pp.162, both direct and indirect methods have advantages and disadvantages

- indirect methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environment interactions
- direct methods are much simpler and are not affected by biases in the design of the model

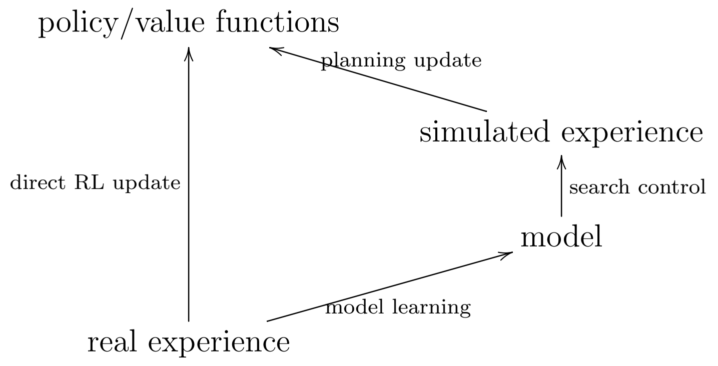
some have argued that indirect methods are always superior to direct ones, while others have argued that direct methods are responsible for most human and animal learning, related debates in psychology and artificial intelligence concern the relative importance of cognition as opposed to trial-and-error learning, and of deliberative planning as opposed to reactive decision making
pp.162, Dyna-Q includes all of the processes—planning, acting, model-learning, and direct RL—all occurring continually

planning: the random-sample one-step tabular Q-planning method, which randomly samples only from state-action pairs that have previously been experienced, so the model is never queried with a pair about which it has no information

model-learning: table-based and assumes the environment is deterministic—the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow, thus if the model is queried with a state-action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction

direct RL: one-step tabular Q-learning

pp.163, the overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown below



where 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, typically as in Dyna-Q the same reinforcement learning method is used both for learning from real experience and for planning from simulated experience, learning and planning are deeply integrated in the sense that they share almost all the same machinery differing only in the source of their experience
pp.163, conceptually planning, acting, model-learning, and direct RL occur simultaneously and in parallel in Dyna agents, however for concreteness and implementation on a serial computer we fully specify the order in which they occur within a time step, in Dyna-Q the acting, model-learning, and direct RL processes require little computation, and we assume they consume just a fraction of the time, the remaining time in each step can be devoted to the planning process, which is inherently computation-intensive

pp.165, the reason why the planning agents found the solution so much faster than the nonplanning agent is that, without planning (the $n = 0$ agent is a nonplanning agent, using only direct reinforcement learning—one-step tabular Q-learning) each episode adds only one additional step to the policy, and so only one step (the last) has been learned so far, with planning again only one step is learned during the first episode, but here during the second episode an extensive policy has been developed that by the end of the episode will reach almost back to the start state, this policy is built by the planning process while the agent is still wandering near the state state

when the model is wrong: pp.166, in some cases the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error, this tends to happen when the model is optimistic in the sense of predicting greater reward or better state transitions than are actually possible, greater difficulties arise when the environment changes to become better than it was before, and yet the formerly correct policy does not reveal the improvement, in these cases the modeling error may not be detected for a long time if ever pp.167, the general problem here is another version of the conflict between exploration and exploitation—we want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded, as in the earlier exploration/exploitation conflict, there probably is no solution that is both perfect and practical, but simple heuristics are often effective, the Dyna-Q+ agent that did solve the shortcut maze uses one such heuristic, to encourage behavior that tests long-untried actions, a special "bonus reward" is given on simulated experiences involving these actions, in particular if the modeled reward for a transition is r and the transition has not been tried in τ time steps, then planning updates are done as if that transition produced a reward of $r + \kappa\sqrt{\tau}$ for some small κ (Exercise 8.4 alternatively we can select the action that maximizes $Q(S_t, a) + \kappa\sqrt{\tau(S_t, a)}$), of course all this testing has its cost, but in many cases as in the shortcut maze, this kind of computational curiosity is well worth the extra exploration

prioritized sweeping: pp.168, a uniform selection is usually not the best, planning can be much more efficient if simulated transitions and updates are focused on particular state-action pairs; pp.169, in the much larger problems that are our real objective, the number of states is so large that an unfocused search would be extremely inefficient, in general 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

pp.169, the predecessor pairs of those that have changed a lot are more likely to also change a lot, in a stochastic environment variations in estimated transition probabilities also contribute to variations in the sizes of changes and in the urgency with which pairs need to be updated, it is natural to prioritize the updates according to a measure of their urgency and perform them in order of priority, this is the idea behind *prioritized sweeping*, a queue is maintained of every state-action pair whose estimated value would change nontrivially if updated, prioritized by the size of the change—if the effect is greater than some small threshold then the pair is inserted in the queue with the new priority, in this way the effects of changes are efficiently propagated backward until quiescence

pp.170, extensions of prioritized sweeping to stochastic environments are straightforward, the model is maintained by keeping counts of the number of times each state-action pair has been experienced and of what the next states were, it is natural then to update each pair not with a sample update but with an expected update, taking into account all possible

next states and their probabilities of occurring
pp.170, one of prioritized sweeping's limitations is that it uses expected updates, which in stochastic environments may waste lots of computation on low-probability transitions, sample updates can win because they break the overall backing-up computation into smaller pieces—those corresponding to individual transitions—which then enables it to be focused more narrowly on the pieces that will have the largest impact, this idea was taken to what may be its logical limit—update along a single transition, like a sample update but based on the probability of the transition without sampling, as in an expected update, by selecting the order in which small updates are done it is possible to greatly improve planning efficiency beyond that possible with prioritized sweeping
pp.171, in this section we have emphasized backward focusing, but this is just one strategy, for example another would be to focus on states according to how easily they can be reached from the states that are visited frequently under the current policy, which might be called *forward focusing*

expected vs. sample updates: pp.172, focusing for the moment on one-step updates, they vary primarily along three binary dimensions

1. whether they update state values or action values
2. whether they estimate the value for the optimal policy or for an arbitrary given policy
3. whether the updates are expected updates considering all possible events that might happen, or sample updates considering single sample of what might happen

the first two dimensions give rise to four classes of updates for approximating the four value functions q_*, v_*, q_π, v_π, these three binary dimensions give rise to eight cases, e.g. the Dyna-Q agents use q_* sample updates, and the Dyna-AC system uses v_π sample updates, for stochastic problems prioritized sweeping is always done using one of the expected updates; pp.173, the expected update for a state-action pair (s, a) for a model in the form of estimated dynamics $\hat{p}(s', r|s, a)$ is

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s', r|s, a) \left[r + \gamma \max_{a'} Q(s', a')\right]$$

the corresponding sample update for (s, a) is

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma \max_{a'} Q(s', a') - Q(s,a)\right]$$

pp.174, by causing estimates to be more accurate sooner, sample updates will have a second advantage (besides cheaper computation) in that the values backed up from the successor states will be more accurate, these results suggest that sample updates are likely to be superior to expected updates on problems with large stochastic branching factors (i.e. the number of possible next states s' for which $\hat{p}(s'|s, a) > 0$) and too many states to be solved exactly

summary of the chapter: pp.188, any of the learning methods can be converted into planning methods simply by applying them to simulated (model-generated) experience rather than to real experience, they are possibly identical algorithms operating on two different sources of experience

summary of Part I: dimensions: pp.189, all of the methods we have explored so far in this book have three key ideas in common

- they all seek to estimate value functions
- they all operate by backing up values along actual or possible state trajectories
- they all follow the general strategy of generalized policy iteration (GPI), meaning that they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other

pp.190, two of the most important dimensions along which the methods vary are

width of update: ranging from sample updated TD to expectation updated dynamic programming

depth of update: i.e. the degree of bootstrapping, ranging from one-step TD to full-return Monte Carlo

pp.191, the most important dimension not mentioned here and not covered in Part I of this book is that of function approximation, this dimension is explored in Part II

2.5 Littman PhD Dissertation Chapter 2

introduction: pp.26, the problem addressed is, given a complete and correct model of the environment dynamics and a goal structure, find an optimal way to behave, because we are interested in stochastic domains, we must depart from the traditional model and compute solutions in the form of policies instead of action sequences

Markov decision processes: pp.26, an MDP is a model of an agent interacting synchronously with its environment, in the MDP framework it is assumed that although there may be a great deal of uncertainty about the effects of an agent's actions, there is never any uncertainty about the agent's current state, it has complete and perfect perceptual abilities

basic framework: pp.27, a Markov decision process is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, T, R, \beta \rangle$ where

- $\mathcal{S}$ = a finite set of states of the environment
- $\mathcal{A}$ = a finite set of actions
- $T : \mathcal{S} \times \mathcal{A} \to \prod(\mathcal{S})$ is the state transition function, $T(s, a, s')$ is the probability of ending in state s' given that the agent starts in state s and takes action a
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, $R(s, a)$ is the expected reward for taking action a in state s
- $0 < \beta < 1$ is a discount factor

acting optimally: I focus on algorithms for the discounted infinite-horizon case with some attention to the simpler finite-horizon case, a policy is a description of the behavior of an agent, I consider two important kinds of policies

stationary: $\pi : \mathcal{S} \to \mathcal{A}$, the choice of action depends only on the state and is independent of the time step

non-stationary: $\delta = \langle \pi_t, \ldots, \pi_1 \rangle$, a sequence of state-action mappings indexed by time where π_t is used to choose the action on the tth-to-last step as a function of the current state

pp.27, in the finite-horizon model there is rarely a stationary optimal policy, since the way an agent chooses its actions on the final step is generally different than the way it chooses them when it has a large number of steps left, by contrast in the discounted infinite-horizon model the quantity $1-\beta$ can be viewed as the probability that the agent will cease to accrue additional reward, therefore it is as if the agent always has a constant expected number of steps remaining: $1/(1-\beta)$, because the expected distance to the horizon never changes, there is no reason to change action strategies as a function of time, there is a stationary optimal policy
pp.28, in the finite-horizon case let $\delta_k = \langle \pi_k, \ldots, \pi_1 \rangle$ be a k-step non-stationary policy and let $V_t^{\delta}(s)$ be the expected future reward starting in state s and executing non-stationary policy δ for t steps, the value of the final step is the immediate reward $V_1^{\delta_t}(s) = R(s, \pi_1(s))$, for $t > 1$ we can define $V_t^{\delta_t}(s)$ inductively by

$$V_t^{\delta_t}(s) = R\left(s, \pi_t(s)\right) + \beta \sum_{s' \in \mathcal{S}} T\left(s, \pi_t(s), s'\right) V_{t-1}^{\delta_t}(s')$$

let $V^{\pi}(s)$ be the expected discounted future reward for starting in state s and executing stationary policy π indefinitely, the infinite-horizon value is recursively defined by

$$V^{\pi}(s) = R\left(s, \pi(s)\right) + \beta \sum_{s' \in \mathcal{S}} T\left(s, \pi(s), s'\right) V^{\pi}(s')$$

the value function for policy π is the unique solution of this set of simultaneous linear equation one for each state s pp.28, given any value function V the *greedy policy* with respect to that value function π_V is defined as

$$\pi_V(s) = \arg\max_a \left[R(s,a) + \beta \sum_{s' \in \mathcal{S}} T(s,a,s') V(s') \right]$$

pp.29, it is not hard to find the optimal finite-horizon policy for a given MDP $\delta_k^* = \langle \pi_k^*, \ldots, \pi_1^* \rangle$, for $\leq t \leq k$ we can define π_t^* as follows: on the final step the agent should maximize its immediate reward

$$\pi_1^*(s) = \arg\max_a R(s,a)$$

we can define π_t^* in terms of the value function for the optimal $(t-1)$-step policy $V_{t-1}^{\delta_k^*}$ (written for simplicity as V_{t-1})

$$\pi_t^*(s) = \arg\max_a \left[R(s,a) + \beta \sum_{s' \in \mathcal{S}} T(s,a,s') V_{t-1}(s') \right]$$

it need not be unique
pp.29, in the discounted infinite-horizon case, Howard [68] showed that there exists a stationary policy π^* that is optimal for every starting state, the value function for this policy (written V^*) is defined by the set of equations

$$V^*(s) = \max_a \left[R(s,a) + \beta \sum_{s' \in \mathcal{S}} T(s,a,s') V^*(s') \right]$$

and any greedy policy with respect to this value function is optimal [126], the presence of the maximization operator in the equation means the system of equations is not linear

algorithms for solving Markov decision processes: pp.30, there are many methods for finding optimal policies for MDPs, all of these can be found in Puterman's textbook [126]

value iteration: value iteration proceeds by computing the sequence V_t of discounted finite-horizon optimal value functions, it makes use of an auxiliary function $Q_t(s, a)$ which are also known as Q functions, the algorithm terminates when the maximum difference between two successive value functions, the *Bellman error magnitude*, is less than some predetermined ϵ, theoretically it can be shown that there exists a t^*, polynomial in $|\mathcal{S}|$, $|\mathcal{A}|$, $\log \max_{s,a} |R(s, a)|$, and $1/(1 - \beta)$ such that the greedy policy with respect to V_{t^*} is an optimal infinite-horizon policy [162], rather than running value iteration for that long, we can use the Bellman error magnitude to decide when our current value function is good enough to generate a near-optimal greedy policy

$$\max_{s \in \mathcal{S}} |V^{\pi_{V_t}}(s) - V^*(s)| \leq 2\epsilon \frac{\beta}{1 - \beta}$$

policy iteration: pp.31, define a sequence of infinite-horizon non-stationary policies where $\delta_t = \langle \pi_t, \ldots, \pi_1, \pi_0, \pi_0, \pi_0, \ldots \rangle$, we can then view V_t as the infinite-horizon value function corresponding to δ_t, which shares the convergence properties of the value iteration algorithm, first $V^*(s) \geq V_t(s)$ for all s, a more useful fact is that by adopting the state-action mapping π_t as a stationary infinite-horizon policy, an agent is guaranteed total expected reward that is no worse than it would obtain following δ_t, that is $V^{\pi_t}(s) \geq V^t(s)$ for all $s \in \mathcal{S}$
combining these insights leads to an elegant approach to solving MDPs due to Howard [68], like value iteration, policy iteration computes a sequence of value functions V_t, however in policy iteration each value function V_t is the value function for the greedy policy with respect to the previous value function V_{t-1}, Puterman [126] shows that the sequence of value functions produced in policy iteration converges to V^* no more slowly than the value functions produced in value iteration, policy, however policy iteration can converge in fewer iterations than value iteration, the increased speed of convergence of policy iteration can be more than offset by the increased computation per iteration
pp.32, another fundamental difference between value iteration and policy iteration is the stopping criterion, whereas value iteration can converge to the optimal value function very gradually, policy iteration proceeds in discrete jumps, in particular each value function generated in policy iteration is associated with a particular policy (of which there are $|\mathcal{A}|^{|\mathcal{S}|}$) and each value function V_t is strictly closer to V^* than is V_{t-1}, putting these facts together tells us that policy iteration requires at most $|\mathcal{A}|^{|\mathcal{S}|}$ iterations to generate an optimal value function

linear programming: pp.33, the V^* equations can be expressed in the more descriptive mathematical language of *linear programming*, a linear program consists of a set variables, a set of linear inequalities over these variables, and a linear objective function
an important fact from the theory of linear programming is that, every linear program has an equivalent linear program in which the roles of the variables and the constraints are reversed, the resulting linear program, known as dual, can also be used to solve MDPs, one advantage of the dual formulation is that it makes it possible to express and incorporate additional constraints on the form of the policy found, the dual variables can be thought of as indicating the amount of "policy flow" through state s'

that exists via action a, under this interpretation the constraints are flow-conservation constraints—the total flow exiting state s' is equal to the flow beginning at state s' (always 1) plus the flow entering state s' via all possible combinations of states and actions weighted by their probability, the objective then is to maximize the value of the flow

other methods: pp.34, a different approach is illustrated in *modified policy iteration* which has the basic form of policy iteration with the difference that a successive-approximation algorithm (value iteration with the policy held fixed) is used to find an approximation to the value function for policy π_t, the connection between value iteration for MDPs and successive approximation for evaluating stationary policies is explored in somewhat more detail in Chapter 3

algorithms for deterministic MDPs: pp.35, a deterministic MDP is one in which $T(s, a, s')$ is either 0 or 1 for all $s, s' \in \mathcal{S}$, $a \in \mathcal{A}$, the notation $N(s, a)$ represents the unique next state resulting from taking action a from state s in a deterministic MDP, in this section I present two algorithms for this problem—one runs efficiently in parallel, and the other links the deterministic MDP problem to a general class of shortest-path problems which results in an efficient sequential algorithm

