Chapter 19

Reinforcement learning

Reinforcement learning (RL) is a sequential decision-making framework in which agents learn to perform actions in an environment with the goal of maximizing received rewards. For example, an RL algorithm might control the moves (actions) of a character (the agent) in a video game (the environment), aiming to maximize the score (the reward). In robotics, an RL algorithm might control the movements (actions) of a robot (the agent) in the world (the environment) to perform a task (earning a reward). In finance, an RL algorithm might control a virtual trader (the agent) who buys or sells assets (the actions) on a financial exchange (the environment) to maximize profit (the reward).

Consider learning to play chess. Here, there is a reward of $+1$, -1, or 0 at the end of the game if the agent wins, loses, or draws and 0 at every other time step. This illustrates the challenges of RL. First, the reward is sparse; here, we must play an entire game to receive feedback. Second, the reward is temporally offset from the action that caused it; a decisive advantage might be gained thirty moves before victory. We must associate the reward with this critical action. This is termed the *temporal credit assignment problem*. Third, the environment is stochastic; the opponent doesn't always make the same move in the same situation, so it's hard to know if an action was truly good or just lucky. Finally, the agent must balance exploring the environment (e.g., trying new opening moves) with exploiting what it already knows (e.g., sticking to a previously successful opening). This is termed the *exploration-exploitation trade-off*.

Reinforcement learning is an overarching framework that does not necessarily require deep learning. However, in practice, state-of-the-art systems often use deep networks. They encode the environment (the video game display, robot sensors, financial time series, or chessboard) and map this directly or indirectly to the next action (figure 1.13).

19.1 Markov decision processes, returns, and policies

Reinforcement learning maps observations of an environment to actions, aiming to maximize a numerical quantity that is connected to the rewards received. In the most common case, we learn a *policy* that maximizes the expected *return* in a *Markov decision process*. This section explains these terms.

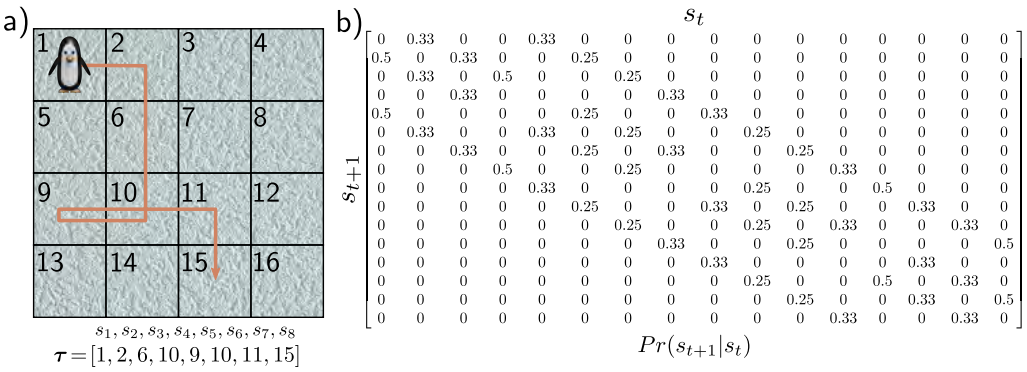

Figure 19.1 Markov process. A Markov process consists of a set of states and transition probabilities $Pr(s_{t+1}|s_t)$ that define the probability of moving to state s_{t+1} given the current state is s_t. a) The penguin can visit 16 different positions (states) on the ice. b) The ice is slippery, so at each time, it has an equal probability of moving to any adjacent state. For example, in position 6, it has a 25% chance of moving to states 2, 5, 7, and 10. A trajectory $\tau = [s_1, s_2, s_3, \ldots]$ from this process consists of a sequence of states.

19.1.1 Markov process

A *Markov process* assumes that the world is always in one of a set of possible states. The word *Markov* implies that the probability of being in a state depends only on the previous state and not on the states before. The changes between states are captured by the *transition probabilities* $Pr(s_{t+1}|s_t)$ of moving to the next state s_{t+1} given the current state s_t, where t indexes the time step. Hence, a Markov process is an evolving system that produces a sequence $s_1, s_2, s_3 \ldots$ of states (figure 19.1).

19.1.2 Markov reward process

A *Markov reward process* extends the Markov process to include a distribution $Pr(r_{t+1}|s_t)$ over the possible rewards r_{t+1} received at the next time step, given that we are in state s_t. This produces a sequence $s_1, r_2, s_2, r_3, s_3, r_4 \ldots$ of states and the associated rewards (figure 19.2). The Markov reward process also includes a *discount factor* $\gamma \in (0, 1]$ that is used to compute the *return* G_t at time t:

Problem 19.1

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{19.1}$$

The return is the sum of the cumulative discounted future rewards; it measures the future benefit of being on this trajectory. A discount factor of less than one makes rewards that are closer in time more valuable than rewards that are further away.

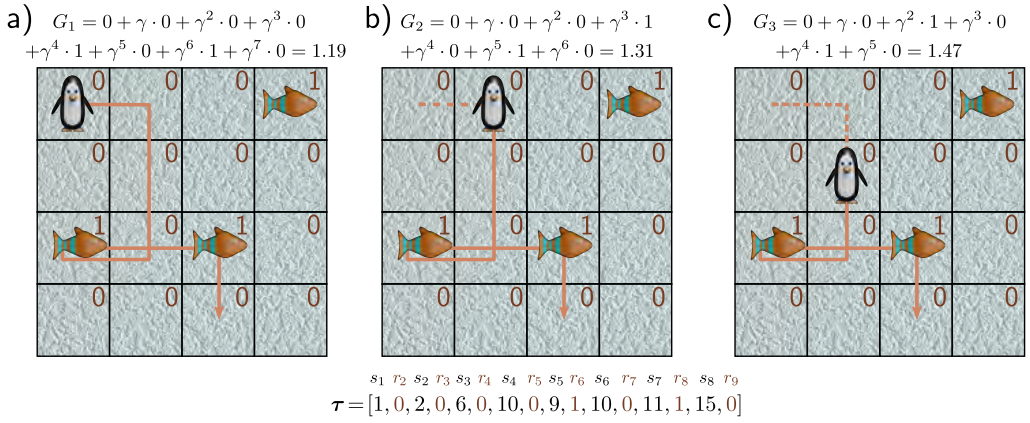

Figure 19.2 Markov reward process. This associates a distribution $Pr(r_{t+1}|s_t)$ of rewards r_{t+1} with each state s_t. a) Here, the rewards are deterministic; the penguin will receive a reward of $+1$ if it lands on a fish and 0 otherwise. The trajectory τ now consists of a sequence $s_1, r_2, s_2, r_3, s_3, r_4 \ldots$ of alternating states and rewards, terminating after eight steps. The return G_t of the sequence is the sum of discounted future rewards, here with discount factor $\gamma = 0.9$. b-c) As the penguin proceeds along the trajectory and gets closer to reaching the rewards, the return increases.

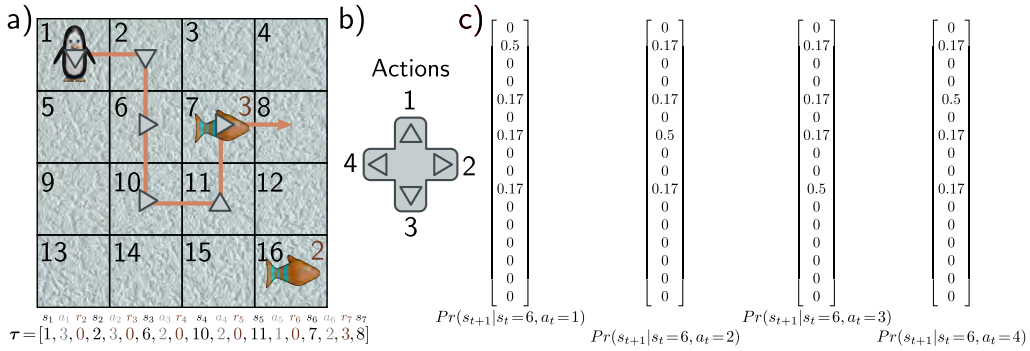

Figure 19.3 Markov decision process. a) The agent (penguin) can perform one of a set of actions in each state. b) Here, the four actions correspond to moving up, right, down, and left. c) For any state (here, state 6), the action changes the probability of moving to the next state. The penguin moves in the intended direction with 50% probability, but the ice is slippery, so it may slide to one of the other adjacent positions with equal probability. Accordingly, in panel (a), the action taken (gray arrows) doesn't always line up with the trajectory (orange line). In general, the action can also influence the probability of receiving rewards, but in this example the reward is the same regardless of the action, so $Pr(r_{t+1}|s_t, a_t) = Pr(r_{t+1}|s_t)$. The trajectory $\boldsymbol{\tau}$ from an MDP consists of a sequence $s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4 \ldots$ of alternating states s_t, actions a_t, and rewards, r_{t+1}. Note that here the penguin receives the reward when it *leaves* a state with a fish (i.e., the reward is received for passing through the fish square, regardless of whether the penguin arrived there intentionally or not).

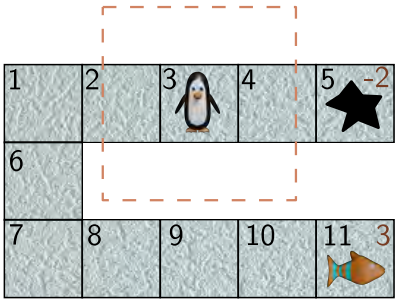

Figure 19.4 Partially observable Markov decision process (POMDP). In a POMDP, the agent does not have access to the entire state. Here, the penguin does not know the current state and can only see tiles in the vicinity (dashed box). Unfortunately, the true state (three) is indistinguishable from what it would see in state nine. In the first case, moving right leads to the hole in the ice (with -2 reward) and, in the latter, to the fish (with +3 reward).

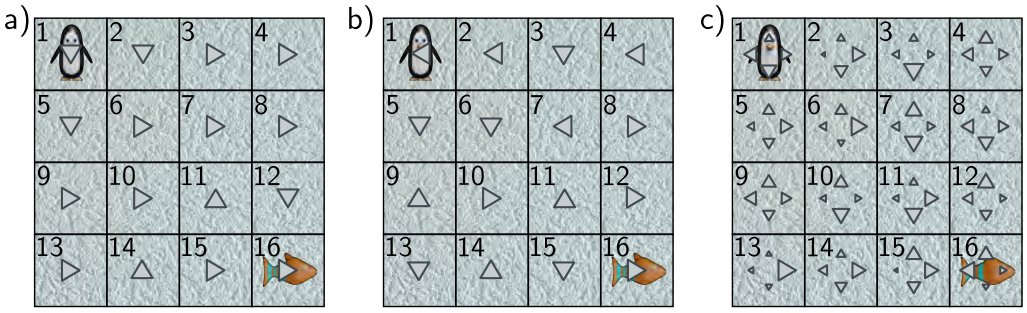

Figure 19.5 Policies. a) A deterministic policy always chooses the same action in each state (indicated by arrow). Some policies are better than others. This policy is not optimal but still generally steers the penguin from top-left to bottom-right where the reward lies. b) This policy is more random. c) A stochastic policy has a probability distribution over actions for each state (probability indicated by size of arrows). This has the advantage that the agent explores the states more thoroughly and can be necessary for optimal performance in partially observable Markov decision processes.

