Part IV

Reinforcement Learning

13 Introduction to Reinforcement Learning

This part of the course concerns *Reinforcement Learning (RL)*, the conceptual underpinning of several modern technologies such as self-driving technologies in new cars. It is the third major category of machine learning, in addition to the two previously seen categories of supervised and unsupervised learning. In class we saw a video of robots (made by Boston Dynamics) doing parkour, dancing, and overall doing a pretty good job of imitating the peak human physique. That is also achieved via RL.

The basic idea of RL involves the concept of an *agent* learning to make a *sequence of actions* in a *dynamic* environment. At each discrete time step, the agent is able to take one of a menu of actions. Each choice of action leads to changes in the state of the world (*i.e.*, the agent and its surroundings). The agent has an internal representation of the current and potential states of the world (*e.g.*, using vision or other sensing modules). Under this setting, the agent takes a sequence of actions towards a certain goal.

The world contains uncertainty due to a variety of factors. For instance, there may be other agents in the environment that also take actions to their own benefit, or the sensing modules may be imperfect. Thus taking the same action from the same state of the world may lead to different evolution of state in the future — that is, RL is *non-deterministic*.

In this chapter, we introduce the basic elements of RL using real-world examples, and what it means for the agent to act optimally. Chapter 14 focuses on the setting where the underlying environment (*e.g.*, the number of states, the current state, the probability distribution) is completely known to the agent. [1] In Chapter 15, we will present the case where the environment is not fully available to the agent, and the agent learns about the environment while also learning to act in it. [2]

[1] Think of playing a game where you know the complete set of rules.

[2] Think of playing a Role-playing Game (RPG) where you need to unlock parts of the map by advancing the story.

13.1 Basic Elements of Reinforcement Learning

Now we formalize several of the basic elements of reinforcement learning that were sketched above.

13.1.1 States and Actions

There is a finite set S of states, and the entire system agent + environment exists in one of these states at any time. At each state $s \in S$, the agent makes an action $a \in A_s$, where A_s is the set of allowed actions at state s. We denote $A = \bigcup\limits_{s \in S} A_s$ to be the set of all possible actions in the whole RL environment.

Example 13.1.1. *Consider a game of chess. Each state s can be represented as a pair (C, p) where C denotes the current configuration of pieces and p denotes the player to play next. For example, "white king at e1, black king at e8, and it is white turn to move" would be a possible state s of the game. An action a is a valid movement of a piece, given a state of the game. For example, "white king to e2" (i.e., Ke2) would be a possible action of the agent playing white in state s.*

Example 13.1.2. *Self-driving cars, like those built by Tesla, are becoming increasingly popular. Let's imagine how we could construct a state diagram for the task of driving autonomously. Each state can be represented by the current configuration of a number of factors (e.g., the car speed, distance from lane boundaries, distance to nearest vehicle, etc.) Possible actions include increasing/decreasing speed, changing gear, changing direction, changing lane, etc.*

13.1.2 Modeling Uncertainty via Transition Probabilities

As mentioned, the agent has many sources of uncertainty in its knowledge about the environment, and we can use concepts from probability to model uncertainty.

Suppose $S = \{s_1, s_2, \ldots, s_n\}$ contains n states. When the agent takes action a while in state s, it will *transition* into another (potentially the same) state s'. The catch is: the agent does not know exactly which state it will end up in. Instead, there is a probability p_i of ending up in state s_i for each $s_i \in S$. Here $\sum_i p_i = 1$, meaning each (state, action) pair is associated with a *probability distribution* over the next state that the agent will enter. Formally, we define it as follows:

Definition 13.1.3 (Transition Probabilities)**.** *Given a state $s \in S$ and an action $a \in A_s$, there is an associated* ***transition probability*** *$p(* \mid s, a)$ distributed over S such that state $s' \in S$ happens with probability $p(s' \mid s, a)$*

when action a is taken at state s and $\sum_{s' \in S} p(s' \mid s, a) = 1$*. If* $p(s' \mid s, a) > 0$*, we say that the state s' is reachable from s when action a is taken.*

In general, not all states are reachable, given a state s and an action a. That is, some transition probability $p(s' \mid s, a)$ is zero. For these states, it is conventional to leave out the corresponding transitions when representing the RL environment as a state diagram as in Figure 13.1 or Figure 13.2.

Example 13.1.4. *Consider the state diagram shown in Figure 13.1. This is a special case where there is only one action a in the set A. In other words, the agent is not making any choices; instead, it is just following probabilistic transition over time steps. To calculate the probability of reaching state s_3 from s_0, we note there are two different paths. The first path is $s_0 - s_1 - s_3$ and the second path is $s_0 - s_2 - s_3$. We thus calculate the probabilities of each of these paths and note that the overall probability of reaching s_3 will be the sum of both:* $0.2 \cdot 0.7 + 0.8 \cdot 0.4 = 0.46$.

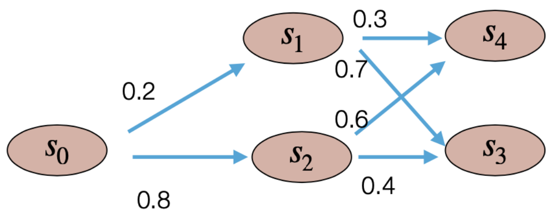


Figure 13.1: An example diagram where $|A| = 1$. The agent simply follows probabilistic transitions.

Now we consider an example where there is more than one action to make. In this case, each action induces a different probability distribution on the set of states, so we need to draw a diagram for each option.