- Papadimitriou and Tsitsiklis [116] give a dynamic-programming algorithm for solving deterministic MDPs efficiently on a parallel machine, a sequential version of their algorithm runs in $|\mathcal{S}|^2 + |\mathcal{S}||\mathcal{A}| + 2|\mathcal{S}|^4$ time
- deterministic MDPs can be cast in the closed semiring framework and then solved in polynomial time using a generic algorithm for solving closed semiring problems, Cormen, Leiserson and Rivest [40] present a generic algorithm for solving path problems on closed semirings, the algorithm can find the optimal value function for a deterministic MDP in $|\mathcal{S}|^2 + |\mathcal{S}||\mathcal{A}| + |\mathcal{S}|^3$ time

pp.36, there are a few things to note about this new closed-semiring-based algorithm

- its sequential run time is an improvement over the algorithm given by Papadimitriou and Tsitsiklis which was designed to prove that the problem is in NC
- casting the problem in a more general framework helps highlight the similarities between deterministic MDPs and general path-related problems
- any advances in the area of algorithms for closed semirings immediately translate into advances for deterministic MDPs

algorithmic analysis: pp.36, we use B to designate the maximum number of bits needed to represent any numerator or denominator of β or one of the components of T or R

linear programming: pp.36, the linear program has $|\mathcal{S}||\mathcal{A}|$ constraints and $|\mathcal{S}|$ variables, there are algorithms for solving rational linear programs that taken time polynomial in the number of variables and constraints as well as the number of bits used to represent the coefficients, thus MDPs can be solved in time polynomial in $|\mathcal{S}|$, $|\mathcal{A}|$, and B
pp.37, the fact that the optimal value function for an MDP can be expressed as the solution to a polynomial-size linear program has several important implications

1. it provides a theoretically efficient way of solving MDPs
2. it provides a practical method for solving MDPs using commercial-grade implementations

3. it puts a convenient bound on the complexity of the optimal value function

value iteration: pp.38, in the general case each iteration takes $|\mathcal{A}||\mathcal{S}|^2$ steps
lemma 2.1 the number of iterations required by value iteration to reach an optimal policy is bounded above by a polynomial in $|\mathcal{S}|$, $|\mathcal{A}|$, B, and $1/(1-\beta)$
lemma 2.2 the number of iterations required by value iteration to reach an optimal policy is bounded below by $1/(1-\beta)\log(1/(1-\beta))$ (grow linearly), thus value iteration is not a polynomial-time algorithm in general

policy iteration: pp.40, since there are $|\mathcal{A}|^{|\mathcal{S}|}$ distinct policies and each iteration of policy iteration strictly improves the approximation [126], it is obvious that policy iteration terminates in at most an exponential number of steps, each step of policy iteration consists of a value-iteration-like policy-improvement step, which can be performed in $O(|\mathcal{A}||\mathcal{S}|^2)$ arithmetic operations, and a policy-evaluation step, which can be performed in $O(|\mathcal{S}|^3)$ operations by solving a system of linear equations
pp.41, while direct complexity analyses of policy iteration have been scarce, several researchers have examined a simplified family of variations of policy iteration

sequential improvement policy iteration: pp.41, whereas standard policy iteration defines the policy on step t to be $\pi_t(s) = \arg\max_a Q_t(s,a)$, sequential improvement policy iteration defines $\pi_t(s) = \pi_{t-1}(s)$ for all but one state, to be more precise, from the set of states s for which $Q_t(s, \pi_{t-1}(s)) < \max_a Q_t(s,a)$ i.e. the previous action choice for s is no longer maximal, one is selected and $\pi_t(s)$ is set to $\arg\max_a Q_t(s,a)$
pp.42, simple policy iteration requires an exponential number of iterations to generate an optimal solution to the family of MDPs in figure 2.3, although this example was constructed to hold for an undiscounted all-policies-proper criterion, it also holds under the discounted criterion regardless of discount rate, this is because the introduction of β scales down the values of the succeeding states equally, but does not change the relative order of the two choices

parallel improvement policy iteration: pp.42, when the policy is improved at all states in parallel as in standard policy iteration, the algorithm no longer has a direct simplex analogue, it is an open question whether this can lead to exponential run time in the worst case or whether the resulting algorithm is guaranteed to converge in polynomial time, however this version is strictly more efficient than the simple policy iteration algorithm mentioned above
pp.43, Puterman [126] showed that $V^{\pi_t}(s) \geq V_t(s)$ for all $s \in \mathcal{S}$ and therefore that policy iteration converges no more slowly than value iteration for discounted infinite-horizon MDPs, when combined with lemma 2.1 that bounds the time needed for value iteration to find an optimal policy, this shows that policy iteration takes polynomial time for a fixed discount factor, furthermore if the discount factor is included as part of the input as a rational number with the denominator written in unary, policy iteration takes polynomial time, this makes policy iteration a pseudopolynomial-time algorithm

summary: pp.43, to solve families of MDPs with a fixed discount factor, value iteration, policy iteration, and linear programming take polynomial time, while simply policy iteration can take exponential time
when the discount factor is included as part of the input, value iteration takes pseudopolynomial time, policy iteration takes no more than pseudopolynomial time,

and linear programming takes polynomial time, no strongly polynomial-time algorithm is known

complexity results: pp.44, at this time there is no algorithm that solves general MDPs in a number of arithmetic operations polynomial in $|\mathcal{S}|$ and $|\mathcal{A}|$ i.e. no known algorithm is strongly polynomial, however using linear programming the problem can be solved in a number of operations polynomial in $|\mathcal{S}|$, $|\mathcal{A}|$, and B, where B measures the number of bits needed to write down the transitions, rewards, and discount factor
pp.44, Papadimitriou and Tsitsiklis [116] showed that under discounted, average-reward, and polynomial-horizon criteria, the problem is P-complete, this means that although it is solvable in polynomial time, if an efficient parallel algorithm were available, then all problems in P would be solvable efficiently in parallel, an outcome considered likely by researchers in the field, since the linear-programming problem is also P-complete, this result implies that in terms of parallelizability MDPs and linear programs are equivalent—a fast parallel algorithm for one would yield a fast parallel algorithm for the other, it is not known whether the two problems are equivalent with respect to strong polynomiality—although it is clear that a strongly polynomial algorithm for solving linear programs would yield one for MDPs, the converse is still open

reinforcement learning in MDPs: pp.44, in the previous sections I explained how to find the optimal policy for an MDP given a complete description of its states, actions, rewards, and transitions, now I describe two reinforcement-learning algorithms for finding optimal policies from experience

Q-learning: pp.45, Q-learning can be viewed as a sampled asynchronous method for estimating the optimal state-action values or Q function for an unknown MDP, the entry $Q[s, a]$ is an estimate for the corresponding component of the optimal Q function defined by

$$Q^*(s, a) = R(s, a) + \beta \sum_{s'} T(s, a, s')V^*(s')$$

where V^* is the optimal value function
pp.45, the Q function is an ideal data structure for reinforcement learning, recall that there are three fundamental functions in the value-iteration algorithm—the value function V, the Q function Q, and the policy π, without access to T and R only the Q function can be used to reconstruct the other two $V(s) = \max_a Q(s, a)$ and $\pi(s) = \arg\max_a Q(s, a)$
pp.45, given an experience tuple $\langle s, a, r, s'\rangle$ the Q-learning rule is

$$Q[s, a] := (1 - \alpha)Q[s, a] + \alpha \left(r + \beta \max_{a'} Q[s', a']\right)$$

the learning rate α blends our present estimate with our previous estimates to produce a best guess at $Q(s, a)$, it needs to be decreased slowly for the Q values to converge to Q^*

model-based reinforcement learning: pp.46, although it guaranteed to find optimal policies eventually and uses very little computation time per experience, Q-learning makes extremely inefficient use of the data it gathers, it often requires a great deal of experience to achieve good performance, in model-base reinforcement learning, a

model of the environment is unknown in advance, but is learned from experience
pp.46, in the *certainty-equivalence* approach, an optimal policy for the estimated model is found at each step, this makes maximal use of the available data at the cost of high computational overhead, in the Dyna [155], prioritized-sweeping [111] and queue-dyna [119] approaches, an estimated value function is maintained and updated according to

$$V[s] := \max_a \left(\frac{R_s[s,a]}{C[s,a]} + \beta \sum_{s'} \frac{T_c[s,a,s']}{C[s,a]} V[s'] \right)$$

where $C[s,a] =$ the number of times action a has been chosen in state s, $T_c[s,a,s'] =$ the number of times this has resulted in a transition to state s', and a sum $R_s[s,a]$ of the resulting reward, because the agent has access to a model of the environment, updates can be performed at any state at any time

3 Week 3 Articles

3.1 Sutton & Barto Reinforcement Learning Chapter 7

pp.141, in this chapter we unify the Monte Carlo (MC) methods and the one-step temporal-difference (TD) methods presented in the previous two chapters, we present *n-step TD methods* that generalize both methods, 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
pp.141, in many applications one wants to be able to update the action very fast to take into account anything that has changed, but bootstrapping works best if it is over a length of time in which a significant and recognizable state change has occurred, with one-step TD methods, these time intervals are the same, and so a compromise must be made, n-step methods enable bootstrapping to occur over multiple steps freeing us from the tyranny of the single time step

n-step TD prediction: pp.142, 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 (∞-step TD), on the other hand the update of one-step TD methods or TD(0) 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, methods in which the temporal difference extends over n steps are called *n-step TD methods*
pp.143, whereas in Monte Carlo updates the target is the return

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$$

in one-step updates the target is the first reward plus the discounted estimated value of the next state which we call the one-step return

$$G_{t:t+1} := R_{t+1} + \gamma V_t(S_{t+1})$$

where $V_t : \mathcal{S} \to \mathbb{R}$ is the estimate at time t of v_π, similarly the target for an arbitrary n-step update is the n-step return

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

pp.143, the natural state-value learning algorithm for using n-step returns is thus

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

while the values of all other states remain unchanged i.e. $V_{t+n}(s) = V_{t+n-1}(s)$ for all $s \neq S_t$, we call this algorithm *n-step TD*
pp.144, an important property of n-step returns is that 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}

$$\max_s \left|\mathbb{E}_\pi\left[G_{t:t+n}|S_t = s\right] - v_\pi(s)\right| \leq \gamma^n \max_s \left|V_{t+n-1}(s) - v_\pi(s)\right| \quad \text{for all } n \geq 1$$

this is called the error reduction property of n-step returns, because of the error reduction property one can show formally that all n-step TD methods converge to the correct predictions under appropriate technical conditions

n-step Sarsa: pp.146, the n-step version of Sarsa we call n-step Sarsa, and the original version presented in the previous chapter we henceforth call one-step Sarsa or Sarsa(0), the main idea is to simply switch states for actions (state-action pairs) and then use an ϵ-greedy policy, we redefine n-step returns (update targets) in terms of estimated action values

$$G_{t:t+n} := R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \quad t+n < T$$

with $G_{t:t+n} := G_t$ if $t + n \geq T$, the natural algorithm is then

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

while the values of all other states remain unchanged i.e. $Q_{t+n}(s, a) = Q_{t+n-1}(s, a)$ for all s, a such that $s \neq S_t$ or $a \neq A_t$; pp.147, the reason why it can speed up learning compared to one-step methods is that the one-step method strengthens only the last action of the sequence of actions that lead to the high reward, whereas the n-step method strengthens the last n actions of the sequence, so that much more is learned from the one episode
pp.148, the n-step return of Sarsa can be written exactly in terms of a novel TD error as

$$G_{t:t+n} = Q_{t-1}(S_t, A_t) + \sum_{k=t}^{\min(t+n,T)-1} \gamma^{k-t}\left[R_{k+1} + \gamma Q_k(S_{k+1}, A_{k+1}) - Q_{k-1}(S_k, A_k)\right]$$

pp.148, the n-step version of expected Sarsa consists of a linear string of sample actions and states, just as in n-step Sarsa, except that its last element is a branch over all action possibilities weighted by their probability under π, this algorithm can be described by the same equation as n-step Sarsa above except with the n-step return redefined as

$$G_{t:t+n} := R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \overline{V}_{t+n-1}(S_{t+n}) \quad t+n < T$$

with $G_{t:t+n} := G_t$ if $t + n \geq T$, where $\overline{V}_t(s)$ is the expected approximate value of state s using the estimated action values at time t under the target policy

$$\overline{V}_t(s) := \sum_a \pi(a|s) Q_t(s, a) \quad \text{for all } s \in \mathcal{S}$$

if s is terminal then its expected approximate value is defined to be 0

summary: pp.157, methods that involve an intermediate amount of bootstrapping are important because they will typically perform better than either extreme
pp.157, all n-step methods involve a delay of n time steps before updating as only then are all the required future events known, a further drawback is that they involve more computation per time step than previous methods, compared to one-step methods n-step methods also require more memory to record the states, actions, rewards, and sometimes other variables, in Chapter 12 we will see how multi-step TD methods can be implemented with minimal memory and computational complexity using eligibility traces, but there will always be some additional computation beyond one-step methods

3.2 Sutton & Barto Reinforcement Learning Chapter 12

pp.287, almost any temporal-difference (TD) method such as Q-learning or Sarsa can be combined with eligibility traces to obtain a more general method that may learn more efficiently, eligibility traces unify and generalize TD and Monte Carlo methods, when TD methods are augmented with eligibility traces, they produce a family of methods spanning a spectrum that has Monte Carlo methods at one end ($\lambda = 1$) and one-step TD methods at the other ($\lambda = 0$), in between are intermediate methods that are often better than either extreme method
pp.287, of course we have already seen that the n-step TD can unify TD and Monte Carlo, what eligibility traces offer beyond it is an elegant algorithmic mechanism with significant computational advantages, the mechanism is a short-term memory vector—the *eligibility trace* $\mathbf{z}_t \in \mathbb{R}^d$—that parallels the long-term weight vctor $\mathbf{w}_t \in \mathbb{R}^d$, the primary computational advantage of eligibility traces over n-step methods is that

1. only a single trace vector is required rather than a store of the last n feature vectors
2. learning also occurs continually and uniformly in time rather than being delayed and then catching up at the end of the episode
3. learning can occur and affect behavior immediately after a state is encountered rather than being delayed n steps

pp.287, eligibility traces illustrate that a learning algorithm can sometimes be implemented in a different way to obtain computational advantages, many algorithms are most naturally formulated and understood as an update of a state's value based on events that follow that state over multiple future time steps, such formulations based on looking forward from the updated state are called forward views, forward views are always somewhat complex to implement because the update depends on later things that are not available at the time, however it is often possible to achieve nearly the same updates—and sometimes exactly the same updates—with an algorithm that uses the current TD error, looking backward to recently visited states using an eligibility trace, these alternate ways of looking at and implementing learning algorithms are called backward views
pp.288, our treatment pays special attention to the case of linear function approximation, for which the results with eligibility traces are stronger, all these results apply also to the tabular and state aggregation cases because these are special cases of linear function approximation

the λ-return: pp.289, the TD(λ) algorithm can be understood as one particular way of averaging n-step updates, this average contains all the n-step updates each weighted proportionally to λ^{n-1} and is normalized by a factor of $1-\lambda$ to ensure that the weights sum to 1, the resulting update is toward a return called the λ-return defined in its state-based form by

$$G_t^\lambda := (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n}$$

after a terminal state has been reached all subsequent n-step returns are equal to the conventional return G_t, we can separate these post-termination terms from the main sum yielding

$$G_t^\lambda := (1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{T-t-1}G_t$$

$\Rightarrow$ it is easy to verify that

$$(1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1} = 1-\lambda^{T-t-1}$$

thus for $\lambda = 1$ updating according to the λ-return is a Monte Carlo algorithm, for $\lambda = 0$ updating according to the λ-return is a one-step TD method; pp.291, the λ-return gives us an alternative way of moving smoothly between Monte Carlo and one-step TD methods that can be compared with the n-step bootstrapping way developed in Chapter 7
pp.290, the off-line λ-return algorithm, as an off-line algorithm, makes no changes to the weight vector during the episode, then at the end of the episode a whole sequence of off-line updates are made according to our usual semi-gradient rule, using the λ-return as the target

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\left[G_t^\lambda - \hat{v}(S_t,\mathbf{w}_t)\right]\nabla\hat{v}(S_t,\mathbf{w}_t) \qquad t = 0,\ldots,T-1$$

where $\hat{v}(s,\mathbf{w})$ is the approximate value of state s given weight vector $\mathbf{w}$ (Chapter 9)
pp.291, the approach that we have been taking so far is what we call the theoretical or forward view of a learning algorithm, for each state visited we look forward in time to all the future rewards and decide how best to combine them, after looking forward from and updating one state, we move on to the next and never have to work with the preceding state again, on the other hand future states are viewed and processed repeatedly once from each vantage point preceding them