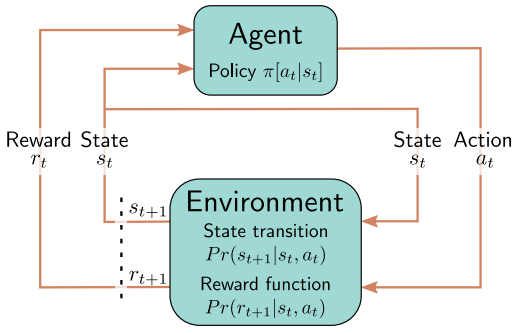

Figure 19.6 Reinforcement learning loop. The agent takes an action a_t at time t based on the state s_t, according to the policy $\pi[a_t|s_t]$. This triggers the generation of a new state s_{t+1} (via the state transition function) and a reward r_{t+1} (via the reward function). Both are passed back to the agent, which then chooses a new action.

19.1.3 Markov decision process

A *Markov decision process* or *MDP* adds a set of possible *actions* at each time step. The action a_t changes the transition probabilities, which are now written as $Pr(s_{t+1}|s_t, a_t)$. The rewards can also depend on the action and are now written as $Pr(r_{t+1}|s_t, a_t)$. An MDP produces a sequence $(s_1, a_1, r_2), (s_2, a_2, r_3), (s_3, a_3, r_4) \ldots$ of states s_t, actions a_t, and rewards r_{t+1} which are received at the subsequent time-step (figure 19.3). The entity that performs the actions is known as the *agent*.

19.1.4 Partially observable Markov decision process

In a *partially observable Markov decision process* or *POMDP*, the state is not directly visible (figure 19.4). Instead, the agent receives an observation o_t drawn from $Pr(o_t|s_t)$. Hence, a POMDP generates a sequence $s_1, o_1, a_1, r_2, s_2, o_2, a_2, r_3, o_3, a_3, s_3, r_4, \ldots$ of states, observations, actions, and rewards. In general, each observation will be more compatible with some states than others but insufficient to identify the state uniquely.

19.1.5 Policy

The rules that determine the agent's action for each state are known as the *policy* (figure 19.5). This may be stochastic (the policy defines a distribution over actions for each state) or deterministic (the agent always takes the same action in a given state). A stochastic policy $\pi[a|s]$ returns a probability distribution over each possible action a for state s, from which a new action is sampled. A deterministic policy $\pi[a|s]$ returns one for the action a that is chosen for state s and zero otherwise. A *stationary* policy depends only on the current state. A *non-stationary* policy also depends on the time step.

Notebook 19.1 Markov decision processes

The environment and the agent form a loop (figure 19.6). The agent receives the state s_t and reward r_t from the last time step. Based on this, it can modify the policy $\pi[a_t|s_t]$ if desired and choose the next action a_t. The environment then advances to the next state according to $Pr(s_{t+1}|s_t, a_t)$ and issues a reward according to $Pr(r_{t+1}|s_t, a_t)$.

19.2 Expected return

The previous section introduced the Markov decision process and the idea of an agent carrying out actions according to a policy. We want to choose a policy that maximizes the expected return. In this section, we make this idea mathematically precise. To do that, we assign a *value* to each state s_t and state-action pair $\{s_t, a_t\}$.

19.2.1 State and action values

The return G_t depends on the state s_t and the policy $\pi[a|s]$. From this state, the agent will pass through a sequence of states, taking actions and receiving rewards. This

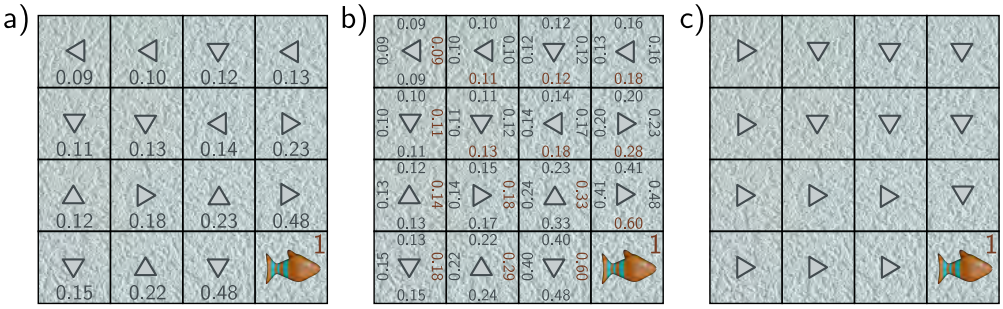

Figure 19.7 State and action values. a) The value $v[s_t|\pi]$ of a state s_t (number at each position) is the expected return for this state for a given policy π (gray arrows). It is the average sum of discounted rewards received over many trajectories started from this state. Here, states closer to the fish are more valuable. b) The value $q[s_t, a_t, \pi]$ of an action a_t in state s_t (four numbers at each position/state corresponding to four actions) is the expected return given that this particular action is taken in this state. In this case, it gets larger as we get closer to the fish and is larger for actions that head in the direction of the fish. c) If we know the action values at a state, then the policy can be modified so that it chooses the maximum of these values (red numbers in panel b).

sequence differs every time the agent starts in the same place since, in general, the policy $\pi[a_t|s_t]$, the state transitions $Pr(s_{t+1}|s_t, a_t)$, and the rewards issued $Pr(r_{t+1}|s_t, a_t)$ are all stochastic.

We can characterize how "good" a state is under a given policy π by considering the expected return $v[s_t|\pi]$. This is the return that would be received on average from sequences that start from this state and is termed the *state value* or *state-value function* (figure 19.7a):

Appendix C.2 Expectation

$$v[s_t|\pi] = \mathbb{E}\Big[G_t|s_t, \pi\Big]. \tag{19.2}$$

Informally, the state value tells us the *long-term* reward we can expect on average if we start in this state and follow the specified policy thereafter. It is highest for states where it's probable that subsequent transitions will bring large rewards soon (assuming the discount factor γ is less than one).

Similarly, the *action value* or *state-action value function* $q[s_t, a_t|\pi]$ is the expected return from executing action a_t in state s_t (figure 19.7b):

$$q[s_t, a_t|\pi] = \mathbb{E}\Big[G_t|s_t, a_t, \pi\Big]. \tag{19.3}$$

The action value tells us the long-term reward we can expect on average if we start in this state, take this action, and follow the specified policy thereafter. Through this quantity, reinforcement learning algorithms connect future rewards to current actions (i.e., resolve the temporal credit assignment problem).

19.2.2 Optimal policy

We want a policy that maximizes the expected return. For MDPs (but not POMDPs), there is always a deterministic, stationary policy that maximizes the value of every state. If we know this optimal policy, then we get the optimal state-value function $v^*[s_t]$:

$$v^*[s_t] = \max_{\pi} \Big[\mathbb{E}\Big[G_t | s_t, \pi \Big] \Big]. \tag{19.4}$$

Similarly, the optimal state-action value function is obtained under the optimal policy:

$$q^*[s_t, a_t] = \max_{\pi} \Big[\mathbb{E}\Big[G_t | s_t, a_t, \pi \Big] \Big]. \tag{19.5}$$

Turning this on its head, if we knew the optimal action-values $q^*[s_t, a_t]$, then we could derive the optimal policy by choosing the action a_t with the highest value (figure 19.7c):[1]

$$\pi[a_t|s_t] \leftarrow \underset{a_t}{\text{argmax}} \Big[q^*[s_t, a_t] \Big]. \tag{19.6}$$

Indeed, some reinforcement learning algorithms are based on alternately estimating the action values and the policy (see section 19.3).

19.2.3 Bellman equations

We may not know the state values $v[s_t]$ or action values $q[s_t, a_t]$ for any policy.[2] However, we know that they must be consistent with one another, and it's easy to write relations between these quantities. The state value $v[s_t]$ can be found by taking a weighted sum of the action values $q[s_t, a_t]$, where the weights depend on the probability under the policy $\pi[a_t|s_t]$ of taking that action (figure 19.8):

$$v[s_t] = \sum_{a_t} \pi[a_t|s_t] q[s_t, a_t]. \tag{19.7}$$

Similarly, the value of an action is the immediate reward $r_{t+1} = r[s_t, a_t]$ generated by taking the action, plus the value $v[s_{t+1}]$ of being in the subsequent state s_{t+1} discounted by γ (figure 19.9).[3] Since the assignment of s_{t+1} is not deterministic, we weight the values $v[s_{t+1}]$ according to the transition probabilities $Pr(s_{t+1}|s_t, a_t)$:

$$q[s_t, a_t] = r[s_t, a_t] + \gamma \cdot \sum_{s_{t+1}} Pr(s_{t+1}|s_t, a_t) v[s_{t+1}]. \tag{19.8}$$

Substituting equation 19.8 into equation 19.7 provides a relation between the state value at time t and $t+1$:

[1] The notation $\pi[a_t|s_t] \leftarrow a$ in equations 19.6, 19.12, and 19.13 means set $\pi[a_t|s]$ to one for action a and $\pi[a_t|s]$ to zero for other actions.

[2] For simplicity, we will just write $v[s_t]$ and $q[s_t, a_t]$ instead of $v[s_t|\pi]$ and $q[s_t, a_t|\pi]$ from now on.

[3] We also assume from now on that the rewards are deterministic and can be written as $r[s_t, a_t]$.

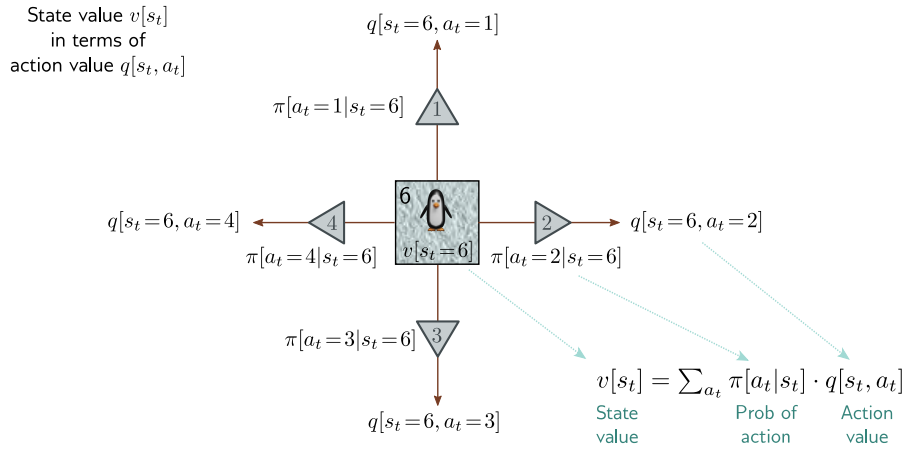

Figure 19.8 Relationship between state values and action values. The value of state six $v[s_t=6]$ is a weighted sum of the action values $q[s_t=6, a_t]$ at state six, where the weights are the policy probabilities $\pi[a_t|s_t=6]$ of taking that action.