Example 13.1.5. *We can model a baby learning to walk through RL. As shown in Figure 13.2, we can define the state s_0 = **standing but feeling unsteady**, s_1 = **standing and feeling secure**, and s_2 = **on ground**. The baby has two actions to take: a = **not grab onto nearest support** and a' = **grab onto nearest support**. The state diagram on the top represents the transition probabilities when the baby takes the action a. See that the baby has a high chance of entering state s_2 — falling to the ground. On the other hand, the state diagram on the bottom represents the transition probabilities when the baby takes the action a'. The baby now has a high chance of entering state s_1 — standing securely on the ground. The two actions have **different** probability distributions associated with the relevant transitions.*

Example 13.1.6. *Mechanical ventilators are used to stabilize breathing for patients. Suppose we wished to construct a state diagram. We could define*

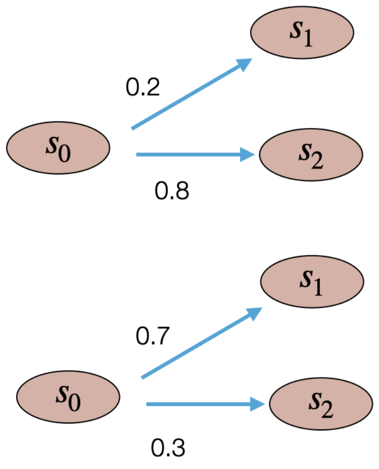


Figure 13.2: An example diagram showing how an action determines the probability of outgoing transitions.

states that consider the pressure and CO_2 *level in the patient's level for the past k seconds. Actions might include adjusting the flow rate of oxygen via valve settings as needed. Possible transitions might include the typical mechanical response of the lungs, or unexpected spasms. Finally, we can define the goal to be maintaining steady pressure in the patient without "overshooting" and causing damage.*

13.1.3 *Agent's Motivation/Goals*

In general, an agent is a participant in RL models driven by the need to maximize "rewards." In a probabilistic setting, the agent wishes to maximize their *expected* rewards. In a natural setting, the "rewards" could be innate satisfaction, such as getting to eat food, being entertained, etc. But in the usual artificial settings such as robots and self-driving cars, rewards are sprinkled by the system designer into the framework. Some examples appear later.[3]

[3] While reward/punishment as a way to shape human or animal behavior is a very old idea, mathematical modeling of agents as reward-maximisers appears in several disciplines that flowered around the middle of the 20th century (*e.g.*, *behaviorism* in psychology, profit-maximisation in economics, and of course RL).

At each step, the agent takes an action, and is given a reward (which could be negative, *i.e.*, is a punishment) based on the action, current state, and next state.

Definition 13.1.7 (Reward)**.** *For each valid 3-tuple* (a, s, s') *where* $s' \in S$ *is a state reachable from state* $s \in S$ *by taking action* $a \in A_s$*, we define a corresponding* ***reward*** $r(a \mid s_1, s_2) \in \mathbb{R}$.

Example 13.1.8 (Example 13.1.5 revisited)**.** *When the baby stands and feels secure after grabbing onto something, the parents applaud the baby, and the baby receives a positive reward:* $r(a' \mid s_0, s_1) = 5$*. When the baby feels secure without grabbing onto the nearest support, the parents feel even prouder and the baby gets a more positive reward:* $r(a \mid s_0, s_1) = 10$*. When the baby falls to the ground, the baby feels pain and receives negative reward:*

$r(a \mid s_0, s_2) = r(a' \mid s_0, s_2) = -5.$

Typically, the designer of a RL model gets to define the rewards throughout the framework based on the designer's judgment. For instance, in Example 13.1.2, we might design an RL model such that if the car drifts into an adjacent lane, we assign a negative reward. If another vehicle is in the lane, we might assign an even larger negative reward. This will induce a RL model to "learn" the proper way to driving — staying in lane.

13.1.4 *Comparison with NFA*

Recall the Non-deterministic Finite Automata (NFA) you learned in COS 126. In an NFA, there is a finite number of states, and for each state, we know the set of next possible states, based on the next input character.

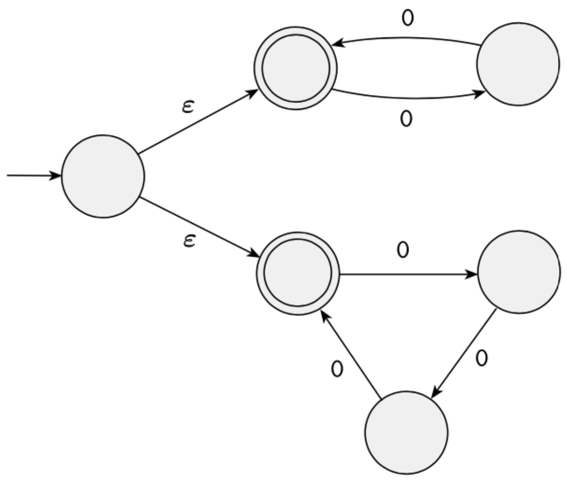


Figure 13.3: A sample Non-deterministic Finite Automata. Source: *Introduction to the Theory of Computation* by Michael Sipser.