TD(λ): pp.292, TD(λ) improves over the off-line λ-return algorithm in three ways

1. it updates the weight vector on every step of an episode rather than only at the end, and thus its estimates may be better sooner
2. its computations are equally distributed in time rather than all at the end of the episode
3. it can be applied to continuing problems rather than just to episodic problems

in this section we present the semi-gradient version of TD(λ) with function approximation
pp.292, with function approximation the eligibility trace is a vector $\mathbf{z}_t \in \mathbb{R}^d$ with the same number of components as the weight vector $\mathbf{w}_t$, eligibility traces assist in the learning process, their only consequence is that they affect the weight vector, in TD(λ) the eligibility

trace vector is initialized to zero at the beginning of the episode, is incremented on each time step by the value gradient, and then fades away by $\gamma\lambda$

$$\begin{aligned}\mathbf{z}_{-1} &:= 0\\ \mathbf{z}_t &:= \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t, \mathbf{w}_t) \qquad 0 \le t \le T\end{aligned}$$

the TD error for state-value prediction is

$$\delta_t := R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)$$

in TD(λ) the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t$$

pp.294, TD(λ) is oriented backward in time, at each moment we look at the current TD error and assign it backward to each prior state according to how much that state contributed to the current eligibility trace at that time, where the TD error and traces come together we get the update given above, if $\lambda = 0$ then the trace at t is exactly the value gradient corresponding to S_t, thus the TD(λ) update reduces to the one-step semi-gradient TD update treated in Chapter 9, this is why that algorithm was called TD(0), for larger values of λ but still $\lambda < 1$ more of the preceding states are updated, but each more temporally distant state is updated less because the corresponding eligibility trace is smaller, if $\lambda = 1$ then the credit given to earlier states falls only by γ per step, if $\lambda = 1$ and $\gamma = 1$ then the eligibility traces do not decay at all with time, in this case the method behaves like a Monte Carlo method for an undiscounted episodic task
pp.294, TD(1) is a way of implementing Monte Carlo algorithms that is more general than those presented earlier and that significantly increases their range of applicability, whereas the earlier Monte Carlo methods were limited to episodic tasks, TD(1) can be applied to discounted continuing tasks as well, moreover TD(1) can be performed incrementally and online, if something unusually good or bad happens during an episode, control methods based on TD(1) can learn immediately and alter their behavior on that same episode
pp.295, linear TD(λ) has been proved to converge in the on-policy case if the step-size parameter is reduced over time according to the usual conditions on the step sizes

$$\sum_{n=1}^{\infty}\alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty}\alpha_n^2(a) < \infty$$

just as discussed in Section 9.4 convergence is not to the minimum-error weight vector but to a nearby weight vector that depends on λ, the bound on solution quality presented in that section (9.14) can now be generalized to apply for any λ, for the continuing discounted case

$$\overline{\mathrm{VE}}(\mathbf{w}_\infty) \le \frac{1-\gamma\lambda}{1-\gamma}\min_{\mathbf{w}}\overline{\mathrm{VE}}(\mathbf{w})$$

that is the asymptotic error is no more than $(1-\gamma\lambda)/(1-\gamma)$ times the smallest possible error, as λ approaches 1 the bound approaches the minimum error and it is loosest at $\lambda = 0$, however in practice $\lambda = 1$ is often the poorest choice
pp.295, if the weight updates over an episode were computed on each step but not actually used to change the weights (**w** remained fixed), then the sum of TD(λ)'s weight updates would be the same as the sum of the off-line λ-return algorithm's updates

n-step truncated λ-return methods: pp.296, in the continuing case the λ-return is technically never known, as it depends on n-step returns for arbitrarily large n, and thus on rewards arbitrarily far in the future, a natural approximation then would be to truncate the sequence after some number of steps, in general we define the *truncated λ-return* for time t given data only up to some later horizon h as

$$G_{t:h}^{\lambda} := (1-\lambda)\sum_{n=1}^{h-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{h-t-1}G_{t:h} \qquad 0 \le t < h \le T$$

the horizon h is playing the same role as was previously played by T the time of termination, whereas in the λ-return there is a residual weight given to the conventional return G_t, here it is given to the longest available n-step return $G_{t:h}$
pp.296, the truncated λ-return immediately gives rise to a family of n-step λ-return algorithms similar to the n-step methods of Chapter 7, in the state-value case this family of algorithm is known as truncated TD(λ) or TTD(λ) which is defined by

$$\mathbf{w}_{t+n} := \mathbf{w}_{t+n-1} + \alpha\left[G_{t:t+n}^{\lambda} - \hat{v}(S_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{v}(S_t, \mathbf{w}_{t+n-1}) \qquad 0 \le t < T$$

this algorithm can be implemented efficiently so that per-step computation does not scale with n (though of course memory must), efficient implementation relies on the fact that the k-step λ-return can be written exactly as

$$G_{t:t+k}^{\lambda} = \hat{v}(S_t, \mathbf{w}_{t-1}) + \sum_{i=t}^{t+k-1}(\gamma\lambda)^{i-t}\delta_i'$$

where

$$\delta_t' := R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_{t-1})$$

redoing updates: online λ-return algorithm: pp.297, choosing the truncation parameter n in TTD(λ) involves a tradeoff, n should be large so that the method closely approximates the off-line λ-return algorithm, but it should also be small so that the updates can be made sooner and can influence behavior sooner, we can get the best of both albeit at the cost of computational complexity, the idea is that on each time step as you gather a new increment of data, you go back and redo all the updates since the beginning of the current episode; pp.298, that is each time the horizon is advanced, all the updates are redone starting from $\mathbf{w}_0$ using the weight vector from the preceding horizon

$$\mathbf{w}_{t+1}^{h} := \mathbf{w}_t^{h} + \alpha\left[G_{t:h}^{\lambda} - \hat{v}(S_t, \mathbf{w}_t^{h})\right]\nabla\hat{v}(S_t, \mathbf{w}_t^{h})$$

where $\mathbf{w}_t^h$ denotes the weights used to generate the value at time t in the sequence up to horizon h, the first weight vector $\mathbf{w}_0^h$ in each sequence is that inherited from the previous episode (so they are the same for all h), and the last weight vector $\mathbf{w}_h^h$ in each sequence defines the ultimate weight-vector sequence of the algorithm, this update together with $\mathbf{w}_t := \mathbf{w}_t^t$ defines the online λ-return algorithm

true online TD(λ): pp.299, there is a way to invert the forward-view online λ-return algorithm to produce an efficient backward-view algorithm using eligibility traces for the case of linear function approximation, this implementation is known as the true online TD(λ) algorithm because it is "truer" to the ideal of the online λ-return algorithm than the TD(λ) algorithm is, it turns out that among the sequence of weight vectors produced by the online λ-return algorithm, the $\mathbf{w}_t^t$ are the only ones really needed, the strategy then is to find a compact efficient way of computing each $\mathbf{w}_t^t$ from the one before, if this is done for the linear case in which $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T\mathbf{x}(s)$, then we arrive at the true online TD(λ) algorithm

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t + \alpha\left(\mathbf{w}_t^T\mathbf{x}_t - \mathbf{w}_{t-1}^T\mathbf{x}_t\right)(\mathbf{z}_t - \mathbf{x}_t)$$

where we have used the shorthand $\mathbf{x}_t := \mathbf{x}(S_t)$ and

$$\begin{aligned}\delta_t &:= R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)\\ \mathbf{z}_t &:= \gamma\lambda\mathbf{z}_{t-1} + \left(1 - \alpha\gamma\lambda\mathbf{z}_{t-1}^T\mathbf{x}_t\right)\mathbf{x}_t\end{aligned}$$

pp.300, this algorithm has been proven to produce exactly the same sequence of weight vectors $\mathbf{w}_t$, $0 \leq t \leq T$ as the online λ-return algorithm, however the algorithm is much less expensive, the memory requirements of true online TD(λ) are identical to those of conventional TD(λ), while the per-step computation is increased by about 50% (due to the eligibility-trace update), overall the per-step computational complexity remains of $O(d)$ the same as TD(λ), the eligibility trace used in true online TD(λ) is called a dutch trace to distinguish it from the trace used in TD(λ) which is called an accumulating trace, accumulating traces remain of interest for nonlinear function approximations where dutch traces are not available

Sarsa(λ): pp.303, to learn approximate action values $\hat{q}(s, a, \mathbf{w})$ rather than approximate state values $\hat{v}(s, \mathbf{w})$, we need to use the action-value form of the n-step return from Chapter 10

$$G_{t:t+n} := R_{t+1} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n\hat{q}\left(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}\right) \qquad t + n < T$$

with $G_{t:t+n} := G_t$ if $t + n \geq T$, the action-value form of the off-line λ-return algorithm simply uses $\hat{q}$ rather than $\hat{v}$

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\left[G_t^\lambda - \hat{q}(S_t, \mathbf{w}_t)\right]\nabla\hat{q}(S_t, \mathbf{w}_t) \qquad t = 0, \ldots, T-1$$

pp.303, the temporal-difference method for action values known as Sarsa(λ) approximates this forward view, it has the same update rule as given earlier for TD(λ), $\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t$, except using the action-value form of the TD error

$$\delta_t := R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)$$

and the action-value form of the eligibility trace

$$\begin{aligned}\mathbf{z}_{-1} &:= 0\\ \mathbf{z}_t &:= \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{q}(S_t, A_t, \mathbf{w}_t) \qquad 0 \leq t \leq T\end{aligned}$$

pp.305, there is also an action-value version of our ideal TD method, the online λ-return algorithm (Section 12.4) and its efficient implementation as true online TD(λ) (Section 12.5), everything in Section 12.4 goes through without change other than to use the action-value form of the n-step return given above, the analyses in Sections 12.5 and 12.6 also carry through for action values, the only change being the use of state-action feature vectors $\mathbf{x}_t = \mathbf{x}(S_t, A_t)$ instead of state feature vectors $\mathbf{x}_t = \mathbf{x}(S_t)$; pp.207, finally there is also a truncated version of Sarsa(λ) called forward Sarsa(λ), which appears to be a particularly effective model-free control method for use in conjunction with multi-layer artificial neural networks

Watkins's Q(λ) to tree-backup(λ): pp.312, several methods have been proposed over the years to extend Q-learning to eligibility traces, the original is Watkins's $Q(\lambda)$ which decays its eligibility traces in the usual way as long as a greedy action was taken, then cuts the traces to zero after the first non-greedy action
pp.312, in Chapter 7 we distinguished n-step expected Sarsa from n-step tree backup, where the latter retained the property of not using importance sampling, it remains to present the eligibility trace version of tree backup which we will call Tree-Backup(λ) or TB(λ) for short; pp.314, like all semi-gradient algorithms TB(λ) is not guaranteed to be stable when used with off-policy data and with a powerful function approximator, to obtain those assurances TB(λ) would have to be combined with one of the methods presented in the next section

stable off-policy methods with traces: pp.314, several methods using eligibility traces have been proposed that achieve guarantees of stability under off-policy training, and here we present four of the most important using this book's standard notation including general bootstrapping and discounting functions, all are based on either the gradient-TD or the emphatic-TD ideas presented in Sections 11.7 and 11.8, all the algorithms assume linear function approximation

- GTD(λ) is the eligibility-trace algorithm analogous to TDC, the better of the two state-value gradient-TD prediction algorithms discussed in Section 11.7, its goal is to learn a parameter $\mathbf{w}_t$ such that $\hat{v}(s, \mathbf{w}) := \mathbf{w}_t^T \mathbf{x}(s) \approx v_\pi(s)$ even from data that is due to following another policy b
- GQ(λ) is the gradient-TD algorithm for action values with eligibility traces, its goal is to learn a parameter $\mathbf{w}_t$ such that $\hat{q}(s, a, \mathbf{w}_t) := \mathbf{w}_t^T \mathbf{x}(s, a) \approx q_\pi(s, a)$ from off-policy data, if the target policy is ϵ-greedy or otherwise biased toward the greedy policy for $\hat{q}$, then GQ(λ) can be used as a control algorithm
- HTD(λ) is a hybrid state-value algorithm combining aspects of GTD(λ) and TD(λ), its most appealing feature is that it is a strict generalization of TD(λ) to off-policy learning, meaning that if the behavior policy happens to be the same as the target policy, then HTD(λ) becomes the same as TD(λ) which is not true for GTD(λ), this is appealing because TD(λ) is often faster than GTD(λ) when both algorithms converge, and TD(λ) requires setting only a single step size
in addition to the second set of weights $\mathbf{v}_t$ HTD(λ) also has a second set of eligibility traces $\mathbf{z}_t^b$, these are conventional accumulating eligibility traces for the behavior policy and become equal to $\mathbf{z}_t$ if all the ρ_t are 1, which causes the last term in the $\mathbf{w}_t$ update to be zero and the overall update to reduce to TD(λ)
- emphatic TD(λ) is the extension of the one-step emphatic-TD algorithm (Sections 9.11 and 11.8) to eligibility traces, the resultant algorithm retains strong off-policy convergence guarantees while enabling any degree of bootstrapping albeit at the cost of high variance and potentially slow convergence
in the on-policy case ($\rho_t = 1$ for all t) emphatic-TD(λ) is similar to conventional TD(λ) but still significantly different, in fact whereas emphatic-TD(λ) is guaranteed to converge for all state-dependent λ functions, TD(λ) is not, TD(λ) is guaranteed convergent only for all constant λ

implementation issues: pp.316, in practice implementations on conventional computers may keep track of and update only the few traces that are significantly greater than zero, using this trick the computational expense of using traces in tabular methods is typically just a few times that of a one-step method

when function approximation is used, the computational advantages of not using traces generally decrease

truncated λ-return methods (Section 12.3) can be computationally efficient on conventional computers though they always require some additional memory

conclusions: pp.317, this chapter has offered an introduction to the elegant emerging theoretical understanding of eligibility traces, one aspect of this elegant theory is true online methods, which exactly reproduce the behavior of expensive ideal methods while retaining the computational congeniality of conventional TD methods, another aspect is the possibility of derivations that automatically convert from intuitive forward-view methods to more efficient incremental backward-view algorithms

pp.317, as we mentioned in Chapter 5, Monte Carlo methods may have advantages in non-Markov tasks because they do not bootstrap, because eligibility traces make TD methods more like Monte Carlo methods, they also can have advantages in these cases, if one wants to use TD methods because of their other advantages, but the task is at least partially non-Markov, then the use of an eligibility trace method is indicated

pp.317, by adjusting λ we can place eligibility trace methods anywhere along a continuum from Monte Carlo to one-step TD methods, where shall we place them? we do not yet have a good theoretical answer to this question, on tasks with many steps per episode or many steps within the half-life of discounting, it appears significantly better to use eligibility traces than not to, on the other hand if the traces are so long as to produce a pure Monte Carlo method or nearly so, then performance degrades sharply, an intermediate mixture appears to be the best choice

pp.317, methods using eligibility traces require more computation than one-step methods, but in return they offer significantly faster learning particularly when rewards are delayed by many steps, thus it often makes sense to use eligibility traces when data are scarce and cannot be repeatedly processed, as is often the case in online applications, on the other hand in off-line applications in which data can be generated cheaply, then it often does not pay to use eligibility traces, in these cases the speedup per datum due to traces is typically not worth their computational cost, and one-step methods are favored

3.3 Littman PhD Dissertation Chapter 3

introduction: pp.50, the model I develop here, the *generalized Markov decision process* includes all the models discussed in the thesis: MDPs, alternating Markov games, Markov games, and information-state MDPs, as well as several less-studied models, the generalized MDP model applies to several different optimality objectives: finite-horizon, all-policies-proper, expected discounted reward, and risk-sensitive discounted reward, to name a few

pp.51, the main result is that all these models have a notion of an optimal value function and an optimal policy, and that a general form of the value-iteration algorithm converges to the optimal value function

pp.51, I define a version of policy iteration for generalized MDPs and show that the algorithm converges to an optimal policy, however it is only useful for a subclass of generalized MDPs that obey additional properties, for a different subclass of generalized MDPs I show that a form of Q-learning converges to the optimal Q function under the appropriate conditions, I also show that a simple model-based reinforcement-learning algorithm converges to an optimal value function for all finite-state generalized MDPs

generalized Markov decision processes: pp.51, a generalized Markove decision process is a tuple $\langle \mathcal{X}, \mathcal{U}, T, R, N, \otimes, \oplus, \beta \rangle$ where the fundamental quantities are