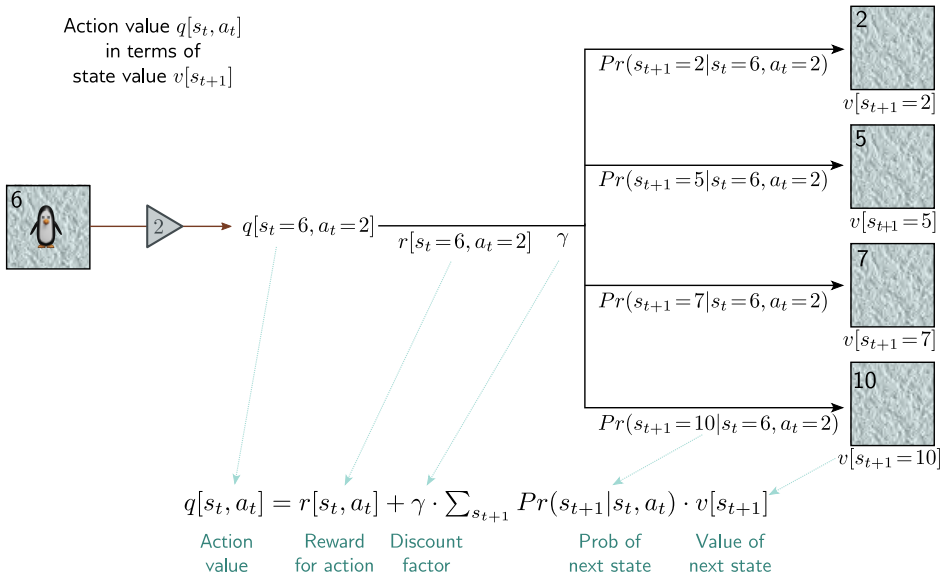

Figure 19.9 Relationship between action values and state values. The value $q[s_t=6, a_t=2]$ of taking action two in state six is the reward $r[s_t=6, a_t=2]$ from taking that action plus a weighted sum of the discounted values $v[s_{t+1}]$ of being in successor states, where the weights are the transition probabilities $Pr(s_{t+1}|s_t=6, a_t=2)$. The Bellman equations chain this relation with that of figure 19.8 to link the current and next (i) state values and (ii) action values.

$$v[s_t] = \sum_{a_t} \pi[a_t|s_t] \left(r[s_t, a_t] + \gamma \cdot \sum_{s_{t+1}} Pr(s_{t+1}|s_t, a_t) v[s_{t+1}] \right). \tag{19.9}$$

Similarly, substituting equation 19.7 into equation 19.8 provides a relation between the action value at time t and $t+1$:

$$q[s_t, a_t] = r[s_t, a_t] + \gamma \cdot \sum_{s_{t+1}} Pr(s_{t+1}|s_t, a_t) \left(\sum_{a_{t+1}} \pi[a_{t+1}|s_{t+1}] q[s_{t+1}, a_{t+1}] \right). \tag{19.10}$$

The latter two relations are the *Bellman equations* and are the backbone of many RL methods. In short, they say that the state (action) values have to be self-consistent. Consequently, when we update an estimate of one state (action) value, this will have a ripple effect that causes modifications to all the others.

19.3 Tabular reinforcement learning

Tabular RL algorithms (i.e., those that don't rely on function approximation) are divided into *model-based* and *model-free* methods. *Model-based methods*[4] use the MDP structure explicitly and find the best policy from the transition matrix $Pr(s_{t+1}|s_t, a_t)$ and reward structure $r[s, a]$. If these are known, this is a straightforward optimization problem that can be tackled using *dynamic programming*. If they are unknown, they can (in principle) be estimated from observed MDP trajectories.[5]

Conversely, *model-free* methods assume that the transition matrix and reward structure of the underlying MDP are unknown. These methods fall into two families:

1. *Value estimation* approaches estimate the optimal state-action value function and then assign the policy according to the action in each state with the greatest value.
2. *Policy estimation* approaches directly estimate the optimal policy using a gradient descent technique without the intermediate steps of estimating the model or values.

Within each family, *Monte Carlo* methods simulate many trajectories through the MDP for a given policy to gather information about how to improve this policy. Sometimes it is not feasible or practical to simulate many trajectories before updating the policy. *Temporal difference (TD)* methods update the policy *while* the agent traverses the MDP.

We now briefly describe dynamic programming methods, Monte Carlo value estimation methods, and TD value estimation methods. Section 19.4 describes how deep networks have been used in TD value estimation methods. We return to policy estimation in section 19.5.

[4]The term *model* refers here to the MDP and not a machine learning model.

[5]In RL, a *trajectory* is an observed sequence of states, rewards, and actions. A *rollout* is a *simulated* trajectory. An *episode* is a trajectory that starts in an initial state and ends in a terminal state (e.g., a full game of chess starting from the standard opening position and ending in a win, lose, or draw.)

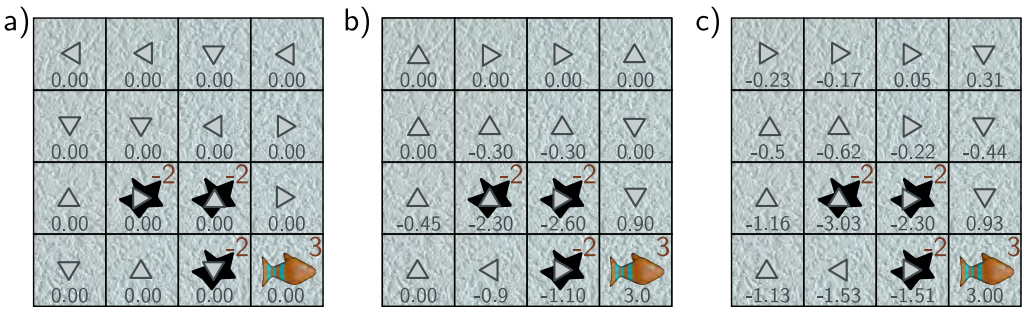

Figure 19.10 Dynamic programming. a) The state values are initialized to zero, and the policy (arrows) is chosen randomly. b) The state values are updated to be consistent with their neighbors (equation 19.11, shown after two iterations). The policy is updated to move the agent to states with the highest value (equation 19.12). c) After several iterations, the algorithm converges to the optimal policy, in which the penguin tries to avoid the holes and reach the fish.

19.3.1 Dynamic programming

Dynamic programming algorithms assume we have *perfect* knowledge of the transition and reward structure. In this respect, they are distinguished from most RL algorithms which observe the agent interacting with the environment to gather information about these quantities indirectly.

The state values $v[s]$ are initialized arbitrarily (usually to zero). The deterministic policy $\pi[a|s]$ is also initialized (e.g., by choosing a random action for each state). The algorithm then alternates between iteratively computing the state values for the current policy (*policy evaluation*) and improving that policy (*policy improvement*).

Policy evaluation: We sweep through the states s_t, updating their values:

$$v[s_t] \leftarrow \sum_{a_t} \pi[a_t|s_t] \left(r[s_t, a_t] + \gamma \cdot \sum_{s_{t+1}} Pr(s_{t+1}|s_t, a_t) v[s_{t+1}] \right), \tag{19.11}$$

where s_{t+1} is the successor state and $Pr(s_{t+1}|s_t, a_t)$ is the state transition probability. Each update makes $v[s_t]$ consistent with the value at the successor state s_{t+1} using the Bellman equation for state values (equation 19.9). This is termed *bootstrapping*.

Policy improvement: To update the policy, we greedily choose the action that maximizes the value for each state:

$$\pi[a_t|s_t] \leftarrow \underset{a_t}{\text{argmax}} \left[r[s_t, a_t] + \gamma \cdot \sum_{s_{t+1}} Pr(s_{t+1}|s_t, a_t) v[s_{t+1}] \right]. \tag{19.12}$$

This is guaranteed to improve the policy according to the *policy improvement theorem*.

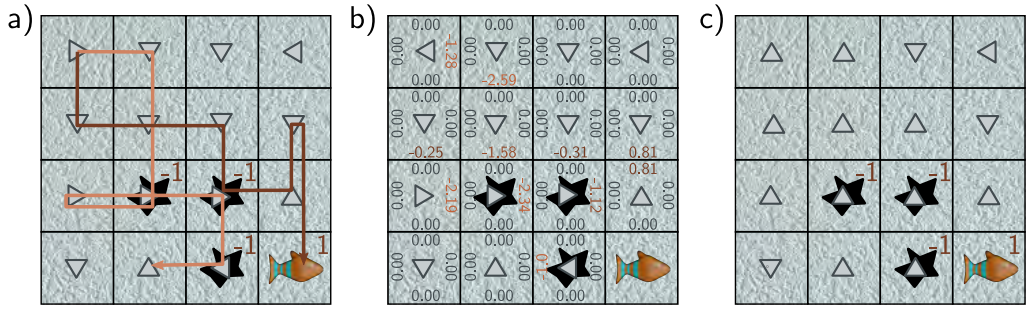

Figure 19.11 Monte Carlo methods. a) The policy (arrows) is initialized randomly. The MDP is repeatedly simulated, and the trajectories of these episodes are stored (orange and brown paths represent two trajectories). b) The action values are empirically estimated based on the observed returns averaged over these trajectories. In this case, the action values were all initially zero and have been updated where an action was observed. c) The policy can then be updated according to the action which received the best (or least bad) reward.

Problems 19.2–19.3

These two steps are iterated until the policy converges (figure 19.10).

There are many variations of this approach. In *policy iteration*, the policy evaluation step is iterated until convergence before policy improvement. The values can be updated either in place or synchronously in each sweep. In *value iteration*, the policy evaluation procedure sweeps through the values just once before policy improvement. *Asynchronous* dynamic programming algorithms don't have to systematically sweep through all the values at each step but can update a subset of the states in place in an arbitrary order.

Notebook 19.2 Dynamic programming

19.3.2 Monte Carlo methods

Unlike dynamic programming algorithms, Monte Carlo methods don't assume knowledge of the MDP's transition probabilities and reward structure. Instead, they gain experience by repeatedly sampling trajectories from the MDP and observing the rewards. They alternate between computing the action values (based on this experience) and updating the policy (based on the action values).

To estimate the action values $q[s, a]$, a series of *episodes* are run. Each starts with a given state and action and thereafter follows the current policy, producing a series of actions, states, and rewards (figure 19.11a). The action value for a given state-action pair under the current policy is estimated as the average of the empirical returns (i.e., cumulative sums of time-discounted rewards) that follow each time this pair occurs (figure 19.11b). Then the policy is updated by choosing the action with the maximum value at every state (figure 19.11c):

$$\pi[a|s] \leftarrow \underset{a}{\text{argmax}}\Big[q[s, a]\Big]. \tag{19.13}$$

This is an *on-policy* method; the current best policy is used to guide the agent through the environment. This policy is based on the observed action values in every state, but of course, it's not possible to estimate the value of actions that haven't been used, and there is nothing to encourage the algorithm to explore these. One solution is to use *exploring starts*. Here, episodes with all possible state-action pairs are initiated, so every combination is observed at least once. However, this is impractical if the number of states is large or the starting point cannot be controlled. A different approach is to use an *epsilon greedy* policy, in which a random action is taken with probability ϵ, and the optimal action is allotted the remaining probability $1-\epsilon$. The choice of ϵ trades off exploitation and exploration. Here, an on-policy method will seek the best policy from this epsilon-greedy family, which will *not* generally be the best overall policy.