We can consider the following analogy between RL and NFA — there is someone behind an NFA, who can observe its current state and type in the next input character. This person will be called an *agent*, and the choice of input character that is typed in will be called an *action*. Each action can lead to a finite set of next possible states, but because of some uncertainty in the world, the agent cannot specify which particular state will be the next one. This is similar to an NFA in the sense that the actions are non-deterministic. Also, just like in an NFA, the change in the current state is also referred to as a *transition*. One major difference between RL and NFA is that while an NFA only cares about the final state of the automata (*i.e.*, whether it is an accept state or a reject state), in RL, the agent is given a *reward* after each transition. The goal of the agent will be to take a sequence of actions so as to maximize the sum of the reward throughout the sequence of actions.

13.2 *Useful Resource: MuJoCo-based RL Environments*

Real-life robots with precise and reliable hardware can get very expensive to buy, let alone train. An easier playground for students (especially those trying to work with a single GPU on CoLab) is doing RL in a virtual environment.

MuJoCo is a famous physics engine that allows creating virtual objects with somewhat realistic "joints" that can be commanded to move similar to real-life robots. OpenAI and DeepMind have open-source environments that allow experimentation in the MuJoCo environment. The official website gives a pretty good overview of the software: [4]

[4] Source: https://mujoco.org.

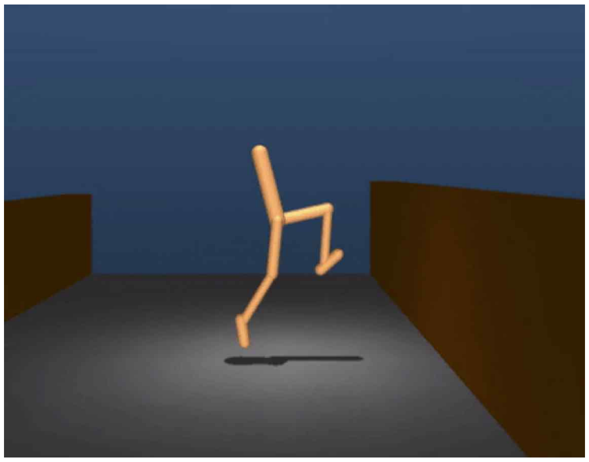


Figure 13.4: An example of a MuJoCo Walker.

MuJoCo is a physics engine that aims to facilitate research and development in robotics, biomechanics, graphics and animation, and other areas where fast and accurate simulation is needed. MuJoCo offers a unique combination of speed, accuracy and modeling power, yet it is not merely a better simulator. Instead it is the first full-featured simulator designed from the ground up for the purpose of model-based optimization, and in particular optimization through contacts.

One aspect of MuJoCo simulation involves a representation of a humanoid figure (*i.e.*, the agent) learning how to navigate an obstacle course (*i.e.*, the environment). Training videos are readily available online and show how the agent learns over time (sometimes, to comedic effect).

Example 13.2.1. *Let's analyze the example of an agent navigating an obstacle course through a RL framework. The states can be considered to be the set of coordinates, velocity, and acceleration for each limb, the velocity* *and acceleration for the motors in each joint, and the environment itself straight ahead. The actions can include the agent increasing or decreasing motor speed in their joints. Finally, the final goal is to stay upright, run forward at a reasonable pace, and avoid obstacles.*

13.3 *Illustrative Example: Optimum Cake Eating*

Let's consider an extended example which ties together the elements of RL discussed previously. Suppose you buy a small cake with three slices. The reward of eating one slice at one sitting is 1, but eating two or three slices at one sitting is 1.5 and 1.8 respectively. [5]

[5] This sense of diminishing rewards is known as the satiation effect.

Problem 13.3.1. *Suppose you plan to eat the cake over a period of three days. What eating schedule will maximize the internal reward?*

Now let's introduce your roommate, who is oblivious to basic understandings of ownership and adheres to the "finders keepers" faith. We define the probability $\Pr[\text{sneakily eats a slice overnight}] = \frac{1}{2}$. To account for this uncertainty, we can create a *look-ahead tree* for different initial actions. We first consider the action where you decide to eat two out of the three slices on the first night. Successive states and associated probabilities are shown in the Figure 13.5.

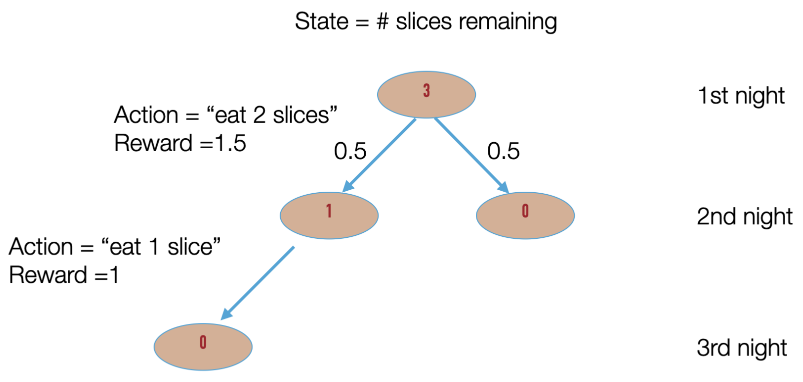


Figure 13.5: The diagram (look-ahead tree) of the cake problem where you decide to eat two slices on the first night. Each state represents the number of slices remaining, and each action represents the number of slices eaten on one night.

Even though you only eat only two out of the three slices during the first night, there is a $\frac{1}{2}$ chance that your roommate eats the remaining slice overnight. Therefore, the action of "eating 2 slices" can lead to two possible states — "1 slice left" or "0 slice left" — each with probability $\frac{1}{2}$.