- a set of states $\mathcal{X}$, can be infinite
- a finite set of actions $\mathcal{U}$
- a reward function $R : \mathcal{X} \times \mathcal{U} \to \mathbb{R}$
- a transition function $T : \mathcal{X} \times \mathcal{U} \to \prod(\mathcal{X})$
- a next-state function N mapping $\mathcal{X} \times \mathcal{U}$ to finite subsets of $\mathcal{X}$, finite because $\mathcal{X}$ can be infinite
- a discount factor β
- a summary operator $\oplus$ that defines the value of transitions based on the value of the successor states
- a summary operator $\otimes$ that defines the value of a state based on the values of all state-action pairs

pp.52, the two summary operators for a MDP are

$$\overset{(s,a)}{\bigoplus_{s'}} g(s') = \sum_{s'} T(s,a,s') g(s') \qquad \overset{(s)}{\bigotimes_{a}} f(s,a) = \max_a f(s,a)$$

with which we can rewrite the equations for the optimal value function in an MDP as

$$V^*(s) = \overset{(s)}{\bigotimes_{a}} \left(R(s,a) + \beta \overset{(s,a)}{\bigoplus_{s'}} V^*(s') \right)$$

the essence of the generalized Markov decision process framework is that whenever the operators $\oplus$ and $\otimes$ satisfy certain non-expansion properties, then the above equation is a characterization of the unique optimal value function, not all possible definitions of $\oplus$ and $\otimes$ can be used to define a generalized MDPs, they must both be non-expansions for all states and state-action pairs respectively
pp.52, for a summary operator $\odot$ to be a non-expansion, it must satisfy two constraints:

- the summary of a function must lie between the largest and smallest value of the function, i.e. given functions h and h' over a finite set I we must have

$$\min_{i \in I} h(i) \le \bigodot_{i \in I} h(i) \le \max_{i \in I} h(i)$$

- the difference between the summaries of two different functions must be no larger than the largest difference between the functions, i.e. given functions h and h' over a finite set I we must have

$$\left| \bigodot_{i \in I} h(i) - \bigodot_{i \in I} h'(i) \right| \le \max_{i \in I} |h(i) - h'(i)|$$

acting optimally: pp.54, define the dynamic-programming operator H for a generalized MDP, a function that takes a value function and creates a new value function as

$$[HV](x) = \bigotimes_{u}^{(x)} \left(R(x,u) + \beta \bigoplus_{x'}^{(x,u)} V(x') \right)$$

for MDPs H is the mathematical instantiation of a single step of value iteration, we can also define a dynamic-programming operator K that acts on Q functions

$$[KQ](x,u) = R(x,u) + \beta \bigoplus_{x'}^{(x,u)} \bigotimes_{u'}^{(x')} Q(x',u')$$

pp.54, for value functions V_1 and V_2 we define

$$\|V_1 - V_2\| = \sup_x |V_1(x) - V_2(x)|$$

where $\|\cdot\|$ is a distance function known as the L_∞ norm or max norm, we can extend the notion of distance to cover Q functions as well, if Q_1 and Q_2 are Q functions

$$\|Q_1 - Q_2\| = \sup_x \max_u |Q_1(x,u) - Q_2(x,u)|$$

pp.55, the non-expansion properties of $\otimes$ and $\oplus$ lead to a convenient property of these operators with regard to distances: let Q_1 and Q_2 be Q functions and V_1 and V_2 be value functions, then

$$\left\| \bigotimes Q_1 - \bigotimes Q_2 \right\| \leq \|Q_1 - Q_2\| \qquad \left\| \bigoplus V_1 - \bigoplus V_2 \right\| \leq \|V_1 - V_2\|$$

with this distance-based bound we can prove that the H and K operators are contraction mappings if $\beta < 1$, in particular if V_1 and V_2 are value functions and Q_1 and Q_2 are Q functions then

$$\|HV_1 - HV_2\| \leq \beta \|V_1 - V_2\| \qquad \|KQ_1 - KQ_2\| \leq \beta \|Q_1 - Q_2\|$$

pp.56, a weighted max norm for value functions and for Q functions is defined by

$$\|V_1 - V_2\|_w = \sup_x |V_1(x) - V_2(x)|/w(x) \qquad \|Q_1 - Q_2\|_w = \sup_x \max_u |Q_1(x,u) - Q_2(x,u)|/w(x)$$

operator H is a contraction mapping with respect to some weighted max norm w if and only if $\|HV_1 - HV_2\|_w \leq \beta_w \|V_1 - V_2\|_w$ for some $\beta_w < 1$
pp.56, for any generalized MDP in which

$$\bigoplus_{x'}^{(x,u)} g(x') = \sum_{x'} T(x,u,x') g(x')$$

if $\beta = 1$ but all policies are guaranteed to reach a zero-reward absorbing state (the all-policies-proper case), then the H and K operators are contraction mappings with respect to some weighted max norm pp.57, for any generalized Markov decision process if $\beta < 1$ then there is a unique V^* called the optimal value function such that $V^* = HV^*$, a unique Q^* called the optimal Q function such that $Q^* = KQ^*$, and an optimal (possibly stochastic) policy π^* such that $V^*(x) = \sum_u \pi^*(x,u)Q^*(x,u)$, this is also true if $\beta = 1$, the all-policies-proper condition holds, and an expected value criterion is used, the use of the word optimal is somewhat strange since V^* need not be the largest or smallest value function in any sense, it is simply the fixed point of the dynamic-programming operator H

exploration-sensitive MDPs: pp.57, one interesting use of generalized MDPs is as a way to formalize John's [71] exploration-sensitive learning algorithm, John considered the implications of insisting that agents simultaneously act to maximize their reward and explore their environment, in John's formulation the agent is forced to adopt a policy from a restricted set, John's approach requires that the definition of optimality be changed to reflect the restriction on policies, the optimal value function is given by $V^*(x) = \sup_{\pi \in \mathcal{P}} V^\pi(x)$ where $\mathcal{P}$ is the set of permitted stationary policies, and the associated Bellman equations are

$$V^*(x) = \sup_{\pi \in \mathcal{P}} \sum_u \pi(x,u) \left(R(x,u) + \beta \sum_{x'} T(x,u,x')V^*(x') \right)$$

which corresponds toa generalized MDP model with

$$\bigoplus_{x'}^{(x,u)} g(x') = \sum_{x'} T(x,u,x')g(x') \qquad \bigotimes_{u}^{(x)} f(x,u) = \sup_{\pi \in \mathcal{P}} \sum_u \pi(x,u) f(x,u)$$

algorithms for solving generalized MDPs: pp.58,

value iteration: pp.58, let V_t be the value function produced in the tth iteration of value iteration, after t steps of value iteration on a generalized MDP $\|V_t - V^*\| \leq \beta^t \|V_0 - V^*\|$, let $M = \sup_x \max_u |R(x,u)| = \|R\|$, if the agent received a reward of M on every step, its total expected reward would be $\sum_{i=0}^{\infty} \beta^i M = M/(1-\beta)$, thus the zero value function $V_0 = 0$ cannot differ from the optimal value function by more than $M/(1-\beta)$ at any state, this also implies that the value function for any policy cannot differ from the optimal value function by more than $2M/(1-\beta)$ at any state

computing near-optimal policies: pp.59, in this section we relate arbitrary generalized MDPs to the specific generalized MDP resulting from using the summary operator $\otimes_u^{\pi,(x)} f(x,u) = \sum_u \pi(x,u) f(x,u)$ where π is some stationary probabilistic policy, let V be some value function, we define the *myopic policy* with respect to V to be any $\pi : \mathcal{X} \to \Pi(\mathcal{U})$ such that

$$[HV](x) = \sum_u \pi(x,u) \left(R(x,u) + \beta [\bigoplus V](x,u) \right)$$

the existence of such a π follows from the definition of H, it need not be unique, myopic policies are simply greedy policies generalized to models in which reward is not maximized

pp.59, for a policy π define H^π to be the dynamic-programming operator resulting from the generalized MDP that shares its state space, action space, transition function, reward function, next state function, $\oplus$ operator, and discount factor with the generalized MDP in question, but uses $\otimes^\pi$ in place of $\otimes$, because $\otimes^\pi$ is a non-expansion the new model is itself a generalized MDP, therefore we can define V^π to be the unique value function satisfying $V^\pi = H^\pi V^\pi$ which we call the *value function for policy* π, let V be a value function, V^π the value function for the myopic policy with respect to V, and V^* be the optimal value function, let ϵ be the Bellman error magnitude for V, $\epsilon = \|V - HV\|$, then

$$\|V - V^\pi\| \leq \frac{\epsilon}{1-\beta} \qquad \|V - V^*\| \leq \frac{\epsilon}{1-\beta}$$

pp.60, by the definitions of H and π, $HV = \otimes(R + \beta \oplus V) = \otimes^\pi(R + \beta \oplus V)$ and $H^\pi V^\pi = \otimes^\pi(R + \beta \oplus V^\pi)$, we can use these equations to help bound the distance between V^π and the V^* in terms of ϵ, the Bellman error magnitude: let V be a value function, V^π the value function for the myopic policy with respect to V, and V^* the optimal value function, let ϵ be the Bellman error magnitude for V, $\epsilon = \|V - HV\|$, then

$$\|HV - V^\pi\| \leq \frac{\epsilon\beta}{1-\beta} \qquad \|HV - V^*\| \leq \frac{\epsilon\beta}{1-\beta} \qquad \|V^\pi - V^*\| \leq \frac{2\epsilon\beta}{1-\beta}$$

the significance of the result is that a value-iteration algorithm that stops when the Bellman error magnitude is less than or equal to $\epsilon \geq 0$ will produce a good policy with respect to ϵ

policy iteration: pp.61, unlike value iteration, the convergence of policy iteration seems to require that value is maximized with respect to some set of possible actions, to capture this we will restrict our attention to generalized MDPs in which $\otimes$ can be written as

$$[\bigotimes Q](x) = \max_{\rho \in \mathcal{R}} [\overset{\rho}{\bigotimes} Q](x)$$

where $\mathcal{R}$ is a compact set and $\otimes^\rho$ is a non-expansion operator for all $\rho \in \mathcal{R}$, a generalized MDP satisfying the above equation and the monotonicity property discussed below is called a *maximizing* generalized MDP, the term ρ-myopic policy refers to a mapping $\omega : \mathcal{X} \to \mathcal{R}$ such that

$$[\overset{\omega(x)}{\bigotimes} Q](x) = \max_{\rho \in \mathcal{R}} [\overset{\rho}{\bigotimes} Q](x)$$

for all $x \in \mathcal{X}$, the value function for a ρ-myopic policy ω, V^ω, is defined as the optimal value function for the generalized MDP, we characterize policy iteration as follows: start with a value function V and compute its ρ-myopic policy ω and ω's value function V^ω, if $\|V - V^\omega\| \leq \epsilon$, terminate with V as an approximation of the optimal value function, otherwise start over after assigning $V := V^\omega$
pp.62, we can apply this algorithm to MDPs by taking $\mathcal{R}$ to be the set of actions and $\otimes^\rho$ to select the Q value for the action corresponding to ρ: $\max_{\rho \in \mathcal{R}} [\otimes^\rho](x) = \max_{u \in \mathcal{U}} Q(x, u)$, in Markov games we taken $\mathcal{R}$ to be the set of probability distributions over agent actions and $\otimes^\rho$ to be a minimum over opponent actions of the ρ-weighted expected Q value, as we will see in Chapter 5 computing V^ω is equivalent to solving an MDP, which is easier than finding V^* directly

algorithmic analysis: pp.62,

value iteration: pp.62, let $\mathcal{X}$ be finite, and let B be a problem-specific parameter e.g. a bound on the number of bits needed to represent components of T and R, we say that a quantity is polynomially bounded if there is some polynomial in $|\mathcal{X}|$, $|\mathcal{U}|$, B, and $\log(1/(1-\beta))$ that grows asymptotically faster than that quantity, we say a quantity is pseudopolynomially bounded if it is polynomially bounded with respect to $|\mathcal{X}|$, $|\mathcal{U}|$, B, and $1/(1-\beta)$, we know that $\otimes$ and $\oplus$ have the property that for all Q and V, $[\otimes Q](x) = \sum_u \pi(x,u)Q(x,u)$ and $[\oplus V](x,u) = \sum_{x' \in N(x,u)} \tau(x,u,x')V(x')$ for some probability distributions π and τ, if for all Q and V, π and τ can be expressed using only rational numbers with a polynomially bounded number of bits, then we say that $\otimes$ and $\oplus$ are *polynomially expressible*, if $\mathcal{X}$ is finite, the number of bits needed to express $R(x,u)$ for all x and u is polynomially bounded, and $\otimes$ and $\oplus$ are polynomially expressible, then any myopic policy with respect to V_{t^*} is optimal, for some pseudopolynomially bounded t^*

policy iteration: pp.64, for the policy-iteration algorithm above to converge to the optimal value function, we need to place an additional monotonicity condition on the associated summary operators: if $g(x') \geq g'(x')$ then $\oplus_{x'}^{(x,u)} g(x') \geq \oplus_{x'}^{(x,u)} g'(x')$ and similarly for $\otimes^\rho$, not all non-expansion operators satisfy the monotonicity condition, the policy-iteration algorithm can be stated as follows: let ω_0 be an arbitrary function mapping $\mathcal{X} \to \mathcal{R}$, at iteration t let ω_{t-1} be the ρ-myopic policy with respect to V^{ω_t}, terminate when $\|V^{\omega_t} - V^{\omega_{t-1}}\|$ is small enough
pp.65, if $\mathcal{X}$ is finite, the number of bits needed to express $R(x,u)$ for all x and u is polynomially bounded, and $\otimes$ and $\oplus$ are polynomially expressible, then policy iteration converges in a pseudopolynomial number of steps

complexity results: pp.66, the difficulty of solving particular generalized MDPs depends critically on the definitions of $\otimes$ and $\oplus$ and whether $\mathcal{X}$ is finite

reinforcement learning in generalized MDPs: pp.66, in this section I describe how reinforcement-learning algorithms can use experience to converge to optimal policies when T and R are not known in advance, my plan in this section is to introduce a mathematical framework that captures algorithms from both of the model-free (Q-learning) and model-based reinforcement-learning algorithms

a generalized reinforcement-learning method: pp.67, we will consider a learning rule that in the limit converges to HV for a fixed value function V, Q-learning applied to this model begins with an initial value function U_0, we can capture the learning rule in the form of an operator

$$H_t(U,V)(x) = \begin{cases} (1-\alpha_t(x_t))U(x) + \alpha_t(x_t)(r_t + \beta V(x_t')) & \text{if } x = x_t \\ U(x) & \text{otherwise} \end{cases}$$

and define $U_{t+1} = H_t(U_t, V)$, conditions for the convergence of U_t to HV are provided by classic stochastic-approximation theory, a more advanced reinforcement-learning problem is computing $V^* = HV^*$, the fixed point of H, instead of the value of HV for a fixed value function, consider the natural learning algorithm that begins with a value function V_0 and defines $V_{t+1} = H_t(V_t, V_t)$ where H_t is as defined above

a stochastic-approximation theorem: pp.68, let H be a contraction mapping with respect to a weighted max norm with fixed point V^* and let H_t approximate H at V^*, the iterative approximation of V^* is performed by computing $V_{t+1} = H_t(V_t, V_t)$, if there exist functions $0 \leq F_t(x) \leq 1$ satisfying

$$|(H_t(U_1, V^*))(x) - (H_t(U_2, V^*))(x)| \leq G_t(x)\|U_1 - U_2\|$$

and $0 \leq G_t(x) \leq 1$ satisfying

$$|(H_t(U, V^*))(x) - (H_t(U, V))(x)| \leq F_t(x)\|V^* - V\|$$

with probability 1, and there exists $0 \leq \beta < 1$ such that $F_t(x) \leq \beta(1 - G_t(x))$ for large enough t, then V_t converges uniformly to V^* with probability 1; pp.69, $G_t(x)$ is the extent to which the estimated value function depends on its present value and $F_t(x) \approx 1 - G_t(x)$ is the extent to which the estimated value function is based on "new" information, in some applications such as Q-learning, the contribution of new information needs to decay over time to ensure that the process converges, in this case $G_t(x)$ needs to converge to one, but the convergence should be slow enough to incorporate sufficient information, and the requirement that $\beta < 1$ ensures that the use of outdated information in the asynchronous updates does not cause a problem in convergence
pp.69, one of the most noteworthy aspects of this theorem is that it shows how to reduce the problem of approximating V^* to the problem of approximating H at V^*, in many cases the latter is much easier to achieve and also to prove

generalized Q-learning for expected value models: pp.70, the Q-learning rule is equivalent to the approximate dynamic-programming operator

$$H_t(U, V)(x, u) = \begin{cases} (1 - \alpha_t(x_t, u_t))U(x, u) + \alpha_t(x_t, u_t)(r_t + \beta \otimes_u^{(x'_t)} V(x'_t, u)) & \text{if } x = x_t \text{ and } u = u_t \\ U(x, u) & \text{otherwise} \end{cases}$$

in this section I derive the assumptions necessary for this learning algorithm to satisfy the stochastic-approximation theorem above and therefore converge to the optimal Q values, which are