Problem 19.4

Conversely, in *off-policy* methods, the optimal policy π (the *target policy*) is learned based on episodes generated by a different *behavior policy* π'. Typically, the target policy is deterministic, and the behavior policy is stochastic (e.g., an epsilon-greedy policy). Hence, the behavior policy can explore the environment, but the learned target policy remains efficient. Some off-policy methods explicitly use importance sampling (section 17.8.1) to estimate the action value under policy π using samples from π'. Others, such as Q-learning (described in the next section), estimate the values based on the greedy action, even though this is not necessarily what was chosen.

Notebook 19.3 Monte Carlo methods

19.3.3 Temporal difference methods

Dynamic programming methods use a bootstrapping process to update the values to make them self-consistent under the current policy. Monte Carlo methods sample the MDP to acquire information. Temporal difference (TD) methods combine both bootstrapping and sampling. However, unlike Monte Carlo methods, they update the values and policy *while* the agent traverses the states of the MDP instead of afterward.

SARSA (State-Action-Reward-State-Action) is an on-policy algorithm with update:

$$q[s_t, a_t] \leftarrow q[s_t, a_t] + \alpha\Big(r[s_t, a_t] + \gamma \cdot q[s_{t+1}, a_{t+1}] - q[s_t, a_t]\Big), \tag{19.14}$$

where $\alpha \in \mathbb{R}^+$ is the learning rate. The bracketed term is called the *TD error* and measures the consistency between the estimated action value $q[s_t, a_t]$ and the estimate $r[s_t, a_t]+\gamma \cdot q[s_{t+1}, a_{t+1}]$ after taking a single step.

By contrast, *Q-Learning* is an off-policy algorithm with update (figure 19.12):

$$q[s_t, a_t] \leftarrow q[s_t, a_t] + \alpha\Big(r[s_t, a_t] + \gamma \cdot \max_a\big[q[s_{t+1}, a]\big] - q[s_t, a_t]\Big), \tag{19.15}$$

where now the choice of action at each step is derived from a different behavior policy π'.

Notebook 19.4 Temporal difference methods

In both cases, the policy is updated by taking the maximum of the action values at each state (equation 19.13). It can be shown that these updates are contraction mappings (see equation 16.20); the action values will eventually converge, assuming that every state-action pair is visited an infinite number of times.

Problem 19.5

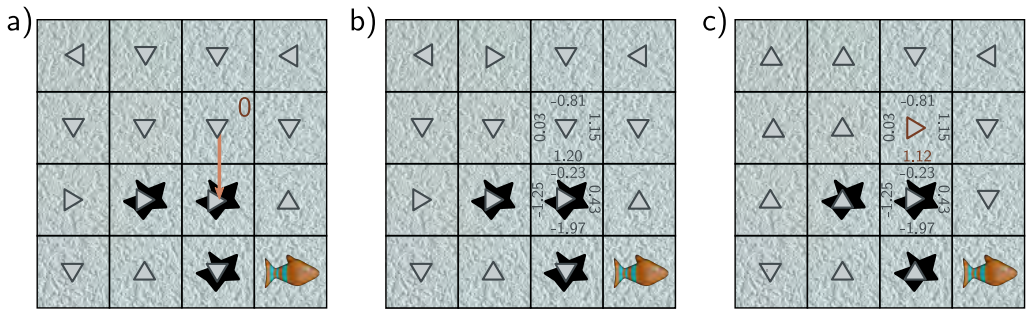

Figure 19.12 Q-learning. a) The agent starts in state s_t and takes action $a_t = 2$ according to the policy. It does not slip on the ice, so it moves downward, receiving reward $r[s_t, a_t] = 0$ for leaving the original state. b) The maximum action value at the new state is found (here 0.43). c) The action value for action 2 in the original state is updated to 1.12 based on the current estimate of the maximum action value at the subsequent state, the reward, discount factor $\gamma = 0.9$, and learning rate $\alpha = 0.1$. This changes the highest action value at the original state, so the policy changes.

19.4 Fitted Q-learning

The tabular Monte Carlo and TD algorithms described above repeatedly traverse the entire MDP and update the action values. However, this is only practical if the state-action space is small. Unfortunately, this is rarely the case; even for the constrained environment of a chessboard, there are more than 10^{40} possible legal states.

In *fitted Q-learning*, the discrete representation $q[s_t, a_t]$ of the action values is replaced by a machine learning model $q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]$, where now the state is represented by a vector $\mathbf{s}_t$ rather than just an index. We then define a least squares loss based on the consistency of adjacent action values (similar to the loss in Q-learning, see equation 19.15):

$$L[\boldsymbol{\phi}] = \left(r[\mathbf{s}_t, a_t] + \gamma \cdot \max_a\Big[q[\mathbf{s}_{t+1}, a, \boldsymbol{\phi}]\Big] - q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]\right)^2, \tag{19.16}$$

which in turn leads to the update:

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \alpha\left(r[\mathbf{s}_t, a_t] + \gamma \cdot \max_a\Big[q[\mathbf{s}_{t+1}, a, \boldsymbol{\phi}]\Big] - q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]\right)\frac{\partial q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]}{\partial \boldsymbol{\phi}}. \tag{19.17}$$

Fitted Q-learning differs from Q-Learning in that convergence is no longer guaranteed. A change to the parameters potentially modifies both the target $r[\mathbf{s}_t, a_t] + \gamma \cdot \max_{a_{t+1}}[q[\mathbf{s}_{t+1}, a_{t+1}, \boldsymbol{\phi}]]$ (the maximum value may change) and the prediction $q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]$. This can be shown both theoretically and empirically to damage convergence.

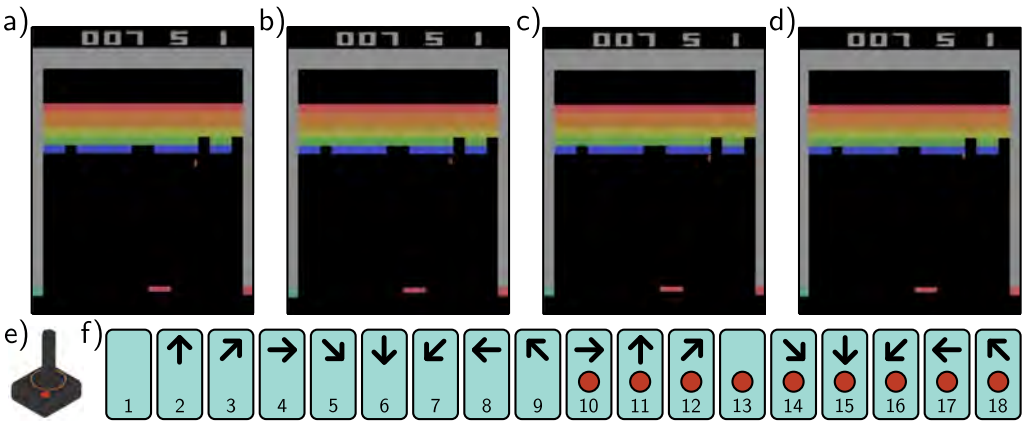

Figure 19.13 Atari Benchmark. The Atari benchmark consists of 49 Atari 2600 games, including Breakout (pictured), Pong, and various shoot-em-up, platform, and other types of games. a-d) Even for games with a single screen, the state is not fully observable from a single frame because the velocity of the objects is unknown. Consequently, it is usual to use several adjacent frames (here, four) to represent the state. e) The action simulates the user input via a joystick. f) There are eighteen actions corresponding to eight directions of movement or no movement, and for each of these nine cases, the button being pressed or not.

19.4.1 Deep Q-networks for playing ATARI games

Deep networks are ideally suited to making predictions from a high-dimensional state space, so they are a natural choice for the model in fitted Q-learning. In principle, they could take both state and action as input and predict the values, but in practice, the network takes only the state and simultaneously predicts the values for each action.

The *Deep Q-Network* was a breakthrough reinforcement learning architecture that exploited deep networks to learn to play ATARI 2600 games. The observed data comprises 220×160 images with 128 possible colors at each pixel (figure 19.13). This was reshaped to size 84×84, and only the brightness value was retained. Unfortunately, the full state is not observable from a single frame. For example, the velocity of game objects is unknown. To help resolve this problem, the network ingests the last four frames at each time step to form $\mathbf{s}_t$. It maps these frames through three convolutional layers followed by a fully connected layer to predict the value of every action (figure 19.14).

Several modifications were made to the standard training procedure. First, the rewards (which were driven by the score in the game) were clipped to -1 for a negative change and $+1$ for a positive change. This compensates for the wide variation in scores between different games and allows the same learning rate to be used. Second, the system exploited *experience replay*. Rather than update the network based on the tuple $<\mathbf{s}_t, a_t, r_{t+1}, \mathbf{s}_{t+1}>$ at the current step or with a batch of the last I tuples, all recent

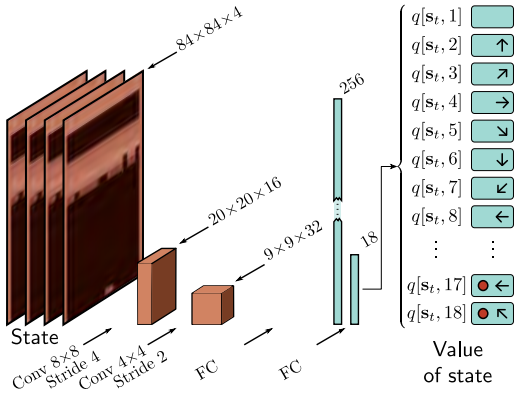

Figure 19.14 Deep Q-network architecture. The input $\mathbf{s}_t$ consists of four adjacent frames of the ATARI game. Each is resized to 84×84 and converted to grayscale. These frames are represented as four channels and processed by an 8×8 convolution with stride four, followed by a 4×4 convolution with stride 2, followed by two fully connected layers. The final output predicts the action value $q[\mathbf{s}_t, a_t]$ for each of the 18 actions in this state.

tuples were stored in a buffer. This buffer was sampled randomly to generate a batch at each step. This approach reuses data samples many times and reduces correlations between the samples in the batch that arise due to the similarity of adjacent frames.

Finally, the issue of convergence in fitted Q-Networks was tackled by fixing the target parameters to values $\boldsymbol{\phi}^-$ and only updating them periodically. This gives the update:

$$\boldsymbol{\phi} \leftarrow \boldsymbol{\phi} + \alpha\left(r[\mathbf{s}_t, a_t] + \gamma \cdot \max_a\Big[q[\mathbf{s}_{t+1}, a, \boldsymbol{\phi}^-]\Big] - q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]\right)\frac{\partial q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]}{\partial \boldsymbol{\phi}}. \tag{19.18}$$

Now the network no longer chases a moving target and is less prone to oscillation.