Example 13.3.2. *We can calculate the expected reward associated with eating two slices on the first night by analyzing the Figure 13.5. You first gain reward of* 1.5 *by eating the two slices on the first night. Then with probability* $\frac{1}{2}$ *(where the roommate does not eat the remaining slice overnight),* *you get to eat the last slice on the second night and gain additional reward of* 1*. With probability of* $\frac{1}{2}$ *(where the roommate eats the remaining slice), you cannot gain anymore reward. That is, the expected reward is* $1.5 + 0.5 \cdot 1 + 0.5 \cdot 0 = 2$.

Problem 13.3.3. *Consider the result of Example 13.3.2. Would you prefer to take two slices on the first night or three slices?*

We next consider the action where you decide to eat one out of the three slices on the first night. Successive states and associated probabilities are shown in the Figure 13.6.

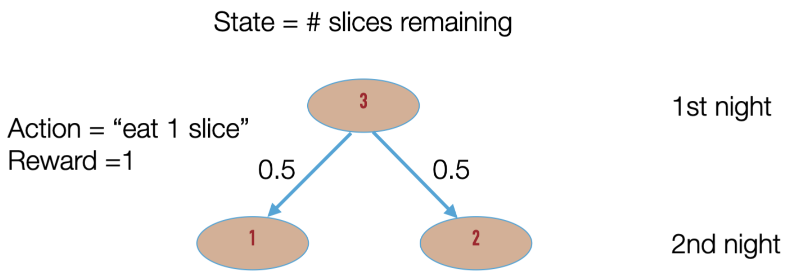


Figure 13.6: The diagram (look-ahead tree) of the cake problem where you decide to eat one slice on the first night.

The difficulty in this example in contrast to the Figure 13.5 is that if the roommate does not eat a slice after the first night, you have two slices at your disposal on the second night. You have two actions you can take in this "2 slices left" state — "eat 1 slice" (and hope the third slice is still there on the third night) or "eat 2 slices" — and it is not immediately obvious which one is more optimal. It turns out that the expected reward you can get from the remaining 2 slices is 1.5 for both options.

Problem 13.3.4. *Verify the previous claim that both options on the second night have the same expected reward.*

Example 13.3.5. *Given the previous analysis and the look-ahead tree in the Figure 13.6, we note that the total expected reward is* $1 + 0.5 \cdot 1 + 0.5 \cdot 1.5 = 2.25$*. You first receive a reward of* 1 *by eating* 1 *slice on the first night. Then with probability* $\frac{1}{2}$*, the roommate eats one slice over night, and you gain reward of* 1 *by eating the last slice on the second night. With the remaining probability* $\frac{1}{2}$*, the roommate does not eat a slice, and you are expected to gain reward of* 1.5 *from the remaining* 2 *slices, regardless of the action you choose to take on the second night.*

Problem 13.3.6. *Consider the result of Example 13.3.5. Would you prefer to take two slices on the first night or one slice?*

14
Markov Decision Process

In this chapter, we formally introduce the *Markov Decision Process (MDP)*, a way to formulate an RL environment. We then present ways to find the optimal strategy of an agent, provided that the agent knows the full details of the MDP — that is, knows everything about the environment.

14.1 Markov Decision Process (MDP)

Let's review the key ingredients of RL. We have the *agent*, who senses the environment and captures it as the current *state*. There is a finite number of actions available at any given state, and taking an action a in state s will cause a transition to s' with probability $p(s' \mid s, a)$. Each transition is accompanied by a reward $r(a \mid s, s_i) \in \mathbb{R}$. Finally, the goal of the agent is to maximize the expected reward via a sequence of actions.

A *Markov Decision Process (MDP)* is a formalization of these concepts. It is a *directed graph* which consists of four key features:

- A set S which contains all possible states
- A set A which contains all possible actions
- For each valid tuple of action a and states s_1, s_2, there is an assigned probability $p(s_2 \mid s_1, a)$ probability of transition to s_2 if action a is taken in s_1
- For each valid tuple of action a and states s_1, s_2, there is an assigned reward $r(a \mid s_1, s_2)$, which is obtained if action a is taken to transition from s_1 to s_2

If a designed MDP has M actions and N states, we can specify the MDP by a table of transition probabilities (with MN^2 numbers) and a table for rewards (with MN^2 numbers).

14.1.1 *Revisiting the Cake Eating Example*

Let's return to the case study on eating cake from Subsection 13.3, and formally express it through a MDP. The set of states is given as $S = \{0, 1, 2, 3\}$, where each state represents the number of slices left. The set of actions is given as $A = \{1, 2, 3\}$, where each action represents the number of slices you choose to eat on a given night. Notice that reward only depends on how many slices you take, not how many slices are left after your roommate goes through the fridge. That is, we can define the reward $r(a \mid s, *)$ for each $a \in A$ to be the same for every $s \in S$ where a is feasible. [1]

[1] We still need to include the previous state s because not all actions are feasible at each state. For example, you can't eat 2 slices when there is only 1 slice left.

Example 14.1.1. *Let's revisit Example 13.3.2 as a motivating example. If we let $a = 2$, $s_1 = 3$, and $s_2 = 0$, then the probability of the specified transition is $p(s_2 \mid s_1, a) = 0.5$. The associated reward is $r(a \mid s_1, s_2) = 1.5$ as discussed earlier.*

We are now ready to generalize to the full MDP, which is shown in Figure 14.1. Note that every transition is labeled with its probability, associated action, and associated reward.