- $\otimes$ is a non-expansion and does not depend on T or R
- the expected value of r given x and u is $R(x, u)$ and r has finite variance
- every state-action pair is updated infinitely often and the learning rates are decayed so that $\sum_{t:x_t=x,u_t=u} \alpha_t(x, u) = \infty$ and $\sum_{t:x_t=x,u_t=u} \alpha_t(x, u)^2 < \infty$

so that

$$G_t(x, u) = \begin{cases} 1 - \alpha_t(x, u) & \text{if } x = x_t \text{ and } u = u_t \\ 0 & \text{otherwise} \end{cases}$$

$$F_t(x, u) = \begin{cases} \beta\alpha_t(x, u) & \text{if } x = x_t \text{ and } u = u_t \\ 0 & \text{otherwise} \end{cases}$$

satisfy the conditions of the stochastic-approximation theorem

model-based methods: pp.71, the fundamental assumption of reinforcement learning is that the reward and transition functions are not known in advance, although Q-learning shows that optimal value functions can sometimes be estimated without ever explicitly learning R and T, learning R and T makes more efficient use of experience at the expense of additional storage and computation, in model-based reinforcement learning

R and T are estimated on-line, and the value function is updated according to the approximate dynamic-programming operator derived from these estimates
in this section we assume that $\oplus$ may depend on T and/or R but $\otimes$ may not, in model-based reinforcement learning R and T are estimated by the quantities R_t and T_t, and $\oplus^t$ is an estimate of the $\oplus$ operator defined using R_t and T_t
pp.72, assuming T and R can be estimated in a way that results in the convergence of $\oplus^t$ to $\oplus$, the approximate dynamic-programming operator H_t defined by

$$H_t(U,V)(x) = \begin{cases} \otimes_u^{(x)} \Big(R_t(x,u) + \beta \oplus_{x'}^{t,(x,u)} V(x') \Big) & \text{if } x \in \tau_t \\ U(x) & \text{otherwise} \end{cases}$$

converges to H with probability 1 uniformly, here $\tau_t \subseteq \mathcal{X}$ represents the set of states whose values are updated on step t, one popular choice is to set $\tau_t = \{x_t\}$, other algorithms use a larger τ_t set to speed up learning, the functions

$$G_t(x) = \begin{cases} 0 & \text{if } x \in \tau_t \\ 1 & \text{otherwise} \end{cases} \qquad F_t(x) = \begin{cases} \beta & \text{if } x \in \tau_t \\ 0 & \text{otherwise} \end{cases}$$

satisfy the conditions of the stochastic-approximation theorem

open problems: pp.73, they are

- can generalized MDPs be extended to infinite action spaces?
- some natural summary operators like Boltzmann weighting do not have the non-expansion property, in the case of Boltzmann weighting there are examples where the H operator has multiple fixed points
- model-free reinforcement-learning updates appear to require the use of the expected reward summary operator $\oplus_{x'}^{(x,u)} g(x') = \sum_{x'} T(x,u,x') g(x')$

4 Week 4 Articles

4.1 Sutton & Barto Reinforcement Learning Chapter 17

designing reward signals: pp.469, a major advantage of reinforcement learning over supervised learning is that reinforcement learning does not rely on detailed instructional information—generating a reward signal does not depend on knowledge of what the agent's correct actions should be, but the success of a reinforcement learning application strongly depends on how well the reward signal frames the goal of the application's designer and how well the signal assesses progress in reaching that goal, for these reasons designing a reward signal is a critical part of any application of reinforcement learning
pp.469, some problems involve goals that are difficult to translate into reward signals, this is especially true when the problem requires the agent to skillfully perform a complex task or set of tasks, further reinforcement learning agents can discover unexpected ways to make their environments deliver reward, some of which might be undesirable or even dangerous, even when there is a simple and easily identifiable goal, the problem of *sparse reward* often arises, delivering non-zero reward frequently enough to allow the agent to achieve the goal once, let alone to learn to achieve it efficiently from multiple initial conditions, can be a daunting challenge, in practice designing a reward signal is often left to an informal trial-and-error search for a signal that produces acceptable results

pp.470, it is tempting to address the sparse reward problem by rewarding the agent for achieving subgoals that the designer thinks are important way stations to overall goals, but augmenting the reward signal with well-intentioned supplemental rewards may lead the agent to behave differently from what is intended; the agent may end up not achieving the overall goal, a better way to provide such guidance is to augment the value-function approximation with an initial guess of what it should ultimately be, or augment it with initial guesses as to what certain parts of it should be, e.g. $\hat{v}(s, \mathbf{w}) := \mathbf{w}^T\mathbf{x}(s) + v_0(s)$
pp.470, a particularly effective approach to the sparse reward problem is the shaping technique introduced by the psychologist B. F. Skinner, the effectiveness of this technique relies on the fact that sparse reward problems are not just problems with the reward signal; they are also problems with an agent's policy in that it prevents the agent from frequently encountering rewarding states, shaping involves changing the reward signal as learning proceeds, starting from a reward signal that is not sparse given the agent's initial behavior, and gradually modifying it toward a reward signal suited to the problem of original interest, shaping might also involve modifying the dynamics of the task as learning proceeds, the agent faces a sequence of increasingly-difficult reinforcement learning problems, where what is learned at each stage makes the next-harder problem relatively easy because the agent now encounters reward more frequently than it would if it did not have prior experience with easier problems, this kind of shaping is an essential technique in training animals, and it is effective in computational reinforcement learning as well
pp.470, learning from an expert's behavior can be done either by learning directly by supervised learning or by extracting a reward signal using what is known as "inverse reinforcement learning", the task of inverse reinforcement learning as explored by Ng and Russell (2000) is to try to recover the expert's reward signal from the expert's behavior alone, this cannot be done exactly because a policy can be optimal with respect to many different reward signals, but it is possible to find plausible reward signal candidates, unfortunately strong assumptions are required including knowledge of the environment's dynamics and of the feature vectors in which the reward signal is linear, the method also requires completely solving the problem multiple times
pp.471, another approach to finding a good reward signal is to automate the trial-and-error search for a good signal, the search for a good reward signal can be automated by defining a space of feasible candidates and applying an optimization algorithm, reward signals can even be improved via online gradient ascent, where the gradient is that of the high-level objective function, however the performance of a reinforcement learning agent as evaluated by the high-level objective function can be very sensitive to details of the agent's reward signal in subtle ways determined by the agent's limitations and the environment in which it acts and learns, an agent's goal should not always be the same as the goal of the agent's designer, the agent has to learn under various kinds of constraints such as limited computational power, limited access to information about its environment, or limited time to learn, when there are constraints like these, learning to achieve a goal that is different from the designer's goal can sometimes end up getting closer to the designer's goal than if that goal were pursued directly, e.g. the nutritional value of most foods vs. tastes; pp.472, reward signals may also depend on properties of the learning process itself, such as measures of how much progress learning is making

5 Week 5 Articles

5.1 Sutton & Barto Reinforcement Learning Chapter 2

pp.25, purely evaluative feedback indicates how good the action taken was, but not whether it was the best or the worst action possible, on the other hand purely instructive feedback indicates the correct action to take, independently of the action actually taken, this kind of feedback is the basis of supervised learning, in their pure forms these two kinds of feedback are quite distinct—evaluative feedback depends entirely on the action taken, whereas instructive feedback is independent of the action taken, in this chapter we study the evaluative aspect of reinforcement learning in a simplified nonassociative setting—when the best action depends on the situation

a k-armed bandit problem: pp.26, we denote the action selected on time step t as A_t and the corresponding reward as R_t, the value then of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected: $q_*(a) := \mathbb{E}[R_t | A_t = a]$, we denote the estimated value of action a at time step t as $Q_t(a)$, we would like $Q_t(a)$ to be close to $q_*(a)$, at any time step there is at least one action whose estimated value is greatest, we call these the *greedy* actions, when you select one of these actions, we say that you are *exploiting* your current knowledge of the values of the actions, if instead you select one of the nongreedy actions, then we say you are *exploring*, because this enables you to improve your estimate of the nongreedy action's value, exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce the greater total reward in the long run—if you have many time steps ahead on which to make action selections, then it may be better to explore the nongreedy actions and discover which of them are better than the greedy action, in any specific case whether it is better to explore or exploit depends in a complex way on the precise values of the estimates, uncertainties, and the number of remaining steps; pp.27, in this chapter we present several simple balancing methods for the k-armed bandit problem and show that they work much better than methods that always exploit

action-value methods: pp.27, the true value of an action is the mean reward when that action is selected, one natural way to estimate this is by averaging the rewards actually received, we call this the sample-average method for estimating action values because each estimate is an average of the sample of relevant rewards, 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 $A_t := \arg\max_a Q_t(a)$, greedy action selection always exploits current knowledge to maximize immediate reward, a simple alternative is every once in a while, say with small probability ϵ, instead select randomly from among all the actions with equal probability independently of the action-value estimates, we call methods using this near-greedy action selection rule *ϵ-greedy* methods, an advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the $Q_t(a)$ converge to their respective $q_*(a)$, this implies that the probability of selecting the optimal action converges to greater than $1 - \epsilon$

the 10-armed testbed: pp.30, with noisier rewards it takes more exploration to find the optimal action, and ϵ-greedy methods should fare even better relative to the greedy method, on the other hand if the reward variances were zero, in this case the greedy method might actually perform better because it would soon find the optimal action and then never explore, but even in the deterministic case there is a large advantage to exploring if we weaken some of the other assumptions such as nonstationary

incremental implementation: pp.31, let R_i denote the reward received after the ith selection of this action, and let Q_n denote the estimate of its action value after it has been selected $n-1$ times $Q_n := (R_1 + R_2 + \cdots + R_{n-1})/(n-1)$, given Q_n and the nth reward R_n, the new average of all n rewards can be computed by $Q_{n+1} = Q_n + (1/n)(R_n - Q_n)$, this update rule is of a form that occurs frequently throughout this book, the general form is

$$\text{new estimate} \leftarrow \text{old estimate} + \text{step size} \times [\text{target} - \text{old estimate}]$$

note that the step-size parameter used in the incremental method changes from time step to time step, in this book we denote the step-size parameter by α or more generally by $\alpha_t(a)$

tracking a nonstationary problem: pp.32, we often encounter reinforcement learning problems that are effectively nonstationary, in such cases it makes sense to give more weight to recent rewards than to long-past rewards, one of the most popular ways of doing this is to use a constant step-size parameter, e.g.

$$\begin{aligned} Q_{n+1} &:= Q_n + \alpha\,(R_n - Q_n) \\ &= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i \end{aligned}$$

pp.33, the weight given to R_i decays exponentially according to the exponent on $1-\alpha$, accordingly this is sometimes called an exponential recency-weighted average
pp.33, sometimes it is convenient to vary the step-size parameter from step to step, but of course convergence is not guaranteed for all choices of the sequence $\{\alpha_n(a)\}$, a well-known result in stochastic approximation theory gives us the conditions required to assure convergence with probability 1

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

the first condition is required to guarantee that the steps are large enough to eventually overcome any initial conditions or random fluctuations, the second condition guarantees that eventually the steps become small enough to assure convergence
note that both convergence conditions are met for the sample-average case $\alpha_n(a) = 1/n$ but not for the case of constant step-size parameter $\alpha_n(a) = \alpha$ which however is desirable in a nonstationary environment, in addition sequences of step-size parameters that meet the conditions often converge very slowly or need considerable tuning in order to obtain a satisfactory convergence rate, although sequences of step-size parameters that meet these convergence conditions are often used in theoretical work, they are seldom used in applications and empirical research

optimistic initial values: pp.34, all the methods we have discussed so far are dependent to some extent on the initial action-value estimates $Q_1(a)$, in the language of statistics these methods are biased by their initial estimates, for the sample-average methods the bias disappears once all actions have been selected at least once, but for methods with constant α

the bias is permanent though decreasing over time, the downside is that the initial estimates become in effect a set of parameters that must be picked by the user, the upside is that they provide an easy way to supply some prior knowledge about what level of rewards can be expected
pp.34, initial action values can also be used as a simple way to encourage exploration, we call this technique for encouraging exploration optimistic initial values, we regard it as a simple trick that can be quite effective on stationary problems, but it is far from being a generally useful approach to encouraging exploration, for example it is not well suited to nonstationary problems because its drive for exploration is inherently temporary; pp.35, nevertheless all of these methods are very simple, and one of them or some simple combination of them is often adequate in practice
pp.35, one way to avoid the bias of constant step sizes while retaining their advantages on nonstationary problems is to use a step size of $\beta_n := \alpha/o_n$ where $o_n := o_{n-1} + \alpha(1 - o_{n-1})$ for $n > 0$ with $o_0 := 0$

upper-confidence-bound action selection: pp.35, it would be better to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates, one effective way of doing this is to select actions according to

$$A_t := \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

where $N_t(a)$ denotes the number of times that action a has been selected prior to time t, if $N_t(a) = 0$ then a is considered to be a maximizing action, the idea of this upper confidence bound (UCB) action selection is that the square-root term is a measure of the uncertainty or variance in the estimate of a's value, the quantity being max'ed over is thus a sort of upper bound on the possible true value of action a with c determining the confidence level, each time a is selected the uncertainty is presumably reduced, on the other hand each time an action other than a is selected t increases but $N_t(a)$ does not, because t appears in the numerator the uncertainty estimate increases, all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time
pp.36, UCB often performs well, but is more difficult than ϵ-greedy to extend beyond bandits to the more general reinforcement settings, one difficulty is in dealing with nonstationary problems, another difficulty is dealing with large state spaces, particularly when using function approximation as developed in Part II of this book, in these more advanced settings the idea of UCB action selection is usually not practical

summary: pp.42, in assessing a method we should attend not just to how well it does at its best parameter setting, but also to how sensitive it is to its parameter value, all of these algorithms—ϵ-greedy, greedy with optimistic initialization, UCB, gradient bandit—are fairly insensitive, overall on this problem UCB seems to perform best
pp.43, one well-studied approach to balancing exploration and exploitation in k-armed bandit problems is to compute a special kind of action value called a Gittins index, the Gittins-index approach is an instance of Bayesian methods, which assume a known initial distribution over the action values and then update the distribution exactly after each step

6 Week 6 Articles

6.1 Sutton & Barto Reinforcement Learning Chapter 9

pp.197, the novelty in this chapter is that the approximate value function is represented not as a table but as a parametrized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$, we will write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\mathbf{w}$, typically the number of weights (the dimensionality of $\mathbf{w}$) is much less than the number of states ($d \ll \mathcal{S}$), and changing one weight changes the estimated value of many states, consequently when a single state is updated, the change generalizes from that state to affect the values of many other states, such *generalization* makes the learning potentially more powerful but also potentially more difficult to manage and understand
pp.197, perhaps surprisingly extending reinforcement learning to function approximation also makes it applicable to partially observable problems, in which the full state is not available to the agent, if the parametrized function form for $\hat{v}$ does not allow the estimated value to depend on certain aspects of the state, then it is just as if those aspects are unobservable, in fact all the theoretical results for methods using function approximation presented in this part of the book apply equally well to cases of partial observability

value-function approximation: pp.198, all of the prediction methods covered in this book have been described as updates to an estimated value function that shift its value at a particular state s toward the update target u denoted by $s \mapsto u$, e.g.