Using these and other heuristics and with an ϵ-greedy policy, Deep Q-Networks performed at a level comparable to a professional game tester across a set of 49 games using the same network architecture (trained separately for each game). It should be noted that the training process was data-intensive. It took around 38 full days of experience to learn each game. In some games, the algorithm exceeded human performance. On other games like "Montezuma's Revenge," it barely made any progress. This game features sparse rewards and multiple screens with quite different appearances.

19.4.2 Double Q-learning and double deep Q-networks

One potential flaw of Q-Learning is that the maximization over the actions in the update:

$$q[s_t, a_t] \leftarrow q[s_t, a_t] + \alpha\Big(r[s_t, a_t] + \gamma \cdot \max_a\big[q[s_{t+1}, a]\big] - q[s_t, a_t]\Big) \tag{19.19}$$

leads to a systematic bias in the estimated action values $q[s_t, a_t]$. Consider two actions that provide the same average reward, but one is stochastic and the other deterministic. The stochastic reward will exceed the average roughly half of the time and be chosen by the maximum operation, causing the corresponding action value $q[s_t, a_t]$ to be overestimated. A similar argument can be made about random inaccuracies in the output of the network $q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]$ or random initializations of the q-function.

The underlying problem is that the same network both selects the target (by the maximization operation) and updates the value. Double Q-Learning tackles this problem by training two models $q_1[s_t, a_t, \pi_1]$ and $q_2[s_t, a_t, \pi_2]$ simultaneously:

$$\begin{aligned} q_1[s_t, a_t] &\leftarrow q_1[s_t, a_t] + \alpha\Big(r[s_t, a_t] + \gamma \cdot q_2\left[s_{t+1}, \underset{a}{\operatorname{argmax}}\Big[q_1[s_{t+1}, a]\Big]\right] - q_1[s_t, a_t]\Big) \\ q_2[s_t, a_t] &\leftarrow q_2[s_t, a_t] + \alpha\Big(r[s_t, a_t] + \gamma \cdot q_1\left[s_{t+1}, \underset{a}{\operatorname{argmax}}\Big[q_2[s_{t+1}, a]\Big]\right] - q_2[s_t, a_t]\Big). \end{aligned} \tag{19.20}$$

Now the choice of the target and the target itself are decoupled, which helps prevent these biases. In practice, new tuples $< s, a, r, s' >$ are randomly assigned to update one model or another. This is known as *double Q-learning*. *Double deep Q-networks* or *double DQNs* use deep networks $q[\mathbf{s}_t, a_t, \boldsymbol{\phi}_1]$ and $q[\mathbf{s}_t, a_t, \boldsymbol{\phi}_2]$ to estimate the action values, and the update becomes:

$$\begin{aligned} \boldsymbol{\phi}_1 &\leftarrow \boldsymbol{\phi}_1 + \alpha\left(r[\mathbf{s}_t, a_t] + \gamma \cdot q\left[\mathbf{s}_{t+1}, \underset{a}{\operatorname{argmax}}\Big[q[\mathbf{s}_{t+1}, a, \boldsymbol{\phi}_1]\Big], \boldsymbol{\phi}_2\right] - q[\mathbf{s}_t, a_t, \boldsymbol{\phi}_1]\right)\frac{\partial q[\mathbf{s}_t, a_t, \boldsymbol{\phi}_1]}{\partial \boldsymbol{\phi}_1} \\ \boldsymbol{\phi}_2 &\leftarrow \boldsymbol{\phi}_2 + \alpha\left(r[\mathbf{s}_t, a_t] + \gamma \cdot q\left[\mathbf{s}_{t+1}, \underset{a}{\operatorname{argmax}}\Big[q[\mathbf{s}_{t+1}, a, \boldsymbol{\phi}_2]\Big], \boldsymbol{\phi}_1\right] - q[\mathbf{s}_t, a_t, \boldsymbol{\phi}_2]\right)\frac{\partial q[\mathbf{s}_t, a_t, \boldsymbol{\phi}_2]}{\partial \boldsymbol{\phi}_2}. \end{aligned} \tag{19.21}$$

19.5 Policy gradient methods

Q-learning estimates the action values first and then uses these to update the policy. Conversely, *policy-based methods* directly learn a stochastic policy $\pi[a_t|\mathbf{s}_t, \boldsymbol{\theta}]$. This is a function with trainable parameters $\boldsymbol{\theta}$ that maps a state $\mathbf{s}_t$ to a distribution $Pr(a_t|\mathbf{s}_t)$ over actions a_t from which we can sample. In MDPs, there is always an optimal deterministic policy. However, there are three reasons to use a stochastic policy:

1. A stochastic policy naturally helps with exploration of the space; we are not obliged to take the best action at each time step.
2. The loss changes smoothly as we modify a stochastic policy. This means we can use gradient descent methods even though the rewards are discrete. This is similar to using maximum likelihood in (discrete) classification problems. The loss changes smoothly as the model parameters change to make the true class more likely.
3. The MDP assumption is often incorrect; we usually don't have complete knowledge of the state. For example, consider an agent navigating in an environment where it can only observe nearby locations (e.g., figure 19.4). If two locations look identical, but the nearby reward structure is different, a stochastic policy allows the possibility of taking different actions until this ambiguity is resolved.

19.5.1 Derivation of gradient update

Consider a trajectory $\boldsymbol{\tau} = [\mathbf{s}_1, a_1, \mathbf{s}_2, a_2, \ldots, \mathbf{s}_T, a_T]$ through an MDP. The probability of this trajectory $Pr(\boldsymbol{\tau}|\boldsymbol{\theta})$ depends on both the state evolution function $Pr(\mathbf{s}_{t+1}|\mathbf{s}_t, a_t)$ and the current stochastic policy $\pi[a_t|\mathbf{s}_t, \boldsymbol{\theta}]$:

$$Pr(\boldsymbol{\tau}|\boldsymbol{\theta}) = Pr(\mathbf{s}_1)\prod_{t=1}^{T}\pi[a_t|\mathbf{s}_t, \boldsymbol{\theta}]Pr(\mathbf{s}_{t+1}|\mathbf{s}_t, a_t). \tag{19.22}$$

Policy gradient algorithms aim to maximize the expected return $r[\boldsymbol{\tau}]$ over many such trajectories:

$$\boldsymbol{\theta} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\Big[\mathbb{E}_{\boldsymbol{\tau}}\Big[r[\boldsymbol{\tau}]\Big]\Big] = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\Big[\int Pr(\boldsymbol{\tau}|\boldsymbol{\theta})r[\boldsymbol{\tau}]d\boldsymbol{\tau}\Big], \tag{19.23}$$

where the return is the sum of all the rewards received along the trajectory.

To maximize this quantity, we use the gradient ascent update:

$$\begin{aligned}\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \alpha\cdot\frac{\partial}{\partial\boldsymbol{\theta}}\int Pr(\boldsymbol{\tau}|\boldsymbol{\theta})r[\boldsymbol{\tau}]d\boldsymbol{\tau} \\ &= \boldsymbol{\theta} + \alpha\cdot\int\frac{\partial Pr(\boldsymbol{\tau}|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}r[\boldsymbol{\tau}]d\boldsymbol{\tau},\end{aligned} \tag{19.24}$$

where α is the learning rate.

We want to approximate this integral with a sum over empirically observed trajectories. These are drawn from the distribution $Pr(\boldsymbol{\tau}|\boldsymbol{\theta})$, so to make progress, we multiply and divide the integrand by this distribution:

$$\begin{aligned}\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \alpha\cdot\int\frac{\partial Pr(\boldsymbol{\tau}|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}r[\boldsymbol{\tau}]d\boldsymbol{\tau} \\ &= \boldsymbol{\theta} + \alpha\cdot\int Pr(\boldsymbol{\tau}|\boldsymbol{\theta})\frac{1}{Pr(\boldsymbol{\tau}|\boldsymbol{\theta})}\frac{\partial Pr(\boldsymbol{\tau}|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}r[\boldsymbol{\tau}]d\boldsymbol{\tau} \\ &\approx \boldsymbol{\theta} + \alpha\cdot\frac{1}{I}\sum_{i=1}^{I}\frac{1}{Pr(\boldsymbol{\tau}_i|\boldsymbol{\theta})}\frac{\partial Pr(\boldsymbol{\tau}_i|\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}r[\boldsymbol{\tau}_i].\end{aligned} \tag{19.25}$$

This equation has a simple interpretation (figure 19.15); the update changes the parameters $\boldsymbol{\theta}$ to increase the likelihood $Pr(\boldsymbol{\tau}_i|\boldsymbol{\theta})$ of an observed trajectory $\boldsymbol{\tau}_i$ in proportion to the reward $r[\boldsymbol{\tau}_i]$ from that trajectory. However, it also normalizes by the probability of observing that trajectory in the first place to compensate for the fact that some trajectories are observed more often than others. If a trajectory is already common and yields high rewards, then we don't need to change much. The biggest updates will come from trajectories that are uncommon but create large rewards.

We can simplify this expression using the *likelihood ratio identity*:

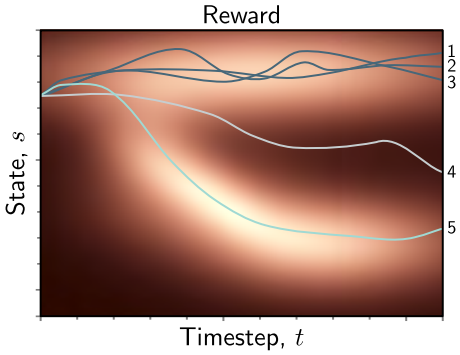

Figure 19.15 Policy gradients. Five episodes for the same policy (brighter indicates higher reward). Trajectories 1, 2, and 3 generate consistently high rewards, but similar trajectories already frequently occur with this policy, so there is no need to change. Conversely, trajectory 4 receives low rewards, so the policy should be modified to avoid producing similar trajectories. Trajectory 5 receives high rewards *and* is unusual. This will cause the largest change to the policy under equation 19.25.

$$\frac{\partial \log[\mathrm{f}[z]]}{\partial z} = \frac{1}{f[z]}\frac{\partial \mathrm{f}[z]}{\partial z}, \tag{19.26}$$

which yields the update:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{I}\sum_{i=1}^{I} \frac{\partial \log\big[Pr(\boldsymbol{\tau}_i|\boldsymbol{\theta})\big]}{\partial \boldsymbol{\theta}} r[\boldsymbol{\tau}_i]. \tag{19.27}$$

The log probability $\log[Pr(\boldsymbol{\tau}|\boldsymbol{\theta})]$ of a trajectory is given by:

$$\begin{aligned}\log[Pr(\boldsymbol{\tau}|\boldsymbol{\theta})] &= \log\Big[Pr(\mathbf{s}_1)\prod_{t=1}^{T}\pi[a_t|\mathbf{s}_t,\boldsymbol{\theta}]Pr(\mathbf{s}_{t+1}|\mathbf{s}_t,a_t)\Big] \\ &= \log\big[Pr(\mathbf{s}_1)\big] + \sum_{t=1}^{T}\log\big[\pi[a_t|\mathbf{s}_t,\boldsymbol{\theta}]\big] + \sum_{t=1}^{T}\log\big[Pr(\mathbf{s}_{t+1}|\mathbf{s}_t,a_t)\big],\end{aligned} \tag{19.28}$$

and noting that only the center term depends on $\boldsymbol{\theta}$, we can rewrite the update from equation 19.27 as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{I}\sum_{i=1}^{I}\sum_{t=1}^{T} \frac{\partial \log\big[\pi[a_{it}|\mathbf{s}_{it},\boldsymbol{\theta}]\big]}{\partial \boldsymbol{\theta}} r[\boldsymbol{\tau}_i], \tag{19.29}$$

where $\mathbf{s}_{it}$ is the state at time t in episode i, and a_{it} is the action taken at time t in episode i. Note that the terms relating to the state evolution $Pr(\mathbf{s}_{t+1}|\mathbf{s}_t,a_t)$ disappear from this formulation. It follows that this parameter update does not assume a Markov time evolution process.