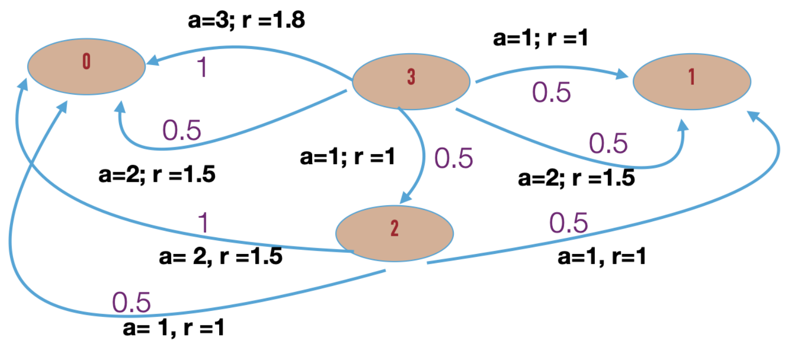


Figure 14.1: The full diagram of the cake problem when described as a MDP.

14.1.2 *Discounting the Future*

The MDP describing cake eating in the previous subsection was acyclic. [2] However, in general, MDPs can have directed cycles, and the agent's actions can allow it to continuously collect rewards along that cycle. For instance, continuing our cake theme, we may have a scenario in which you receive a fresh cake every 3 days. But now we run into a problem: how can we calculate the expected reward when there is an unbounded number of steps?

[2] This is also known as an *Episodic MDP*.

The solution lies in the concept of *future discounting*. The basic idea is to reduce, or *discount*, the amount of reward we get from future steps. In an MDP, we represent this through a discount factor $0 < \gamma \leq 1$ and an associated infinite sum. [3]

[3] This is related to notions of discounting commonly considered in economics.

Definition 14.1.2 (Future Discounting). *If a reward r_t is received at time $t = 0, 1, 2, \ldots$, then the perceived value of these rewards r_d, or the* ***discounted reward****, at $t = 0$ is:*

$$r_d = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots$$

Example 14.1.3. *Consider the cake eating problem again and let r_t denote the reward we get on night t. If the reward is discounted by a factor of γ every night, the total expected discounted reward $E[total]$ can be rewritten as*

$$\mathbb{E}[total] = \mathbb{E}[r_1] + \gamma \cdot \mathbb{E}[r_2] + \gamma^2 \cdot \mathbb{E}[r_3]$$

Consider taking the action $a = 2$ on the first night. If $\gamma = 0.9$, then the expected discounted reward is

$$1.5 + 0.9 \cdot (0.5 \cdot 1 + 0.5 \cdot 0) = 1.95$$

This is the same as in Example 13.3.2 except the reward taken from the second night is discounted by a factor of 0.9. Now consider taking the action $a = 1$ on the first night and on the second night. If $\gamma = 0.9$, the expected discounted reward is

$$1 + 0.9 \cdot (0.5 \cdot 1 + 0.5 \cdot 1) + 0.9^2 \cdot (0.5^2 \cdot 1) = 2.1025$$

Here, we first take the reward of 1 on the first night without any discount factor. Then, we calculate the expected reward from the second night — 1 whether or not the roommate eats a slice — and discount it by a factor of 0.9. Finally, we calculate the expected reward from the third night — 1 only if the roommate did not eat any slice on the first two nights — and discount it by a factor of 0.9^2.

Note that in Definition 14.1.2, if each $r_t \in [-R, R]$ and if $\gamma < 1$, then the discounted reward of the infinite sequence has the following upper bound:

$$|r_d| \leq R(1 + \gamma + \gamma^2 + \cdots) = \frac{R}{1-\gamma} \tag{14.1}$$

(14.1) is derived by considering the formula for the sum of an infinite geometric series, which we can invoke if $\gamma < 1$. In general, γ is up to the system designer. A lower γ would imply that the agent places little importance on future rewards, whereas $\gamma = 1$ would imply that there is effectively no discounting.

14.2 *Policy and Markov Reward Process*

Now that we have discussed what an action is and what it does in an MDP, we want to specify what action an agent has to take in each state. This is known as a *policy*.

Example 14.2.1. *Consider again the cake eating MDP example without a discount factor. We already established through Example 13.3.2 and Example 13.3.5 that to maximize the expected reward, you need to eat one slice per day until all slices are gone. That is, in any state j where $j = 1, 2, 3$, you need to take action 1.*

In general, if S is the set of states, and A is the set of actions, then a *policy* (not necessarily the optimum) π can be defined as a function $\pi : S \to A$

Definition 14.2.2 (Policy)**.** *If S is the set of states, and A is the set of actions, any function $\pi : S \to A$ is called a **policy** that describes which action to take at each state. In particular, each state s should only be mapped to a valid action $a \in A_s$ at that state.*

Recall that if there are M actions and N states, there are at most MN^2 transitions in the graph of the MDP. Because a policy specifies one action per state, there are at most N^2 transitions that remain when we choose a specific policy. Therefore, it can be understood that a policy trims out the MDP.

14.2.1 Markov Reward Process (MRP)

When we have an MDP and a fixed policy, we have what is called a *Markov Reward Process (MRP)*. There are no more decisions to make; instead, all we need to do is take the action specified by the policy; probabilistically follow a transition into a new state; and collect the associated reward.