Monte Carlo: $S_t \mapsto G_t$

TD(0): $S_t \mapsto R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t)$

n-step TD: $S_t \mapsto G_{t:t+n}$

DP policy: $s \mapsto \mathbb{E}_\pi\left[R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) | S_t = s\right]$

it is natural to interpret each update as specifying an example of the desired input-output behavior of the value function, machine learning methods that learn to mimic input-output examples in this way are called supervised learning methods, and when the outputs are numbers like u, the process is often called *function approximation*, in principle we can use any method for supervised learning, however not all function approximation methods are equally well suited for use in reinforcement learning, the most sophisticated artificial neural network and statistical methods all assume a static training set over which multiple passes are made, however in reinforcement learning it is important that learning be able to occur online, to do this requires methods that are able to learn efficiently from incrementally acquired data, in addition reinforcement learning generally requires function approximation methods able to handle nonstationary target functions, methods that cannot easily handle such nonstationarity are less suitable for reinforcement learning

the prediction objective ($\overline{\mathbf{VE}}$): pp.199, by assumption we have far more states than weights, so making one state's estimate more accurate invariably means making others' less accurate, we are obligated then to say which states we care most about, we must specify a state distribution $\mu(s) \geq 0$, $\sum_s \mu(s) = 1$, representing how much we care about the error in each state s, by the error in a state s we mean the square of the difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$, weighting this over the state space by μ

we obtain a natural objective function, the mean square value error, denoted $\overline{\mathrm{VE}}$ as follows

$$\overline{\mathrm{VE}}(\mathbf{w}) := \sum_{s\in\mathcal{S}} \mu(s)\left[v_\pi(s) - \hat{v}(s,\mathbf{w})\right]^2$$

often $\mu(s)$ is chosen to be the fraction of time spent in s, under on-policy training this is called the *on-policy distribution*, we focus entirely on this case in this chapter, in continuing tasks the on-policy distribution is the stationary distribution under π, in an episodic task the on-policy distribution is a little different in that it depends on how the initial states of episodes are chosen, let $\eta(s)$ denote the number of time steps spent on average in state s in a single episode, the on-policy distribution is then the fraction of time spent in each state normalized to sum to one

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')} \quad \forall s \in \mathcal{S}$$

the two cases, continuing and episodic, behave similarly, but with approximation they must be treated separately in formal analyses
pp.200, remember that our ultimate purpose—the reason we are learning a value function—is to find a better policy, the best value function for this purpose is not necessarily the best for minimizing $\overline{\mathrm{VE}}$, nevertheless it is not yet clear what a more useful alternative goal for value prediction might be
pp.200, an ideal goal in terms of $\overline{\mathrm{VE}}$ would be to find a global optimum, a weight vector $\mathbf{w}^*$ for which $\overline{\mathrm{VE}}(\mathbf{w}^*) \leq \overline{\mathrm{VE}}(\mathbf{w})$ for all possible $\mathbf{w}$, which is rarely possible for complex function approximators such as artificial neural networks and decision trees, short of this complex function approximators may seek to converge instead to a local optimum, a weight vector $\mathbf{w}^*$ for which $\overline{\mathrm{VE}}(\mathbf{w}^*) \leq \overline{\mathrm{VE}}(\mathbf{w})$ for all $\mathbf{w}$ in some neighborhood of $\mathbf{w}^*$, although this guarantee is only slightly reassuring, it is typically the best that can be said for nonlinear function approximators, and often it is enough

stochastic-gradient and semi-gradient methods: pp.201, let us assume that on each step we observe a new example $S_t \mapsto v_\pi(S_t)$ consisting of a (possibly randomly selected) state S_t and its true value under the policy, even though we are given the exact correct values $v_\pi(S_t)$ for each S_t, there is generally no $\mathbf{w}$ that gets all the states or even all the examples exactly correct, we assume that states appear in examples with the same distribution μ over which we are trying to minimize the $\overline{\mathrm{VE}}$, a good strategy in this case is to try to minimize error on the observed examples, *stochastic gradient-descent* (SGD) methods do this by

$$\begin{aligned}\mathbf{w} &:= \mathbf{w}_t - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)]^2 \\ &= \mathbf{w}_t + \alpha\left[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)\right]\nabla\hat{v}(S_t,\mathbf{w}_t)\end{aligned}$$

gradient descent methods are called "stochastic" when the update is done on only a single example which might have been selected stochastically, the convergence results for SGD methods assume that α decreases over time, if it decreases in such a way as to satisfy the standard stochastic approximation conditions

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \qquad \text{and} \qquad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

then the SGD method above is guaranteed to converge to a local optimum
pp.201, we turn now to the case in which the target output, here denoted $U_t \in \mathbb{R}$ of

the tth training example $S_t \mapsto U_t$ is not the true value $v_\pi(S_t)$ but some possibly random approximation to it, in this case we can approximate $v_\pi(S_t)$ by substituting U_t in place of $v_\pi(S_t)$, this yields the following general SGD method for state-value prediction

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t)\right] \nabla \hat{v}(S_t, \mathbf{w}_t)$$

if U_t is an unbiased estimate, that is if $\mathbb{E}[U_t | S_t = s] = v_\pi(s)$ for each t, e.g. the Monte Carlo target $U_t := G_t$ is by definition an unbiased estimate of $v_\pi(S_t)$, then $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions above
pp.202, one does not obtain the same guarantees if a bootstrapping estimate of $v_\pi(S_t)$ is used as the target U_t, because bootstrapping targets such as n-step returns $G_{t:t+n}$ or the DP target $\sum_{a,s',r} \pi(a|S_t)p(s', r|S_t, a)[r + \gamma \hat{v}(s', \mathbf{w}_t)]$ all depend on the current value of the weight vector $\mathbf{w}_t$, which implies that they will be biased and that they will not produce a true gradient-descent method, since the key step of taking gradient relies on the target being independent of $\mathbf{w}_t$, this step would not be valid if a bootstrapping estimate were used in place of $v_\pi(S_t)$, bootstrapping methods take into account the effect of changing the weight vector $\mathbf{w}_t$ on the estimate, but ignore its effect on the target, they include only a part of the gradient and accordingly we call them *semi-gradient methods*
although semi-gradient (bootstrapping) methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the linear case discussed in the next section, moreover they offer important advantages that make them often clearly preferred, one reason for this is that they typically enable significantly faster learning, another is that they enable learning to be continual and online, without waiting for the end of an episode, a prototypical semi-gradient method is semi-gradient TD(0), which uses $U_t := R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ as its target

linear methods: pp.204, linear methods approximate the state-value function by the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$

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

the vector $\mathbf{x}(s)$ is called a *feature vector* representing state s, each component $x_i(s)$ of $\mathbf{x}(s)$ is the value of a function $x_i : \mathcal{S} \to \mathbb{R}$, for linear methods features are basis functions because they form a linear basis for the set of approximate functions, the gradient of the approximate value function with respect to $\mathbf{w}$ is $\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$, thus in the linear case the general SGD update reduces to a particularly simple form

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t)\right] \mathbf{x}(S_t)$$

because it is so simple, almost all useful convergence results for learning systems of all kinds are for linear (or simpler) function approximation methods, in particular in the linear case there is only one optimum (or in degenerate cases one set of equally good optima) and thus any method that is guaranteed to converge to or near a local optimum is automatically guaranteed to converge to or near the global optimum
pp.205, the semi-gradient TD(0) algorithm presented in the previous section also converges under linear function approximation, but this does not follow from general results on SGD, a separate theorem is necessary, the weight vector converged to is also not the global optimum, but rather a point near the local optimum, the update at each time t is

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

the expected next weight vector can be written

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

$$\text{where } \mathbf{b} := \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \quad \text{and} \quad \mathbf{A} := \mathbb{E}\left[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T\right] \in \mathbb{R}^{d\times d}$$

if the system converges, it must converge to the weight vector $\mathbf{w}_{\mathrm{TD}} = \mathbf{A}^{-1}\mathbf{b}$, this quantity is called the *TD fixed point*, in fact linear semi-gradient TD(0) converges to this point
pp.207, at the TD fixed point it has also been proven (in the continuing case) that the $\overline{\mathrm{VE}}$ is within a bounded expansion of the lowest possible error

$$\overline{\mathrm{VE}}(\mathbf{w}_{\mathrm{TD}}) \leq \frac{1}{1-\gamma}\min_{\mathbf{w}}\overline{\mathrm{VE}}(\mathbf{w})$$

that is the asymptotic error of the TD method is no more than $1/(1-\gamma)$ times the smallest possible error that attained in the limit by the Monte Carlo method, because γ is often near one, this expansion factor can be quite large, nonetheless the TD methods are often of vastly reduced variance compared to Monte Carlo methods
pp.208, a bound analogous to the above applies to other on-policy bootstrapping methods as well, e.g. linear semi-gradient DP with $U_t := \sum_a \pi(a|S_t)\sum_{s',r} p(s',r|S_t,a)[r + \gamma\hat{v}(s',\mathbf{w}_t)]$ with updates according to the on-policy distribution will also converge to the TD fixed point, one-step semi-gradient action-value methods such as semi-gradient Sarsa(0) covered in the next chapter converge to an analogous fixed point and an analogous bound
pp.208, critical to these convergence results is that states are updated according to the on-policy distribution, for other update distributions bootstrapping methods using function approximation may actually diverge to infinity
pp.209, the key gradient update equation for n-step TD is

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

where the n-step return is

$$G_{t:t+n} := R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n\hat{v}\left(S_{t+n},\mathbf{w}_{t+n-1}\right) \qquad 0 \leq t \leq T-n$$

feature construction for linear methods: pp.210, choosing features appropriate to the task is an important way of adding prior domain knowledge to reinforcement learning systems, 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, it needs instead or in addition features for combinations of these two

polynomials: pp.210, the states of many problems are initially expressed as numbers, in these types of problems function approximation for reinforcement learning has much in common with the familiar tasks of interpolation and regression, polynomials make up one of the simplest families of features used for interpolation and regression, while the basic polynomial features we discuss here do not work as well as other types of features in reinforcement learning, they serve as a good introduction because they are simple and familiar
as an example suppose a reinforcement learning problem has states with two numerical dimensions $s = (s_1, s_2)^T$, we can represent s by the four-dimensional feature vector $\mathbf{x}(s) = (1, s_1, s_2, s_1s_2)^T$, the initial 1 feature allows the representation of affine functions

in the original state numbers, and the final product feature s_1s_2 enables interactions to be taken into account

pp.211, because the number of features in an order-n polynomial basis grows exponentially with the dimension k of the natural state space, it is generally necessary to select a subset of them for function approximation

pp.214, in general we do not recommend using polynomials for online learning

Fourier basis: pp.211, another linear function approximation method is based on Fourier series, which expresses periodic functions as weighted sums of sine and cosine basis functions (features) of different frequencies, if you are interested in approximating an aperiodic function defined over a bounded interval, then you can use these Fourier basis features with τ set to the length of the interval, furthermore if you set τ to twice the length of the interval of interest and restrict attention to the approximation over the half interval $[0, \tau/2]$ then you can use just the cosine features, alternatively it is possible to use just sine features, but it is generally better to keep just the cosine features because "half-even" functions tend to be easier to approximate than "half-odd" functions because the latter are often discontinuous at the origin

the one-dimensional order-n Fourier cosine basis consists of the $n+1$ features $x_i(s) = \cos(i\pi s)$ for $i = 0, \ldots, n$, the Fourier cosine series approximation can be applied to the multidimensional case as follows: 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 $x_i(s) = \cos(\pi \mathbf{s}^T \mathbf{c}^i)$ where $\mathbf{c}^i = (c_1^i, \ldots, c_k^i)^T$ with $c_j^i \in \{0, \ldots, n\}$ for $j = 1, \ldots, k$ and $i = 1, \ldots, (n+1)^k$; pp.213, as an example consider the $k = 2$ case in which $\mathbf{s} = (s_1, s_2)^T$ where each $\mathbf{c}^i = (c_1^i, c_2^i)^T$, the values of c_1 and c_2 determine the frequency along each dimension, and their ratio gives the direction of the interaction

pp.213, when using Fourier cosine features with a learning algorithm such as semi-gradient TD(0) or semi-gradient Sarsa, it may be helpful to use a different step-size parameter for each feature, Konidaris, Osentoski, and Thomas suggest setting the step-size parameter for feature x_i to $\alpha_i = \alpha/\sqrt{(c_1^i)^2 + \cdots + (c_k^i)^2}$, Fourier cosine features with Sarsa can produce good performance compared to several other collections of basis functions including polynomial and radial basis functions, however Fourier features have trouble with discontinuities because it is difficult to avoid "ringing" around points of discontinuity unless very high frequency basis functions are included

pp.214, the number of features in the order-n Fourier basis grows exponentially with the dimension of the state space, thus for high dimension state spaces it is necessary to select a subset of these features, an advantage of Fourier basis features in this regard is that it is easy to select features by setting the $\mathbf{c}^i$ vectors to account for suspected interactions among the state variables and by limiting the values in the $\mathbf{c}^j$ vectors so that the approximation can filter out high frequency components considered to be noise, on the other hand because Fourier features are non-zero over the entire state space, they represent global properties of states, which can make it difficult to find good ways to represent local properties

coarse coding: pp.215, 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, if the state is inside a circle then the corresponding feature has the value 1 and is said to be present; otherwise the feature is 0 and is said to be absent, this kind of 1–0-valued feature is called a binary feature, given a state which binary features are present indicate within which circles the state lies and thus coarsely code for its location, representing a state with features that overlap in this way (although their receptive fields need not be circles or binary; pp.216, with just one dimension the receptive fields were intervals rather than circles) is known as coarse coding

pp.215, if we train at one state then the weights of all circles intersecting that state will be affected, thus the approximate value function will be affected at all states within the union of the circles, moreover the shape of the features will determine the nature of the generalization e.g. elongated in one direction, initial generalization from one point to another is indeed controlled by the size and shape of the receptive fields, but acuity, the finest discrimination ultimately possible, is controlled more by the total number of features, receptive field shape tends to have a strong effect on generalization but little effect on asymptotic solution quality

tile coding: pp.217, tile coding is a form of coarse coding for multi-dimensional continuous spaces, 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, generalization would be complete to all states within the same tile and nonexistence to states outside it, with just one tiling we would not have coarse coding but just a case of state aggregation, 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

an immediate practical advantage of tile coding is that, because it works with partitions, the overall number of features that are active at one time is the same for any state, exactly one feature is present in each tiling, so the total number of features present is always the same as the number of tilings, this allows the step-size parameter α to be set in an easy intuitive way, e.g. choosing $\alpha = 1/n$ where n is the number of tilings results in exact one-trial learning, usually one wishes to change more slowly than this, to allow for generalization and stochastic variation in target outputs, e.g. one might choose $\alpha = 1/10n$

pp.218, tile coding also gains computational advantages from its use of binary feature vectors, because each component is either 0 or 1, the weighted sum making up the approximate value function is almost trivial to compute, one simply computes the indices of the $n \ll d$ active features and then adds up the n corresponding components of the weight vector

pp.218, generalization occurs to states other than the one trained if those states fall within any of the same tiles, proportional to the number of tiles in common, even the choice of how to offset the tilings from each other affects generalization, if they are offset uniformly in each dimension, note how uniform offsets result in a strong effect along the diagonal in many patterns, these artifacts can be avoided if the tilings are offset asymmetrically; pp.220, Miller and Glanz (1996) recommend using displacement vectors consisting of the first odd integers, in particular for a continuous space of dimension k, a good choice is to use the first odd integers $(1, 3, 5, \ldots, 2k-1)$ with n the number of tilings set to an integer power of 2 greater than or equal to $4k$, the shape of the tiles will determine the nature of generalization, tilings need not be grids, they can be arbitrarily shaped and non-uniform while still in many cases being computationally efficient to compute, in practice it is often desirable to use different shaped tiles in different tilings

pp.221, another useful trick for reducing memory requirements is hashing—a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles, hashing produces tiles consisting of noncontiguous disjoint regions randomly spread throughout the state space, but that still form an exhaustive partition, this is possible because high resolution is needed in only a small fraction of the state space, hashing frees us from the curse of dimensionality in the sense that memory requirements need not be exponential in the number of dimensions

radial basis functions: pp.221, radial basis functions (RBFs) are the natural generalization of coarse coding to continuous-valued features, a typical RBF feature x_i has a Gaussian response $x_i(s)$ dependent only on the distance between the state s and the feature's prototypical or center state c_i and relative to the feature's width σ_i