We can further simplify this by noting that:

$$r[\boldsymbol{\tau}_i] = \sum_{t=1}^{T} r_{i,t+1} = \sum_{k=1}^{t} r_{i,k+1} + \sum_{k=t}^{T} r_{i,k+1}, \tag{19.30}$$

where r_{it} is the reward at time t in the i^{th} episode. It can (non-obviously) be proved that the first term (the rewards before time t) does not affect the update from time t, so we can write:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{I}\sum_{i=1}^{I}\sum_{t=1}^{T}\frac{\partial \log\left[\pi[a_{it}|\mathbf{s}_{it},\boldsymbol{\theta}]\right]}{\partial\boldsymbol{\theta}}\sum_{k=t}^{T} r_{i,k+1}. \tag{19.31}$$

19.5.2 REINFORCE algorithm

REINFORCE is an early policy gradient algorithm that exploits this result and incorporates discounting. It is a Monte Carlo method that generates episodes $\boldsymbol{\tau}_i = [\mathbf{s}_{i1}, a_{i1}, r_{i2}, \mathbf{s}_{i2}, a_{i2}, r_{i3}, \ldots, r_{iT}]$ based on the current policy $\pi[a|\mathbf{s},\boldsymbol{\theta}]$. For discrete actions, this policy could be determined by a neural network $\pi[\mathbf{s}|\boldsymbol{\theta}]$, which takes the current state $\mathbf{s}$ and returns one output for each possible action. These outputs are passed through a softmax function to create a distribution over actions, which is sampled at each time step.

For each episode i, we loop through each step t and calculate the empirical discounted return for the partial trajectory $\boldsymbol{\tau}_{it}$ that starts at time t:

$$r[\boldsymbol{\tau}_{it}] = \sum_{k=t+1}^{T}\gamma^{k-1} r_{i,k}, \tag{19.32}$$

and then we update the parameters for each time step t in each trajectory:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{\partial \log\left[\pi_{a_{it}}[\mathbf{s}_{it},\boldsymbol{\theta}]\right]}{\partial\boldsymbol{\theta}} r[\boldsymbol{\tau}_{it}] \qquad \forall\, i,t, \tag{19.33}$$

where $\pi_{a_t}[\mathbf{s}_t,\boldsymbol{\theta}]$ is the probability of a_t produced by the neural network given the current state $\mathbf{s}_t$ and parameters $\boldsymbol{\theta}$, and α is the learning rate.

19.5.3 Baselines

Policy gradient methods exhibit high variance; many episodes may be needed to get stable estimates of the derivatives. One way to reduce this variance is to subtract a baseline b from the trajectory returns $r[\boldsymbol{\tau}]$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{I}\sum_{i=1}^{I}\sum_{t=1}^{T}\frac{\partial \log\left[\pi_{a_{it}}[\mathbf{s}_{it},\boldsymbol{\theta}]\right]}{\partial\boldsymbol{\theta}}\left(r[\boldsymbol{\tau}_{it}] - b\right). \tag{19.34}$$

Problem 19.6

As long as the baseline b doesn't depend on the actions:

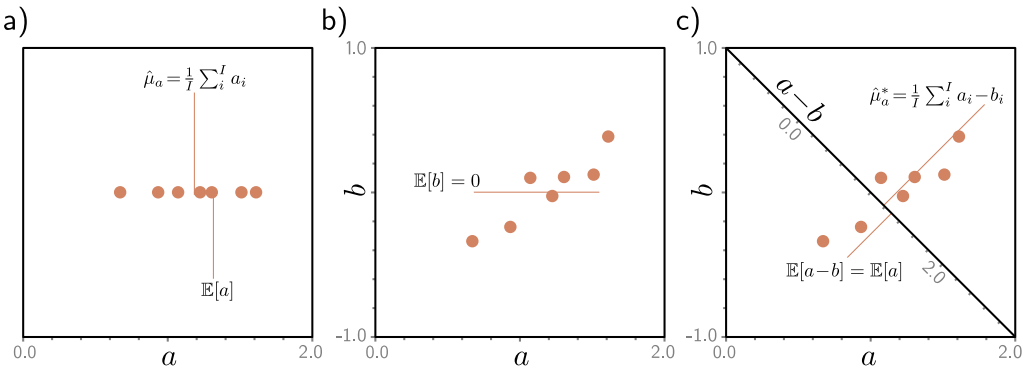

Figure 19.16 Decreasing variance of estimates using control variates. a) Consider trying to estimate $\mathbb{E}[a]$ from a small number of samples. The estimate (the mean of the samples) will vary based on the number of samples and the variance of those samples. b) Now consider observing another variable b that co-varies with a and has $\mathbb{E}[b] = 0$ and the same variance as a. c) The variance of the samples of $a - b$ is much less than that of a, but the expected value $\mathbb{E}[a - b] = \mathbb{E}[a]$, so we get an estimator with lower variance.

$$\mathbb{E}_{\boldsymbol{\tau}}\left[\sum_{t=1}^{T} \frac{\partial \log\left[\pi_{a_{it}}[\mathbf{s}_{it}, \boldsymbol{\theta}]\right]}{\partial \boldsymbol{\theta}} \cdot b\right] = 0, \tag{19.35}$$

and the expected value will not change. However, if the baseline co-varies with irrelevant factors that add uncertainty, then subtracting it reduces the variance (figure 19.16). This is a special case of the method of *control variates* (see problem 19.7).

Notebook 19.5 Control variates

Problem 19.7

This raises the question of how we should choose b. We can find the value of b that minimizes the variance by writing an expression for the variance, taking the derivative with respect to b, setting the result to zero, and solving to yield:

Problem 19.8

$$b = \sum_i \frac{\sum_{t=1}^{T}\left(\partial \log\left[\pi_{a_{it}}[\mathbf{s}_{it}, \boldsymbol{\theta}]\right]/\partial \boldsymbol{\theta}\right)^2 r[\boldsymbol{\tau}_{it}]}{\sum_{t=1}^{T}\left(\partial \log\left[\pi_{a_{it}}[\mathbf{s}_{it}, \boldsymbol{\theta}]\right]/\partial \boldsymbol{\theta}\right)^2}. \tag{19.36}$$

In practice, this is often approximated as:

$$b = \frac{1}{I}\sum_i r[\boldsymbol{\tau}_i]. \tag{19.37}$$

Subtracting this baseline factors out variance that might occur when the returns $r[\boldsymbol{\tau}_i]$ from all trajectories are greater than is typical but only because they happen to pass through states with higher than average returns *whatever* actions are taken.

19.5.4 State-dependent baselines

A better option is to use a baseline $b[\mathbf{s}_{it}]$ that depends on the current state $\mathbf{s}_{it}$.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{I}\sum_{i=1}^{I}\sum_{t=1}^{T} \frac{\partial \log\left[\pi_{a_{it}}[\mathbf{s}_{it}, \boldsymbol{\theta}]\right]}{\partial \boldsymbol{\theta}} \left(r[\boldsymbol{\tau}_{it}] - b[\mathbf{s}_{it}]\right). \tag{19.38}$$

Here, we are compensating for variance introduced by some states having greater overall returns than others, whichever actions we take.

A sensible choice is the expected future reward based on the current state, which is just the state value $v[\mathbf{s}]$. In this case, the difference between the empirically observed rewards and the baseline is known as the *advantage estimate*. Since we are in a Monte Carlo context, this can be parameterized by a neural network $b[\mathbf{s}] = v[\mathbf{s}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$, which we can fit to the observed returns using least squares loss:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I}\sum_{t=1}^{T}\left(v[\mathbf{s}_{it}, \boldsymbol{\phi}] - \sum_{j=t}^{T} r_{i,j+1}\right)^2. \tag{19.39}$$

19.6 Actor-critic methods

Actor-critic algorithms are temporal difference (TD) policy gradient algorithms. They can update the parameters of the policy network at each step. This contrasts with the Monte Carlo REINFORCE algorithm, which *must* wait for one or more episodes to complete before updating the parameters.

In the TD approach, we do not have access to the future rewards $r[\boldsymbol{\tau}_t] = \sum_{k=t}^{T} r_k$ along this trajectory. Actor-critic algorithms approximate the sum over all the future rewards with the observed current reward plus the discounted value of the next state:

$$r[\boldsymbol{\tau}_{it}] \approx r_{i,t+1} + \gamma \cdot v[\mathbf{s}_{i,t+1}, \boldsymbol{\phi}]. \tag{19.40}$$

Here the value $v[\mathbf{s}_{i,t+1}, \boldsymbol{\phi}]$ is estimated by a second neural network with parameters $\boldsymbol{\phi}$.