Example 14.2.3. *Let's revisit Figure 14.1. If we fix the policy to be $\pi(s) = 1$ for any $s \in S$, we can focus our attention to the action $a = 1$. Then there are three trajectories that will lead from state 3 to state 0, based on what the roommate does overnight. The first trajectory is $3 \to 1 \to 0$ with probability 0.5×1 and reward $1 + 1$. The second trajectory is $3 \to 2 \to 0$ with probability 0.5×0.5 and reward $1 + 1$. The last trajectory is $3 \to 2 \to 1 \to 0$ with probability $0.5 \times 0.5 \times 1$ and reward $1 + 1 + 1$.*

In general, when we fix a policy π and an initial state s, we can redraw the transition diagram of an MDP into a tree diagram for the MRP, where each node corresponds to a state, and each edge corresponds to a probabilistic transition. The top node represents the initial state, and each subsequent row of the tree represents the set of possible states after taking an action from their parent node.

Example 14.2.4. *We revisit Example 14.2.3. We now transform Figure 14.1 into a tree diagram for the MRP as shown in Figure 14.2. The top node is the initial state 3. The second row of the tree is all states that can be achieved by taking the action 1 at state 3, and so on.*

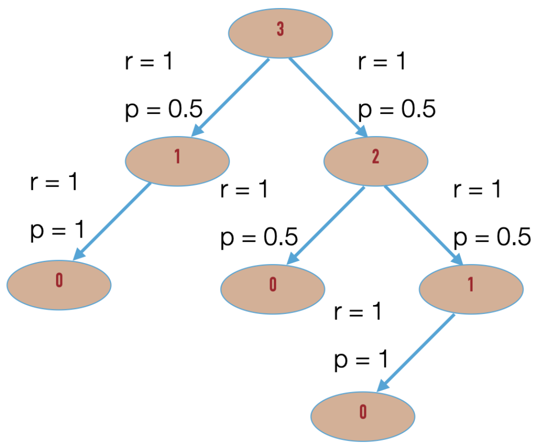


Figure 14.2: A tree representing the MRP in Example 14.2.3.

Note that in an MRP tree, the same state can appear multiple times, but each copy of the same state is identical — that is, the subtree rooted at each copy must be identical. In Figure 14.2, the state 1 appears three time in the tree. Every time it appears, it can only lead to state 0 with probability 1. This is simply the result of fixing a policy π — once we know the state we are in, we only have one choice for the action to take.

The policy also induces a *value function* on this tree. The value function assigns a value to each node of the tree, and each value intuitively measures how much reward the agent should expect to collect once the agent knows they have arrived at that node. By the observation from the previous paragraph, this expected reward should be the same for two nodes, if they are the copy of the same state. Therefore, we can equivalently define the value function for each state s instead. Formally, we define the value function as the following.

Definition 14.2.5 (Value Function). *$v_\pi(s)$, the **value of state** s **under the policy** π, is the expected discounted reward of a random trajectory starting from s. We can define this value by using the following recursive formula:*

$$v_\pi(s) = \sum_{s'} p(s' \mid s, \pi(s)) \cdot (r(\pi(s) \mid s, s') + \gamma v_\pi(s')) \tag{14.2}$$

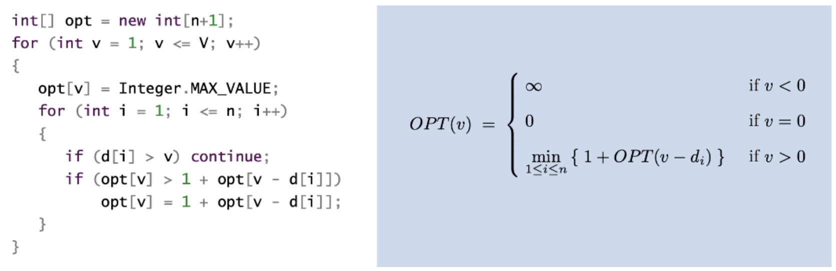
*Computing the value function as in (14.2) is also known as the **Bellman equation**.*

Let us unpack the intuition behind (14.2). Once we take action $\pi(s)$ at state s, it will bring us to state s' with probability $p(s' \mid s, a)$, immediately giving us a reward $r(a \mid s, s')$. Then, the expected reward from that point on is already captured by the value $v_\pi(s')$. We just need to apply the discount factor γ because we already took one time step to reach s' from s.

On the other hand, if we pick any random trajectory starting from s, its next node will be some state s' that is reachable from s. Therefore, the contribution of this particular trajectory to $v_\pi(s)$ is accounted for when we sum over that particular s'.

14.2.2 Connection with Dynamic Programming

In COS 226, you may have seen an implementation of a bottom-up dynamic programming.



```
int[] opt = new int[n+1];
for (int v = 1; v <= V; v++)
{
   opt[v] = Integer.MAX_VALUE;
   for (int i = 1; i <= n; i++)
   {
      if (d[i] > v) continue;
      if (opt[v] > 1 + opt[v - d[i]])
          opt[v] = 1 + opt[v - d[i]];
   }
}
```

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min\limits_{1 \le i \le n} \{\, 1 + OPT(v - d_i) \,\} & \text{if } v > 0 \end{cases}$$

Figure 14.3: A Dynamic Programming implementation of a coin changing problem that uses the bottom-up approach.

In such implementations, the algorithm divides the problem into subproblems arranged as directed acyclic graphs and compute “bottom-up.” The MDP from the cake eating problem is acylic and our method using a look-ahead tree is similar to the dynamic programming algorithms. Therefore, it seems like we can apply a similar algorithm to the cake eating problem.