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

the primary advantage of RBFs over binary features is that they produce approximate functions that vary smoothly and are differentiable, although this is appealing in most cases it has no practical significance, all of these methods require substantial additional computational complexity (over tile coding) and often reduce performance when there are more than two state dimensions, in high dimensions the edges of tiles are much more important, and it has proven difficult to obtain well controlled graded tile activations near the edges

pp.222, an RBF network is a linear function approximator using RBFs for its features, in addition some learning methods for RBF networks change the centers and widths of the features as well, bringing them into the realm of nonlinear function approximators, the downside to RBF networks and to nonlinear RBF networks especially is greater computational complexity and often more manual tuning before learning is robust and efficient

nonlinear function approximation: artificial neural networks: pp.223, in a generic feedforward ANN 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, if an ANN has at least one loop in its connections, it is a recurrent rather than a feedforward ANN, although both feedforward and recurrent ANNs have been used in reinforcement learning, here we look only at the simpler feedforward case

pp.224, 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, this is also true for other nonlinear activation functions that satisfy mild conditions, but nonlinearity is essential—if all the units in a multi-layer feedforward ANN have linear activation functions, the entire network is equivalent to a network with no hidden layers (because linear functions of linear functions are themselves linear), despite this universal approximation property of one-hidden-layer ANNs, both experience and theory show that approximating the complex functions needed for many artificial intelligence tasks is made easier, indeed may require abstractions that are hierarchical compositions of many layers of lower-level abstractions, training the hidden

layers of an ANN is therefore a way to automatically create features appropriate for a given problem so that hierarchical representations can be produced without relying exclusively on hand-crafted features
pp.225, ANNs typically learn by a stochastic gradient method, the most successful way to do this for ANNs with hidden layers 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, the backpropagation algorithm can produce good results for shallow networks having 1 or 2 hidden layers, but it may not work well for deeper ANNs, in fact training a network with $k+1$ hidden layers can actually result in poorer performance than training a network with k hidden layers, because

- the large number of weights in a typical deep ANN makes it difficult to avoid the problem of overfitting; pp.226, albeit it is less of a problem for online reinforcement learning that does not rely on limited training sets
- 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

pp.226, Hinton, Osindero, and Teh (2006) took a major step toward solving the problem of training the deep layers of a deep ANN in their work with deep belief networks, without relying on the overall objective function, unsupervised learning can extract features that capture statistical regularities of the input stream, the deepest layer is trained first, then with input provided by this trained layer the next deepest layer is trained and so on, until the weights in all or many of the network's layers are set to values that now act as initial values for supervised learning, the network is then fine-tuned by backpropagation with respect to the overall objective function
batch normalization (Ioffe and Szegedy 2015) is another technique that makes it easier to train deep ANNs, it has long been know that ANN learning is easier if the network input is normalized, batch normalization for training deep ANNs normalizes the output of deep layers before they feed into the following layer
another technique useful for training deep ANNs is deep residual learning (He, Zhang, Ren, and Sun 2016), sometimes it is easier to learn how a function differs from the identity function than to learn the function itself, then adding this difference or residual function to the input produces the desired function, in deep ANNs a block of layers can be made to learn a residual function simply by adding shortcut, or skip connections around the block
a type of deep ANN that has prove to be very successful in applications including impressive reinforcement learning applications (Chapter 16) is the deep convolutional network, this type of network is specialized for processing high-dimensional data arranged in spatial arrays such as images, 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, the subsampling layers of a deep convolutional network reduce the spatial resolution of the feature maps, subsampling layers reduce the network's sensitivity to the spatial locations of the features detected, that is they help make the network's response spatially invariant

summary: pp.236, reinforcement learning systems must be capable of generalization, to achieve this any of a broad range of existing methods for supervised-learning function approximation can be used simply by treating each update as a training example, perhaps the most suitable supervised learning methods are those using parametrized function approximation, in which the policy is parametrized by a weight vector $\mathbf{w}$, to find a good weight vector the most popular methods are variations of stochastic gradient descent (SGD), in this chapter we have focused on the on-policy case with a fixed policy also known as policy evaluation or prediction, semi-gradient TD methods are not true gradient methods, in such bootstrapping methods (including DP) the weight vector appears in the update target, yet this is not taken into account in computing the gradient

choosing the features is one of the most important ways of adding prior domain knowledge to reinforcement learning systems, they can be chosen as polynomials, but this case generalizes poorly in the online learning setting typically considered in reinforcement learning, tile coding is a form of coarse coding that is particularly computationally efficient and flexible, radial basis functions are useful for one- or two-dimensional tasks in which a smoothly varying response is important

6.2 Sutton & Barto Reinforcement Learning Chapter 10

pp.243, in this chapter we return to the control problem, now with parametric approximation of the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$ where $\mathbf{w} \in \mathbb{R}^d$ is a finite-dimensional weight vector, we continue to restrict attention to the on-policy case leaving off-policy methods to Chapter 11, the present chapter features the semi-gradient Sarsa algorithm, the natural extension of semi-gradient TD(0) (Chapter 9) to action values and to on-policy control, starting first in the episodic case we extend the function approximation ideas presented in the last chapter from state values to action values

episodic semi-gradient control: pp.243, in this case it is the approximate action-value function $\hat{q} \approx q_\pi$ that is presented as a parametrized functional form with weight vector $\mathbf{w}$, whereas before we considered random training examples of the form $S_t \mapsto U_t$, now we consider examples of the form $S_t, A_t \mapsto U_t$, the update target U_t can be any approximation of $q_\pi(S_t, A_t)$, the general gradient-descent update for action-value prediction is

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)\right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

for example the update for the one-step Sarsa method is

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)\right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

we call this method episodic semi-gradient one-step Sarsa, for a constant policy this method converges in the same way that TD(0) does with the same kind of error bound

$$\overline{\mathrm{VE}}(\mathbf{w}_{\mathrm{TD}}) \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\mathrm{VE}}(\mathbf{w})$$

pp.244, to form control methods we need to couple such action-value prediction methods with techniques for policy improvement and action selection, suitable techniques applicable to continuous actions or to actions from large discrete sets are a topic of ongoing research with as yet no clear resolution, on the other hand if the action set is discrete and not

too large, then we can use the techniques already developed in previous chapters, that is for each possible action a available in the next state S_{t+1}, we can compute $\hat{q}(S_{t+1}, a, \mathrm{w}_t)$ and then find the greedy action $A^*_{t+1} = \arg\max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$, policy improvement is then done in the on-policy case treated in this chapter by changing the estimation policy to a soft approximation of the greedy policy such as the ϵ-greedy policy, actions are selected according to this same policy
pp.245, the mountain car task is a simple example of a continuous control task where things have to get worse in a sense (further from the goal) before they can get better, many control methodologies have great difficulties with tasks of this kind unless explicitly aided by a human designer

summary: pp.256, in this chapter we have extended the ideas of parametrized function approximation and semi-gradient descent introduced in the previous chapter to control, the extension is immediate for the episodic case, but for the continuing case we have to introduce a whole new problem formulation based on maximizing the average reward setting per time step, surprisingly the discounted formulation cannot be carried over to control in the presence of approximations, the average reward formulation involves new differential versions of value functions, Bellman equations, and TD errors, but all of these parallel the old ones

6.3 Sutton & Barto Reinforcement Learning Chapter 11

pp.257, the extension to function approximation turns out to be significantly different and harder for off-policy learning than it is for on-policy learning, the tabular off-policy methods developed in Chapters 6 and 7 readily extend to semi-gradient algorithms, but these algorithms do not converge as robustly as they do under on-policy training, in this chapter we explore the convergence problems, in the end we will have improved methods, but the theoretical results will not be as strong, nor the empirical results as satisfying, as they are for on-policy learning
pp.257, in off-policy learning we seek to learn a value function for a target policy π given data due to a different behavior policy b, in the prediction case both policies are static and given, and we seek to learn either state values $\hat{v} \approx v_\pi$ or action values $\hat{q} \approx q_\pi$, in the control case action values are learned and both policies typically change during learning—π being the greedy policy with respect to $\hat{q}$ and b being something more exploratory such as the ϵ-greedy policy with respect to $\hat{q}$
the challenge of off-policy learning can be divided into two parts, the first part has to do with the target of the update and the second part has to do with the distribution of the updates, the techniques related to importance sampling developed in Chapters 5 and 7 deal with the first part, the extension of these techniques to function approximation are quickly dealt with in the first section, something more is needed for the second part of the challenge of off-policy learning with function approximation, because the distribution of updates in the off-policy case is not according to the on-policy distribution, the on-policy distribution is important to the stability of semi-gradient methods, one is to use importance sampling methods again, and the other is to develop true gradient methods that do not rely on any special distribution for stability, this is a cutting-edge research area, and it is not clear which of these approaches is most effective in practice

semi-gradient methods: pp.258, in Chapter 7 we described a variety of tabular off-policy algorithms, many of these algorithms use the per-step importance sampling ratio

$$\rho_t := \rho_{t:t} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$$

to convert them to semi-gradient form we simply replace the update to an array (V or Q) to an update to a weight vector ($\mathbf{w}$), for example the one-step state-value algorithm is semi-gradient off-policy TD(0) which is just like the corresponding on-policy algorithm except for the addition of ρ_t

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\rho_t\delta_t\nabla\hat{v}(S_t, \mathbf{w}_t)$$

where δ_t is defined appropriately depending on whether the problem is episodic and discounted, or continuing and undiscounted using average reward, for action values the one-step algorithm is semi-gradient expected Sarsa

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\delta_t\nabla\hat{q}(S_t, A_t, \mathbf{w}_t)$$

note that this algorithm does not use importance sampling, in the tabular case it is clear that this is appropriate because the only sample action is A_t, and in learning its value we do not have to consider any other actions, with function approximation it is less clear because we might want to weight different state-action pairs differently once they all contribute to the same overall approximation

examples of off-policy divergence: pp.260, suppose perhaps as part of a larger MDP there are two states whose estimated values are of the functional form w and $2w$, in the first state only one action is available, and it results deterministically in a transition to the second state with a reward of 0, the importance sampling ratio ρ_t is 1 on this transition because there is only one action available, in the off-policy semi-gradient TD(0) update the new parameter is the old parameter times a scalar constant $1+\alpha(2\gamma-1)$, if this constant is greater than 1 then the system is unstable and w will go to positive or negative infinity depending on its initial value, key to this example is that the one transition occurs repeatedly without w being updated on other transitions, under on-policy training however each time there is a transition into the $2w$ state, there would also have to be a transition out of the $2w$ state, that transition would reduce w
pp.261, a simple complete MDP example of divergence is Baird's counterexample, if we apply semi-gradient TD(0) to this problem then the weights diverge to infinity, if we alter just the distribution of DP updates in Baird's counterexample, from the uniform distribution to the on-policy distribution, then convergence is guaranteed, this example is striking because the TD and DP methods used are arguably the simplest and best-understood bootstrapping methods, and the linear semi-descent method used is arguably the simplest and best-understood kind of function approximation, the example shows that even the simplest combination of bootstrapping and function approximation can be unstable if the updates are not done according to the on-policy distribution
pp.263, there are also counterexamples similar to Baird's showing divergence for Q-learning, this is cause for concern because otherwise Q-learning has the best convergence guarantees of all control methods
pp.264, another way to try to prevent instability is to use special methods for function approximation, in particular stability is guaranteed for function approximation methods

that do not extrapolate from the observed targets, these methods called *averagers* include nearest neighbor methods and locally weighted regression, but not popular methods such as tile coding and artificial neural networks (ANNs)

the deadly triad: pp.264, the danger of instability and divergence arises whenever we combine all of the following three elements, making up what we call the deadly triad

function approximation: a scalable way of generalizing from a state space much larger than the memory and computational resources

bootstrapping: update targets that include existing estimates rather than relying exclusively on actual rewards and complete returns

off-policy training: training on a distribution of transitions other than that produced by the target policy (sweeping through the state space and updating all states uniformly as in DP does not respect the target policy)

if any two elements of the deadly triad are present but not all three, then instability can be avoided, it is natural then to go through the three and see if there is any one that can be given up
pp.264, of the three function approximation most clearly cannot be given up, because we need methods that scale to large problems and to great expressive power
doing without bootstrapping is possible at the cost of computational and data efficiency, perhaps most important are the losses in computational efficiency, with bootstrapping and eligibility traces data can be dealt with when and where it is generated then need never be used again, the losses in data efficiency by giving up bootstrapping are also significant, bootstrapping often results in faster learning because it allows learning to take advantage of the state property, the ability to recognize a state upon returning to it
off-policy methods free behavior from the target policy, this could be considered an appealing convenience but not a necessity, however off-policy learning is essential to other anticipated use cases, in these use cases the agent learns not just a single value function and single policy but large numbers of them in parallel, there are many target policies and thus the one behavior policy cannot equal all of them, to take full advantage of parallel learning requires off-policy learning

summary: pp.284, one reason to seek off-policy algorithms is to give flexibility in dealing with the tradeoff between exploration and exploitation, another is to free behavior from learning and avoid the tyranny of the target policy, TD learning appears to hold out the possibility of learning about multiple things in parallel, of using one stream of experience to solve many tasks simultaneously
in this chapter we divided the challenge of off-policy learning into two parts, the first part correcting the targets of learning for the behavior policy is straightforwardly dealt with using the techniques devised earlier for the tabular case, the second part of the challenge of off-policy learning emerges as the instability of semi-gradiate TD methods that involve bootstrapping, we seek powerful function approximation, off-policy learning, and the efficiency and flexibility of bootstrapping TD methods, but it is challenging to combine all three aspects of this deadly triad in one algorithm without introducing the potential for instability, after all the whole area of off-policy learning is relatively new and unsettled

6.4 Sutton & Barto Reinforcement Learning Chapter 13

pp.321, in this chapter we consider methods that instead learn a parameterized policy that can select actions without consulting a value function, we use the notation $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ for the policy's parameter vector, thus we write $\pi(a|s, \boldsymbol{\theta}) = \Pr\{A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$ for he probability that action a is taken at time t given that the environment is in state s at time t with parameter $\boldsymbol{\theta}$ in this chapter we consider methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter, these methods seek to maximize performance, so their updates approximate gradient ascent in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$, all methods that follow this general schema we call *policy gradient methods*, whether or not they also learn an approximate value function, methods that learn approximations to both policy and value functions are often called *actor-critic methods*, where 'actor' is a reference to the learned policy and 'critic' refers to the learned value function usually a state-value function

policy approximation and its advantages: pp.322, in practice to ensure exploration we generally require that the policy never becomes deterministic i.e. that $\pi(a|s, \boldsymbol{\theta}) \in (0, 1)$ for all s, a, $\boldsymbol{\theta}$, in this section we introduce the most common parameterization for discrete action spaces and point out the advantages it offers over action-value methods, if the action space is discrete and not too large, then a natural and common kind of parameterization is to form parameterized numerical preferences $h(s, a, \boldsymbol{\theta}) \in \mathbb{R}$ for each state-action pair, the actions with the highest preferences in each state are given the highest probabilities of being selected, for example according to an exponential soft-max distribution

$$\pi(a|s, \boldsymbol{\theta}) := \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}$$

we call this kind of policy parameterization soft-max in action preferences
pp.322, on advantage of parameterizing policies according to the soft-max in action preferences is that the approximate policy can approach a deterministic policy, whereas with ϵ-greedy action selection over action values there is always an ϵ probability of selecting a random action, selecting according to a soft-max distribution based on action values alone would not allow the policy to approach a deterministic policy, instead the action-value estimates would converge to their corresponding true values which would differ by a finite amount, translating to specific probabilities other than 0 and 1
pp.323, a second advantage of parameterizing policies according to the soft-max in action preferences is that it enables the selection of actions with arbitrary probabilities, action-value methods have no natural way of finding stochastic optimal policies whereas policy approximating methods can as shown in example 13.1
pp.324, perhaps the simplest advantage that policy parameterization may have over action-value parameterization is that the policy may be a simpler function to approximate, in which case a policy-based method will typically learn faster and yield a superior asymptotic policy, finally the choice of policy parameterization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the reinforcement learning system, this is often the most important reason for using a policy-based learning method

the policy gradient theorem: pp.324, in addition to the practical advantages of policy parameterization over ϵ-greedy action selection, there is also an important theoretical advantage, with continuous policy parametrization the action probabilities change smoothly as a function of the learned parameter, whereas in ϵ-greedy selection the action probabilities may change dramatically for an arbitrarily small change in the estimated action values if that change results in a different action having the maximal value, largely because of this stronger convergence guarantees are available for policy-gradient methods than for action-value methods, in particular it is the continuity of the policy dependence on the parameters that enables policy-gradient methods to approximate gradient ascent

in this section we treat the episodic case for which we define the performance measure as the value of the start state of the episode $J(\boldsymbol{\theta}) := v_{\pi_{\boldsymbol{\theta}}}(s_0)$, where $v_{\pi_{\boldsymbol{\theta}}}$ is the true value function for $\pi_{\boldsymbol{\theta}}$ the policy determined by $\boldsymbol{\theta}$, from here on in our discussion we will assume no discounting ($\gamma = 1$) for the episodic case

pp.324, with function approximation it may seem challenging to change the policy parameter in a way that ensures improvement, the problem is that performance depends on both the action selections and the distribution of states in which those selections are made, and that both of these are affected by the policy parameter; pp.325, fortunately there is an excellent theoretical answer to this challenge in the form of the *policy gradient theorem*, which provides an analytic expression for the gradient of performance with respect to the policy parameter that does not involve the derivative of the state distribution