Substituting this into equation 19.38 gives the update:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{I}\sum_{i=1}^{I}\sum_{t=1}^{T} \frac{\partial \log\left[Pr(a_{it}|\mathbf{s}_{it}, \boldsymbol{\theta})\right]}{\partial \boldsymbol{\theta}} \Big(r_{i,t+1} + \gamma \cdot v[\mathbf{s}_{i,t+1}, \boldsymbol{\phi}] - v[\mathbf{s}_{i,t}, \boldsymbol{\phi}]\Big). \tag{19.41}$$

Concurrently, we update the parameters $\boldsymbol{\phi}$ by bootstrapping using the loss function:

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I}\sum_{t=1}^{T}\left(r_{i,t+1} + \gamma \cdot v[\mathbf{s}_{i,t+1}, \boldsymbol{\phi}] - v[\mathbf{s}_{i,t}, \boldsymbol{\phi}]\right)^2. \tag{19.42}$$

The policy network $\pi[\mathbf{s}_t, \boldsymbol{\theta}]$ that predicts $Pr(a|\mathbf{s}_t)$ is termed the *actor*. The value network $v[\mathbf{s}_t, \boldsymbol{\phi}]$ is termed the *critic*. Often the same network represents both actor and

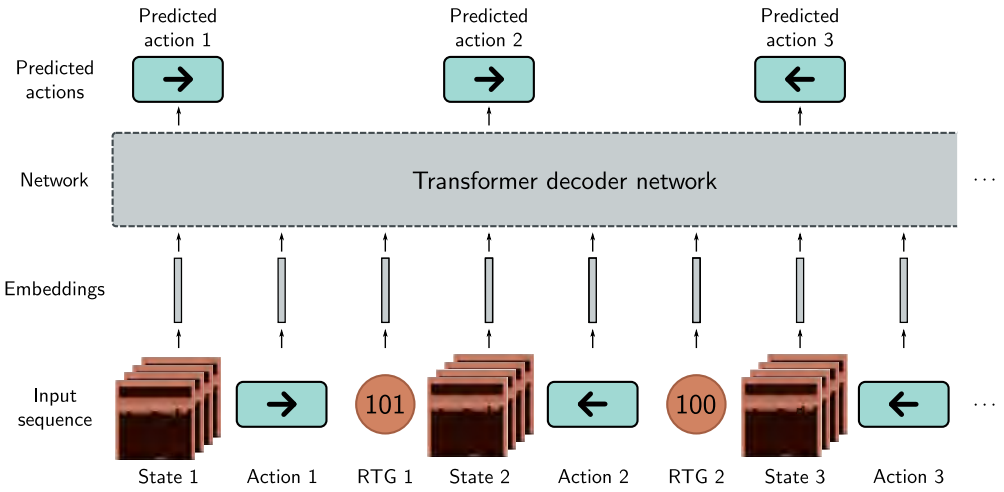

Figure 19.17 Decision transformer. The decision transformer treats offline reinforcement learning as a sequence prediction task. The input is a sequence of states, actions, and returns-to-go (remaining rewards in the episode), each of which is mapped to a fixed-size embedding. At each time step, the network predicts the next action. During testing, the returns-to-go are unknown; in practice, an initial estimate is made from which subsequent observed rewards are subtracted.

the critic, with two sets of outputs that predict the policy and the values, respectively. Note that although actor-critic methods can update the policy parameters at each step, this is rarely done in practice. The agent typically collects a batch of experience over many time steps before the policy is updated.

19.7 Offline reinforcement learning

Interaction with the environment is at the core of reinforcement learning. However, there are some scenarios where it is not practical to send a naïve agent into an environment to explore the effect of different actions. This may be because erratic behavior in the environment is dangerous (e.g., driving autonomous vehicles) or because data collection is time-consuming or expensive (e.g., making financial trades).

However, it *is* possible to gather historical data from human agents in both cases. *Offline RL* or *batch RL* aims to learn how to take actions that maximize rewards on future episodes by observing past sequences $\mathbf{s}_1, a_1, r_2, \mathbf{s}_2, a_2, r_3, \ldots$, without ever interacting with the environment. It is distinct from *imitation learning*, a related technique that (i) does not have access to the rewards and (ii) attempts to replicate the performance of a historical agent rather than improve it.

Although there are offline RL methods based on Q-Learning and policy gradients,

this paradigm opens up new possibilities. In particular, we can treat this as a sequence learning problem, in which the goal is to predict the next action, given the history of states, rewards, and actions. The *decision transformer* exploits a transformer decoder framework (section 12.7) to make these predictions (figure 19.17).

However, the goal is to predict actions based on *future rewards*, and these are not captured in a standard $\mathbf{s}, a, r$ sequence. Hence, the decision transformer replaces the reward r_t with the *returns-to-go* $R_{t:T} = \sum_{t'=t}^{T} r_{t'}$ (i.e., the sum of the empirically observed future rewards). The remaining framework is very similar to a standard transformer decoder. The states, actions, and returns-to-go are converted to fixed-size embeddings via learned mappings. For Atari games, the state embedding might be converted via a convolutional network similar to that in figure 19.14. The embeddings for the actions and returns-to-go can be learned in the same way as word embeddings (figure 12.9). The transformer is trained with masked self-attention and position embeddings.

This formulation is natural during training but poses a quandary during inference because we don't know the returns-to-go. This can be resolved by using the desired total return at the first step and decrementing this as rewards are received. For example, in an Atari game, the desired total return would be the total score required to win.

Decision transformers can also be fine-tuned from online experience and hence learn over time. They have the advantage of dispensing with most of the reinforcement learning machinery and its associated instability and replacing this with standard supervised learning. Transformers can learn from enormous quantities of data and integrate information across large time contexts (making the temporal credit assignment problem more tractable). This represents an intriguing new direction for reinforcement learning.

19.8 Summary

Reinforcement learning is a sequential decision-making framework for Markov decision processes and similar systems. This chapter reviewed tabular approaches to RL, including dynamic programming (in which the environment model is known), Monte Carlo methods (in which multiple episodes are run and the action values and policy subsequently changed based on the rewards received), and temporal difference methods (in which these values are updated while the episode is ongoing).

Deep Q-Learning is a temporal difference method where deep neural networks are used to predict the action value for every state. It can train agents to perform Atari 2600 games at a level similar to humans. Policy gradient methods directly optimize the policy rather than assigning values to actions. They produce stochastic policies, which are important when the environment is partially observable. The updates are noisy, and many refinements have been introduced to reduce their variance.

Offline reinforcement learning is used when we cannot interact with the environment but must learn from historical data. The decision transformer leverages recent progress in deep learning to build a model of the state-action-reward sequence and predict the actions that will maximize the rewards.

Notes

Sutton & Barto (2018) cover tabular reinforcement learning methods in depth. Li (2017), Arulkumaran et al. (2017), François-Lavet et al. (2018), and Wang et al. (2022c) all provide overviews of deep reinforcement learning. Graesser & Keng (2019) is an excellent introductory resource that includes Python code.

Landmarks in deep reinforcement learning: Most landmark achievements of reinforcement learning have been in either video games or real-world games since these provide constrained environments with limited actions and fixed rules. Deep Q-Learning (Mnih et al., 2015) achieved human-level performance across a benchmark of ATARI games. AlphaGo (Silver et al., 2016) beat the world champion at Go. This game was previously considered very difficult for computers to play. Berner et al. (2019) built a system that beat the world champion team in the five vs. five-player game *Defense of the Ancients 2*, which requires cooperation across players. Ye et al. (2021) built a system that could beat humans on Atari games with limited data (in contrast to previous systems, which need much more experience than humans). More recently, the Cicero system demonstrated human-level performance in the game *Diplomacy* which requires natural language negotiations and coordination between players (FAIR, 2022).

RL has also been applied successfully to combinatorial optimization problems (see Mazyavkina et al., 2021). For example, Kool et al. (2019) learned a model that performed similarly to the best heuristics for the traveling salesman problem. Recently, AlphaTensor (Fawzi et al., 2022) treated matrix multiplication as a game and learned faster ways to multiply matrices using fewer multiplication operations. Since deep learning relies heavily on matrix multiplication, this is one of the first examples of self-improvement in AI.

Classical reinforcement learning methods: Very early contributions to the theory of MDPs were made by Thompson (1933) and Thompson (1935). The Bellman recursions were introduced by Bellman (1966). Howard (1960) introduced policy iteration. Sutton & Barto (2018) identify the work of Andreae (1969) as being the first to describe RL using the MDP formalism.

The modern era of reinforcement learning arguably originated in the Ph.D. theses of Sutton (1984) and Watkins (1989). Sutton (1988) introduced the term temporal difference learning. Watkins (1989) and Watkins & Dayan (1992) introduced Q-Learning and showed that it converges to a fixed point by Banach's theorem because the Bellman operator is a contraction mapping. Watkins (1989) made the first explicit connection between dynamic programming and reinforcement learning. SARSA was developed by Rummery & Niranjan (1994). Gordon (1995) introduced *fitted Q-learning* in which a machine learning model is used to predict the action value for each state-action pair. Riedmiller (2005) introduced *neural-fitted Q-learning*, which used a neural network to predict all the action values at once from a state. Early work on Monte Carlo methods was carried out by Singh & Sutton (1996), and the exploring starts algorithm was introduced by Sutton & Barto (1999). Note that this is an extremely cursory summary of more than fifty years of work. A much more thorough treatment can be found in Sutton & Barto (2018).

Deep Q-Networks: Deep Q-Learning was devised by Mnih et al. (2015) and is an intellectual descendent of neural-fitted Q-learning. It exploited the then-recent successes of convolutional networks to develop a fitted Q-Learning method that could achieve human-level performance on a benchmark of ATARI games. Deep Q-Learning suffers from the *deadly triad issue* (Sutton & Barto, 2018): training can be unstable in any scheme that incorporates (i) bootstrapping, (ii) off-policy learning, and (iii) function approximation. Much subsequent work has aimed to make training more stable. Mnih et al. (2015) introduced the experience replay buffer (Lin, 1992), which was subsequently improved by Schaul et al. (2016) to favor more important tuples and hence increase learning speed. This is termed *prioritized experience replay*.

The original Q-Learning paper concatenated four frames so the network could observe the velocities of objects and make the underlying process closer to fully observable. Hausknecht & Stone (2015) introduced *deep recurrent Q-learning*, which used a recurrent network architecture that only ingested a single frame at a time because it could "remember" the previous states. Van Hasselt (2010) identified the systematic overestimation of the state values due to the max operation and proposed double Q-Learning in which two models are trained simultaneously to remedy this. This was subsequently applied in the context of deep Q-learning (Van Hasselt et al., 2016), although its efficacy has since been questioned (Hessel et al., 2018). Wang et al. (2016) introduced *deep dueling networks* in which two heads of the same network predict (i) the state value and (ii) the *advantage* (relative value) of each action. The intuition here is that sometimes it is the state value that is important, and it doesn't matter much which action is taken, and decoupling these estimates improves stability.

Fortunato et al. (2018) introduced *noisy deep Q-Networks*, in which some weights in the Q-Network are multiplied by noise to add stochasticity to the predictions and encourage exploration. The network can learn to decrease the magnitudes of the noise over time as it converges to a sensible policy. Distributional DQN (Bellemare et al., 2017a; Dabney et al., 2018 following Morimura et al., 2010) aims to estimate more complete information about the distribution of returns than just the expectation. This potentially allows the network to mitigate against worst-case outcomes and can also improve performance, as predicting higher moments provides a richer training signal. *Rainbow* (Hessel et al., 2018) combined six improvements to the original deep Q-learning algorithm, including dueling networks, distributional DQN, and noisy DQN, to improve both the training speed and the final performance on the ATARI benchmark.

Policy gradients: Williams (1992) introduced the REINFORCE algorithm. The term "policy gradient method" dates to Sutton et al. (1999). Konda & Tsitsiklis (1999) introduced the actor-critic algorithm. Decreasing the variance by using different baselines is discussed in Greensmith et al. (2004) and Peters & Schaal (2008). It has since been argued that the value baseline primarily reduces the aggressiveness of the updates rather than their variance (Mei et al., 2022).

Policy gradients have been adapted to produce deterministic policies (Silver et al., 2014; Lillicrap et al., 2016; Fujimoto et al., 2018). The most direct approach is to maximize over the possible actions, but if the action space is continuous, this requires an optimization procedure at each step. The *deep deterministic policy gradient* algorithm (Lillicrap et al., 2016) moves the policy in the direction of the gradient of the action value (implying the use of an actor-critic method).