Example 14.2.6. *Consider Example 14.2.3 again, but now with a discount factor of* 0.9*. We will find the value $v_\pi(s)$ of each state s by going bottom-up from the tree in Figure 14.2. We start by noticing that $v_\pi(0) = 0$ as can be seen from the bottom row. Then from the third node of the third row, we can calculate*

$$v_\pi(1) = 1 \cdot (1 + 0.9 \cdot 0) = 1$$

From the second node of the second row, we can calculate

$$v_\pi(2) = 0.5 \cdot (1 + 0.9 \cdot 0) + 0.5 \cdot (1 + 0.9 \cdot 1) = 1.45$$

Finally, from the top node, we can calculate

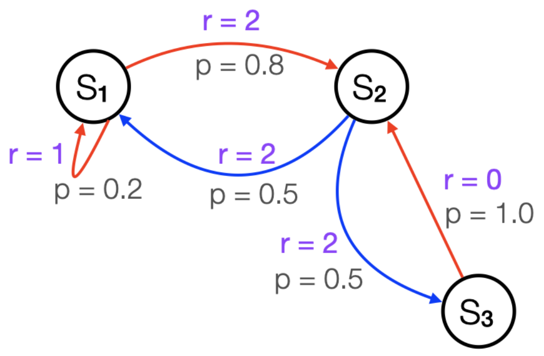
$$v_\pi(3) = 0.5 \cdot (1 + 0.9 \cdot 1) + 0.5 \cdot (1 + 0.9 \cdot 1.45) = 2.1025$$

But in general, the dynamic programming approach does not completely apply to MDP. The biggest assumption for dynamic programming algorithms is that the graph is *acyclic*, but MDPs are generally allowed to have directed cycles if we can return to the same state after a sequence of actions. Therefore, computing the expected reward for even a single policy π involves solving a system of linear equations.

Example 14.2.7. *Assume that we have three states s_1, s_2, s_3 and transitions as in Figure 14.2.2 with a discount factor of $\gamma = 0.7$. Then the value at each state is given as*

$$\begin{aligned}
v_\pi(s_1) &= 0.2 \times (1 + 0.7 v_\pi(s_1)) + 0.8 \times (2 + 0.7 v_\pi(s_2)) \\
v_\pi(s_2) &= 0.5 \times (2 + 0.7 v_\pi(s_1)) + 0.5 \times (2 + 0.7 v_\pi(s_3)) \\
v_\pi(s_3) &= 1 \times (0 + 0.7 v_\pi(s_2))
\end{aligned}$$

Unlike in Example 14.2.6, we cannot compute any of these values one by one because the values are interdependent in a cyclic manner. Instead, we need to solve the linear equation as a whole, which gives us the solution: $v_\pi(s_1) \approx 5.47, v_\pi(s_2) \approx 5.18, v_\pi(s_3) \approx 3.63$.



14.3 Optimal Policy

Out of all choices for a policy, we are interested in the *optimal policy*, the one that maximizes the expected (discounted) reward. Surprisingly, it is known that there always exists a policy π^* that obtains the maximum expected reward from all initial states *simultaneously*; that is $\pi^* = \arg\max_\pi v_\pi(s)$ for every state s. [4]. The value function of the optimal policy is called the *optimal value function* and is often denoted as $v^*(s)$. Then we can express the optimal value function using (14.2) as:

$$v^*(s) = \max_\pi \sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v_\pi(s'))$$

[4] If there are multiple such policies, we denote any one of them by π^*.

This is just restating the fact that the optimal value of state s is the maximum of all possible values $v_\pi(s)$ of s under a policy π — *i. e.*, the Bellman equation evaluated with the values $v_\pi(s')$ of each children node s' under that specific policy π.

But we can even go further than this result. It is known that the optimal value also satisfies the following:

$$v^*(s) = \max_\pi \sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v^*(s')) \qquad (14.3)$$

Notice that $v_\pi(s')$ in the summation has now been replaced with $v^*(s')$. This property, known as the *Bellman Optimality condition*, states that the optimal value is even the maximum when the Bellman equation is evaluated with the values $v^*(s')$, regardless of the choice of the policy π.

Notice that the right-hand side of (14.3) only depends on the choice of the action a of the given state s, not any other states. Therefore, we can rewrite (14.3) as:

$$v^*(s) = \max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \tag{14.4}$$

which also suggests that the optimal action at state s can be expressed as:

$$\pi^*(s) = \arg\max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \tag{14.5}$$

But the problem is: it is unclear how to turn this into an efficient algorithm. Computing the value $v^*(s)$ depends on the value $v^*(s')$, which can also depend on $v^*(s)$, which becomes recursive.

In this section, we present an iterative algorithm called the *value iteration* method which will be used to compute the optimal policy. Before we describe the algorithm, we unpack the underlying ideas.

14.3.1 Developing Intuition about Optimality: Gridworld

To develop intuition about how to find an optimum policy, let's consider a classic example called Gridworld. [5]

[5] Source: Sutton and Barton 2020, `https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf`

Example 14.3.1 (Gridworld)**.** *Consider a 5×5 grid. The set of states is given as the cells of this grid. At each state except for at $A = (1,2)$ and $B = (1,4)$, there are four available actions: move left/right/up/down, each with reward 0, except in the following setting: if the action will make you move off the grid. then the reward is -1, and you are made to stay at the same state instead.*

At A, there is only one action: move to $A' = (5,2)$ with reward 10 and similarly at B, there is one action: move to $B' = (3,4)$ with reward 5. [6] The discount factor is given as 0.9.

[6] The outgoing transition from A and B can be thought of as "wormholes."

How can we compute the reward for a policy in the example above? When beginners try to calculate the exact value using the above definitions, they quickly get bogged down in keeping track of too many variables, equations, and recurrences.