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta})$$

in the episodic case the constant of proportionality is the average length of an episode, and in the continuous case it is 1 so that the relationship is actually an equality, the distribution μ here (as in Chapter 9 and 10) is the on-policy distribution under π

REINFORCE: Monte Carlo policy gradient: pp.326, stochastic gradient ascent requires a way to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter, the sample gradients need only be proportional to the gradient because any constant of proportionality can be absorbed into the step size α, the right-hand side of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right]\end{aligned}$$

this algorithm has been called an all-actions method because its update involves all of the actions

pp.327, the above equation involves an appropriate sum over actions, but each term is not weighted by $\pi(a|S_t,\boldsymbol{\theta})$ as is needed for an expectation under π, so we introduce such a weighting without changing the equality

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \mathbb{E}_\pi\left[\sum_a q_\pi(S_t,a)\nabla\pi(a|S_t,\boldsymbol{\theta})\right] \\
&= \mathbb{E}_\pi\left[\sum_a \pi(a|S_t,\boldsymbol{\theta})q_\pi(S_t,a)\frac{\nabla\pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})}\right] \\
&= \mathbb{E}_\pi\left[q_\pi(S_t,A_t)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right] && \text{replacing } a \text{ by the sample } A_t \sim \pi \\
&= \mathbb{E}_\pi\left[G_t\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}\right] && \text{because } \mathbb{E}_\pi[G_t|S_t,A_t] = q_\pi(S_t,A_t)
\end{aligned}$$

using this sample to instantiate our generic stochastic gradient ascent algorithm yields the REINFORCE update

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \alpha G_t\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} = \boldsymbol{\theta}_t + \alpha G_t \nabla \ln\pi(A_t|S_t,\boldsymbol{\theta})$$

each increment is proportional to the product of a return G_t and a vector which is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to state S_t (referred to as the eligibility vector, ref. pp.328), the update increases the parameter vector in this direction proportional to the return and inversely proportional to the action probability, the latter makes sense because otherwise actions that are selected frequently are at an advantage and might win out even if they do not yield the highest return
pp.327, REINFORCE uses the complete return from time t which includes all future rewards up until the end of the episode, in this sense REINFORCE is a Monte Carlo algorithm and is well defined only after the episode is completed
pp.329, as a stochastic gradient method REINFORCE has good theoretical convergence properties, because by construction the expected update over an episode is in the same direction as the performance gradient which assures an improvement in expected performance for sufficiently small α, as a Monte Carlo method REINFORCE may be of high variance and thus produce slow learning

REINFORCE with baseline: pp.329, the policy gradient theorem can be generalized to include a comparison of the action value to an arbitrary baseline $b(s)$

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s)\sum_a \left(q_\pi(s,a) - b(s)\right)\nabla\pi(a|s,\boldsymbol{\theta})$$

as long as the baseline does not vary with a the equation remains valid because the subtracted quantity is zero

$$\sum_a b(s)\nabla\pi(a|s,\boldsymbol{\theta}) = b(s)\nabla\sum_a \pi(a|s,\boldsymbol{\theta}) = b(s)\nabla 1 = 0$$

the update rule that includes a general baseline is

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \alpha\left(G_t - b(S_t)\right)\frac{\nabla\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}$$

because the baseline could be uniformly zero, this update is a strict generalization of REINFORCE, in general the baseline leaves the expected value of the update unchanged, but it can have a large effect on its variance, one natural choice for the baseline is an estimate of the state value $\hat{v}(S_t, \mathbf{w}_t)$, where $\mathbf{w} \in \mathbb{R}^d$ is a weight vector learned by one of the methods presented in previous chapters, because REINFORCE is a Monte Carlo method for learning the policy parameter $\boldsymbol{\theta}$, it seems natural to also use a Monte Carlo method to learn the state-value weights $\mathbf{w}$; pp.330, adding a baseline to REINFORCE can make it learn much faster

actor-critic methods: pp.331, in REINFORCE with baseline, the learned state-value function estimates the value of the first state of each state transition, this estimate sets a baseline for the subsequent return, but is made prior to the transition's action and thus cannot be used to assess that action, on the other hand in actor-critic methods the state-value function is applied also to the second state of the transition, the estimated value of the second state when discounted and added to the reward constitutes the one-step return $G_{t:t+1}$, which is a useful estimate of the actual return and thus is a way of assessing the action, when the state-value function is used to assess actions in this way it is called a *critic*, and the overall policy-gradient method is termed an *actor-critic* method, one-step actor-critic methods replace the full return of REINFORCE with the one-step return

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \alpha \left(G_{t:t+1} - \hat{v}(S_t, \mathbf{w})\right) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}$$

the main appeal of one-step methods is that they are fully online and incremental, they are a special case of the eligibility trace methods, the natural state-value-function learning method to pair with this is semi-gradient TD(0)
pp.332, the generalizations to the forward view of n-step methods and then to a λ-return algorithm are straightforward, the one-step return $G_{t:t+1}$ is merely replaced by $G_{t:t+n}$ or G_t^λ respectively

policy gradient for continuing problems: pp.333, for continuing problems without episode boundaries we need to define performance in terms of the average rate of reward per time step

$$J(\boldsymbol{\theta}) := \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r$$

where μ is the steady-state distribution under π, $\mu(s) := \lim_{t\to\infty} \Pr\{S_t = s|A_{0:t} \sim \pi\}$, which is assumed to exist and to be independent of S_0 (an ergodicity assumption), this is the special distribution under which if you select actions according to π you remain in the same distribution

$$\sum_s \mu(s) \sum_a \pi(a|s, \boldsymbol{\theta}) p(s'|s, a) = \mu(s') \qquad \text{for all } s' \in \mathcal{S}$$

pp.334, naturally in the continuing case we define values $v_\pi(s) := \mathbb{E}[G_t|S_t = s]$ and $q_\pi(s, a) := \mathbb{E}[G_t|S_t = s, A_t = a]$ with respect to the differential return

$$G_t := R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \cdots$$

with these alternate definitions the policy gradient theorem as given for the episodic case remains true for the continuing case, the forward and backward view equations also remain the same

policy parameterization for continuous actions: pp.335, instead of computing learned probabilities for each of the many actions we instead learn statistics of the probability distribution, to produce a policy parameterization the policy can be defined as the normal probability density over a real-valued scalar action

$$\pi(a|s,\boldsymbol{\theta}) := \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}\right)$$

pp.336, we divide the policy's parameter vector into two parts $\boldsymbol{\theta} = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma]^T$, one part to be used for the approximation of the mean and one part for the approximation of the standard deviation

$$\mu(s,\boldsymbol{\theta}) := \boldsymbol{\theta}_\mu^T \mathbf{x}_\mu(s) \qquad \sigma(s,\boldsymbol{\theta}) := \exp\left(\boldsymbol{\theta}_\sigma^T \mathbf{x}_\sigma(s)\right)$$

where $\mathbf{x}_\mu(s)$ and $\mathbf{x}_\sigma(s)$ are state feature vectors perhaps constructed by one of the methods described in Section 9.5, with these definitions all the algorithms described in the rest of this chapter can be applied to learn to select real-valued actions

summary: pp.337, methods that learn and store a policy parameter have many advantages, they can learn specific probabilities for taking the actions, they can learn appropriate levels of exploration and approach deterministic policies asymptotically, all these things are easy for policy-based methods but awkward or impossible for ϵ-greedy methods and for action-value methods in general
adding a state-value function as a baseline reduces REINFORCE's variance without introducing bias, if the state-value function is also used to assess the policy's action selections, then the value function is called a critic and the policy is called an actor, the overall method is called an actor-critic method, the critic introduces bias into the actor's gradient estimates, but is often desirable for the same reason that bootstrapping TD methods are often superior to Monte Carlo methods (substantially reduced variance)

7 Week 7 Articles

7.1 Sutton & Barto Reinforcement Learning Chapter 17.3

pp.464, it is somewhat surprising and not widely recognized that function approximation includes important aspects of partial observability, e.g. if there is a state variable that is not observable, then the parametrization can be chosen such that the approximate value does not depend on that state variable, because of this all the results obtained for the parametrized case apply to partial observability without change, in this sense the case of parametrized function approximation includes the case of partial observability, nevertheless there are many issues that cannot be investigated without a more explicit treatment of partial observability, there are four changes that would be needed to do so

1. pp.464, we would change the problem, the environment would emit not its states but only observations

2. pp.465, we can recover the idea of state from the sequence of observations and actions, to be a summary of the history a state must be a function of history $S_t = f(H_t)$, the summary would be informationally perfect if it retained all information about the history, in this case the state S_t and the function f are said to have the *Markov property*, a Markov state

summarizes all that is necessary for making any prediction, formally f is Markov if and only if for any test τ and for any histories h and h' that map to the same state under f, the test's probabilities given the two histories are equal

$$f(h) = f(h') \implies p(\tau|h) = p(\tau|h') \quad \forall h, h', \tau \in \{\mathcal{A} \times \mathcal{O}\}^*$$

if f is Markov then there is always a deterministic function π such that choosing $A_t := \pi(f(H_t))$ is an optimal policy

3. pp.465, computationally we want an f that can be compactly implemented with an incremental recursive update that computes S_{t+1} from S_t incorporating only the next increment of data A_t and O_{t+1}

   $$S_{t+1} := u(S_t, A_t, O_{t+1}) \quad \forall t \geq 0$$

   the function u is called the state-update function, a common strategy for finding a Markov state is to look for something compact that is recursively updatable and enables accurate short-term predictions, in fact it is only necessary to make accurate one-step predictions, because if an f is incrementally updatable, then it is Markov if and only if all one-step tests can be accurately predicted, that is if and only if

   $$f(h) = f(h') \implies \Pr\{O_{t+1} = o|H_t = h, A_t = a\} = \Pr\{O_{t+1} = o|H_t = h', A_t = a\}$$

   for all $h, h' \in \{\mathcal{A} \times \mathcal{O}\}^*$, $o \in \mathcal{O}$ and $a \in \mathcal{A}$
   pp.467, an example of obtaining Markov states through a state-update function is provided by the popular Bayesian approach known as *partially observable MDPs* or *POMDPs*, in this approach the environment is assumed to have a well defined latent state X_t that underlies and produces the environment's observations, but is never available to the agent, the natural Markov state S_t for a POMDP is the distribution over the latent states given the history called the *belief state* $S_t := \mathbf{s}_t \in [0, 1]^d$ with components

   $$\mathbf{s}_t[i] := \Pr\{X_t = i|H_t\} \quad \text{for all possible latent states } i \in \{1, 2, \ldots, d\}$$

   the belief state remains the same number of components even as t grows, it can also be incrementally updated by Bayes' rule

   $$u(\mathbf{s}, a, o)[y] := \frac{\sum_{x=1}^{d} \mathbf{s}[x] p(y, o|x, a)}{\sum_{x=1}^{d} \sum_{x'=1}^{d} \mathbf{s}[x] p(x', o|x, a)} \quad \forall a \in \mathcal{A}, o \in \mathcal{O}$$

   this approach is popular in theoretical work and has many significant applications, but its assumptions and computational complexity scale poorly, and we do not recommend it as an approach to artificial intelligence
   pp.467, another example of Markov states is provided by *predictive state representations* or *PSRs*, PSRs address the weakness of the POMDP approach that the semantics of its agent state S_t are grounded in the environment state X_t which is never observed, in PSRs and related approaches the semantics of the agent state is instead grounded in predictions about future observations and actions which are readily observable

4. pp.468, to approach artificial intelligence ambitiously we must embrace approximation, this is just as true for states as it is for value functions, we must accept and work with an approximate notion of state, the general idea is that a state that is good for some predictions

is also good for others—in particular that a Markov state sufficient for one-step predictions is also sufficient for all others, the guarantee provided by the perfect-but-impractical Markov property is replaced by the heuristic that what's good for some predictions may be good for others, both POMDP and PSR approaches can be applied with approximate states, the semantics of the state is often useful in forming the state-update function, however there is not a strong need for the state to be accurate with respect to its semantics in order to retain useful information, learning the state-update function for an approximate state is a major part of the representation learning problem as it arises in reinforcement learning

8 Week 8 Articles

8.1 Sutton & Barto Reinforcement Learning Chapter 17.2

pp.461, one popular idea of formulating multiple tasks with different time scales and different notions of choice and action is to formalize an MDP at a detailed level with a small time step, yet enable planning at higher levels using extended courses of action that correspond to many base-level time steps, to do this we need a notion of course of action that extends over many time steps and includes a notion of termination, a general way to formulate these two ideas is to define a pair of a policy π and a state-dependent termination function γ as a generalized notion of action termed an *option*, to execute an option $\omega = \langle \pi_\omega, \gamma_\omega \rangle$ at time t is to obtain the action to take A_t from $\pi_\omega(\cdot|S_t)$, then terminate at time $t+1$ with probability $1 - \gamma_\omega(S_{t+1})$, if the option does not terminate at $t+1$ then A_{t+1} is selected from $\pi_\omega(\cdot|S_{t+1})$ and the option terminates at $t+2$ with probability $1 - \gamma_\omega(S_{t+2})$ and so on until eventual termination, it is convenient to consider low-level actions to be special cases of options whose termination function is zero $\gamma_\omega(s) = 0$ for all $s \in \mathcal{S}^+$
pp.462, options are designed so that they are interchangable with low-level actions, for example the notion of an action-value function q_π naturally generalizes to an option-value function, we can also generalize the notion of policy to a hierarchical policy that selects from options rather than actions, where options when selected execute until termination, to generalize conventional action models to option models the appropriate model is again of two parts, one corresponding to the state transition resulting from executing the option and one corresponding to the expected cumulative reward along the way, the reward part of an option model is

$$r(s,\omega) := \mathbb{E}\left[R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots + \gamma^{\tau-1} R_\tau \middle| S_0 = s, A_{0:\tau-1} \sim \pi_\omega, \tau \sim \gamma_\omega\right]$$

where τ is the random time step at which the option terminates according to γ_ω, the state-transition part of an option model for option ω specifies for each state s that ω might start executing in and for each state s' that ω might terminate in

$$p(s'|s,\omega) := \sum_{k=1}^{\infty} \gamma^k \Pr\left\{S_k = s', \tau = k | S_0 = s, A_{0:k-1} \sim \pi_\omega, \tau \sim \gamma_\omega\right\}$$

the above definition of the transition part of an option model allows us to formulate Bellman equations and dynamic programming algorithms that apply to all options, for example the general Bellman equation for the state values of a hierarchical policy π is

$$v_\pi(s) = \sum_{\omega \in \Omega(s)} \pi(\omega|s) \left[r(s,\omega) + \sum_{s'} p(s'|s,\omega) v_\pi(s') \right]$$

and the value iteration algorithm with options is

$$v_{k+1}(s) := \max_{\omega \in \Omega(s)} \left[r(s,\omega) + \sum_{s'} p(s'|s,\omega) v_k(s') \right] \qquad \forall s \in \mathcal{S}$$

value iteration will converge to the best hierarchical policy limited to the restricted set of options, although this policy may be sub-optimal convergence can be much faster because fewer options are considered and because each option can jump over many time steps

9 Week 11 Articles

9.1 Sutton & Barto Reinforcement Learning Chapter 17.6

pp.476, in most current applications policies are learned from simulated experience instead of direct interaction with the real world, in addition to avoiding undesirable real-world consequences learning from simulated experience can make virtually unlimited data available generally at less cost than needed to obtain real experience, and because simulations typically run much faster than real time learning can often occur more quickly than if it relied on real experience
pp.477, reinforcement learning agents can discover unexpected ways to make their environments deliver reward, some of which might be undesirable or even dangerous, when we specify what we want a system to learn only indirectly as we do in designing a reinforcement learning system's reward signal, we will not know how closely the agent will fulfill our desire until its learning is complete, and anyone having experience with reinforcement learning has likely seen their systems discover unexpected ways to obtain a lot of reward
pp.477, another challenge if reinforcement learning agents are to act and learn in the real world is not just about what they might learn eventually, but about how they will behave while they are learning; pp.478, one of the most pressing areas for future reinforcement learning research is to adapt and extend methods developed in control engineering with the goal of making it acceptably safe to fully embed reinforcement learning agents into physical environments