Modern policy gradients: We introduced policy gradients in terms of the parameter update. However, they can also be viewed as optimizing a surrogate loss based on importance sampling of the expected rewards, using trajectories from the current policy parameters. This view allows us to take multiple optimization steps validly. However, this can cause very large policy updates. Overstepping is a minor problem in supervised learning, as the trajectory can be corrected later. However, in RL, it affects future data collection and can be extremely destructive.

Several methods have been proposed to moderate these updates. *Natural policy gradients* (Kakade, 2001) are based on natural gradients (Amari, 1998), which modify the descent direction by the Fisher information matrix. This provides a better update which is less likely to get stuck in local plateaus. However, the Fisher matrix is impractical to compute in models with many parameters. In *trust-region policy optimization* or *TRPO* (Schulman et al., 2015), the surrogate objective is maximized subject to a constraint on the KL divergence between the old and new policies. Schulman et al. (2017) propose a simpler formulation in which this KL divergence appears as a regularization term. The regularization weight is adapted based on the distance between the KL divergence and a target indicating how much we want the policy to change. *Proximal policy optimization* or *PPO* (Schulman et al., 2017) is an even simpler approach in which the loss is clipped to ensure smaller updates.

Actor-critic: In the actor-critic algorithm (Konda & Tsitsiklis, 1999) described in section 19.6, the critic used a 1-step estimator. It's also possible to use k-step estimators (in which we

observe k discounted rewards and approximate subsequent rewards with an estimate of the state value). As k increases, the variance of the estimate increases, but the bias decreases. *Generalized advantage estimation* (Schulman et al., 2016) weights together estimates from many steps and parameterizes the weighting by a single term that trades off the bias and the variance. Mnih et al. (2016) introduced *asynchronous actor-critic* or *A3C* in which multiple agents are run independently in parallel environments and update the same parameters. Both the policy and value function are updated every T time steps using a mix of k-step returns. Wang et al. (2017) introduced several methods designed to make asynchronous actor-critic more efficient. *Soft actor-critic* (Haarnoja et al., 2018b) adds an entropy term to the cost function, which encourages exploration and reduces overfitting as the policy is encouraged to be less confident.

Offline RL: In offline reinforcement learning, the policy is learned by observing the behavior of other agents, including the rewards they receive, *without* the ability to change the policy. It is related to imitation learning, where the goal is to copy the behavior of another agent without access to rewards (see Hussein et al., 2017). One approach is to treat offline RL in the same way as off-policy reinforcement learning. However, in practice, the distributional shift between the observed and applied policy manifests in overly optimistic estimates of the action value and poor performance (see Fujimoto et al., 2019; Kumar et al., 2019a; Agarwal et al., 2020). Conservative Q-learning (Kumar et al., 2020b) learns conservative, lower-bound estimates of the value function by regularizing the Q-values. The decision transformer (Chen et al., 2021c) is a simple approach to offline learning that takes advantage of the well-studied self-attention architecture. It can subsequently be fine-tuned with online training (Zheng et al., 2022).

Reinforcement learning and chatbots: Chatbots can be trained using a technique known as *reinforcement learning with human feedback* or *RLHF* (Christiano et al., 2018; Stiennon et al., 2020). For example, *InstructGPT* (the forerunner of ChatGPT, Ouyang et al., 2022) starts with a standard transformer decoder model. This is then fine-tuned based on prompt-response pairs where the response was written by human annotators. During this training step, the model is optimized to predict the next word in the ground truth response.

Unfortunately, such training data are expensive to produce in sufficient quantities to support high-quality performance. To resolve this problem, human annotators then indicate which of several model responses they prefer. These (much cheaper) data are used to train a *reward model*. This is a second transformer network that ingests the prompt and model response and returns a scalar indicating how good the response is. Finally, the fine-tuned chatbot model is further trained to produce high rewards using the reward model as supervision. Here, standard gradient descent cannot be used as it's not possible to compute derivatives through the sampling procedure in the chatbot output. Hence, the model is trained with proximal policy optimization (a policy gradient method where the derivatives are tractable) to generate higher rewards.

Other areas of RL: Reinforcement learning is an enormous area, which easily justifies its own book, and this literature review is extremely superficial. Other notable areas of RL that we have not discussed include *model-based RL*, in which the state transition probabilities and reward functions are modeled (see Moerland et al., 2023). This allows forward planning and has the advantage that the same model can be reused for different reward structures. *Hybrid methods* such as AlphaGo (Silver et al., 2016) and MuZero (Schrittwieser et al., 2020) have separate models for the dynamics of the states, the policy, and the value of future positions.

This chapter has only discussed simple methods for exploration, like the epsilon-greedy approach, noisy Q-learning, and adding an entropy term to penalize overconfident policies. *Intrinsic motivation* refers to methods that add rewards for exploration and thus imbue the agent with "curiosity" (see Barto, 2013; Aubret et al., 2019). *Hierarchical reinforcement learning* (see Pateria et al., 2021) refers to methods that break down the final objective into sub-tasks. *Multi-agent reinforcement learning* (see Zhang et al., 2021a) considers the case where multiple agents coexist in a shared environment. This may be in either a competitive or cooperative context.

Problems

Problem 19.1 Figure 19.18 shows a single trajectory through an example Markov reward process. Calculate the return for each step in the trajectory given that the discount factor γ is 0.9.

Problem 19.2* Prove the policy improvement theorem. Consider changing from policy π to policy π', where for state s_t the new policy π' chooses the action that maximizes the expected return:

$$\pi'[a_t|s_t] \leftarrow \underset{a_t}{\text{argmax}}\Big[r[s_t,a_t] + \gamma \cdot \sum_{s_{t+1}} Pr(s_{t+1}|s_t,a_t)v[s_{t+1}|\pi]\Big]. \tag{19.43}$$

and for all other states, the policies are the same. Show that the value $v[s_t|\pi]$ for the original policy must be less than or equal to $v[s_t|\pi']$ for the new policy (notation indicates using π' for state s_t and π thereafter):

$$\begin{aligned} v[s_t|\pi] &\leq q\Big[s_t, \pi'[a_t|s_t]\Big|\pi\Big] \\ &= \mathbb{E}_{\pi'}\Big[r_{t+1} + \gamma \cdot v[s_{t+1}|\pi]\Big]. \end{aligned} \tag{19.44}$$

Hint: Start by writing the term $v[s_{t+1}|\pi]$ in terms of the new policy.

Problem 19.3 Show that when the state values and policy are initialized as in figure 19.10a, they become those in figure 19.10b after two iterations of (i) policy evaluation (in which all states are updated based on their current values and then replace the previous ones) and (ii) policy improvement. The state transition allots half the probability to the direction the policy indicates and divides the remaining probability equally between the other valid actions. The reward function returns -2 irrespective of the action when the penguin leaves a hole. The reward function returns +3 regardless of the action when the penguin leaves the fish tile and the episode ends, so the fish tile has a value of +3. Assume a discount factor of $\gamma = 0.9$.

Problem 19.4 The *Boltzmann policy* strikes a balance between exploration and exploitation by basing the action probabilities $\pi[a|s]$ on the current state-action reward function $q[s,a]$:

$$\pi[a|s] = \frac{\exp\big[q[s,a]/\tau\big]}{\sum_{a'}\exp\big[q[s,a']/\tau\big]}. \tag{19.45}$$

Explain how the temperature parameter τ can be varied to prioritize exploration or exploitation.

Problem 19.5* When the learning rate α is one, the Q-Learning update is given by:

$$\mathrm{f}\big[q[s,a]\big] = r[s,a] + \gamma \cdot \max_{a}\big[q[s',a]\big]. \tag{19.46}$$

where s is a state and s' is the subsequent state. Show that this is a contraction mapping (equation 16.30) so that:

$$\Big|\Big|\mathrm{f}\big[q_1[s,a]\big] - \mathrm{f}\big[q_2[s,a]\big]\Big|\Big|_\infty < \Big|\Big|q_1[s,a] - q_2[s,a]\Big|\Big|_\infty \qquad \forall\, q_1, q_2. \tag{19.47}$$

Appendix B.3.2 Vector norms

where $||\bullet||_\infty$ represents the ℓ_∞ norm. It follows that a fixed point will exist by Banach's theorem and that the updates will eventually converge.

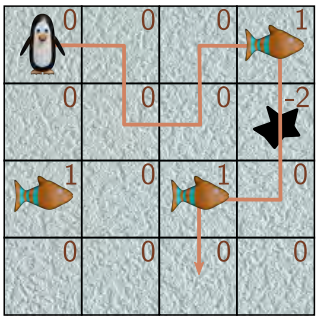

Figure 19.18 One trajectory through an MRP. The penguin receives a reward of $+1$ when it reaches the first fish tile, -2 when it falls in the hole, and $+1$ for reaching the second fish tile. The discount factor γ is 0.9.

Problem 19.6 Show that:

$$\mathbb{E}_{\boldsymbol{\tau}}\left[\frac{\partial}{\partial\boldsymbol{\theta}}\log\bigl[Pr(\boldsymbol{\tau}|\boldsymbol{\theta})\bigr]b\right]=0, \tag{19.48}$$

where b does not depend on τ, so adding a baseline update doesn't change the expected policy gradient update.

Problem 19.7* Suppose that we want to estimate a quantity $\mathbb{E}[a]$ from samples $a_1, a_2 \ldots a_I$. Consider that we also have paired samples $b_1, b_2 \ldots b_I$ that are samples that co-vary with a where $\mathbb{E}[b] = \mu_b$. We define a new variable:

$$a' = a - c(b - \mu_b). \tag{19.49}$$

Show that $\mathrm{Var}[a'] \leq \mathrm{Var}[a]$ when the constant c is chosen judiciously. Find an expression for the optimal value of c.

Problem 19.8 The estimate of the gradient in equation 19.34 can be written as:

$$\mathbb{E}_{\boldsymbol{\tau}}\Bigl[\mathrm{g}[\boldsymbol{\theta}](r[\boldsymbol{\tau}_t]-b)\Bigr], \tag{19.50}$$

where

$$\mathrm{g}[\boldsymbol{\theta},\tau]=\sum_{t=1}^{T}\frac{\partial\log\bigl[Pr(a_t|\mathbf{s}_t,\boldsymbol{\theta})\bigr]}{\partial\boldsymbol{\theta}}, \tag{19.51}$$

and

$$r[\boldsymbol{\tau}]=\sum_{k=t}^{T}r_k. \tag{19.52}$$

Show that the value of b that minimizes the variance of the gradient estimate is given by:

$$b=\frac{\mathbb{E}\bigl[\mathrm{g}[\boldsymbol{\theta},\boldsymbol{\tau}]^2 r[\boldsymbol{\tau}]\bigr]}{\mathbb{E}\bigl[\mathrm{g}[\boldsymbol{\theta},\boldsymbol{\tau}]^2\bigr]}. \tag{19.53}$$

You will need to use the result from equation 19.35.