Instead, let's try to think intuitively about what an optimal policy **should** be trying to do. Since the wormholes are the only source of rewards, an optimal policy should be trying to utilize the wormholes as much as possible. Using this kind of intuition, we can design a

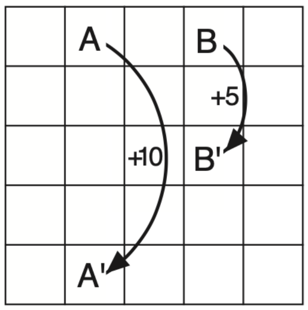


Figure 14.4: Visual representation of the Gridworld.

policy that looks at least near-optimal, and use its value as a **lower bound** for the optimal policy.

First, let $v^*(s)$ denote the value $v_{\pi^*}(s)$ of state s for an optimal policy π^*. Since there is only one action to choose from at state A, we know that

$$v^*(A) = 10 + \gamma v^*(A') \tag{14.6}$$

Now, at the state A', one possible trajectory you can follow is "go up four steps" (each with reward 0) back to A. We know that the optimal value has to be at least as great as this value. That is

$$v^*(A') \geq \gamma^4 v^*(A) \tag{14.7}$$

Combining (14.6) and (14.7), we get

$$v^*(A) \geq 10 + \gamma^5 v^*(A)$$

If we solve for $v^*(A)$, we get

$$v^*(A) \geq \frac{10}{1-\gamma^5} \approx 24.4$$

The value iteration method discussed below is based on this intuition — we can provide a lower bound for the optimal policy by suggesting some potential policy. If we repeat this process, the lower bound for the optimal policy can only go up. At the end of the section, we will prove that this process converges to the actual optimal value.

14.3.2 *Value Iteration Method*

Value Iteration is a method guaranteed to find the optimal policy. At each step of the iteration, we are given a lower bound on the optimal values of each state s. Using the values of the immediate children nodes in the tree, we can compute an improved lower bound on $v^*(s)$.

Example 14.3.2. *See Figure 14.5. Suppose there are two actions to take at state s. The first action, labeled as blue, will lead to state s_1 with reward -1 with probability 0.5 and s_3 with reward -1 with probability 0.5. The second action, labeled as red, will lead to state s_2 with reward 2 with probability 1. The discount factor is given as 0.6. Now assume that someone tells us that they know a way to get expected reward of 12 starting from s_1, 1 from s_2, and 4 from s_3, regardless of the choice of initial action at s. In other words, the optimal values for these three states are lower bounded by: $v^*(s_1) \geq 12, v^*(s_2) \geq 1$ and $v^*(s_3) \geq 4$. Using this fact, we consider two strategies [7] — (1) first take action blue at state s and play optimally thereon based on the other person's knowledge; (2) first take action red at state s and play optimally thereon. The lower bound for the expected reward for each of the two strategies can be computed as:*

[7] This is not necessarily a *policy* because the second part of playing optimally may require you to return to state s and take an action that is inconsistent with your initial choice of action.

$$v_{blue}(s) \geq 0.5 \times (-1 + 0.6 \times 12) + 0.5 \times (-1 + 0.6 \times 4) = 3.8$$
$$v_{red}(s) \geq 1.0 \times (2 + 0.6 \times 1) = 2.6$$

The Bellman Optimality condition in (14.4) guarantees that the optimal policy is at least as good as either of these strategies. Therefore $v^*(s)$ has to be larger than both v_{blue}, v_{red}; that is, $v^*(s) \geq 3.8$.

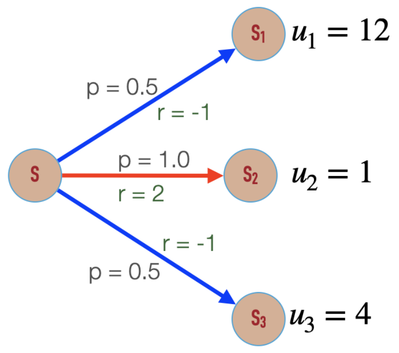


Figure 14.5: There are two actions you can take at state s, and you will end up in one of the three states: s_1, s_2, s_3.

In general, the value iteration algorithm looks like:

1. Initialize some values $v_0(s)$ for each state s such that we are guaranteed $v_0(s) \leq v^*(s)$
2. For each time step $k = 1, 2, \ldots$, and for each state s, use the values $v_k(s')$ of the immediate children s' to compute an updated value $v_{k+1}(s)$ such that $v_{k+1}(s) \leq v^*(s)$. [8]
3. When $k \to \infty$, each $v_k(s)$ will converge to the optimal value $v^*(s)$.

[8] These values $v_k(s)$ maintained by the algorithm is *not* necessarily associated with a specific policy. They are just a lower bound for the optimal value $v^*(s)$ that will be improved over time.

Recall from (14.1) that if all transition rewards are within $[-R, R]$, then the expected rewards at any state for any policy lies in $\left[-\frac{R}{1-\gamma}, \frac{R}{1-\gamma}\right]$.

Therefore, we can set the initial value $v_0(s) = -\frac{R}{1-\gamma}$ to be the lower bound for each state s. [9]

[9] Our proof assumes this special initialization where all $v_0(s) = -\frac{R}{1-\gamma}$ for all states s. It turns out the value iteration method converges to the optimal value for arbitrary initialization, but the proof is more complicated.

After the k-th iteration of the algorithm, we will maintain a value $v_k(s)$ for state s, where the condition $v_k(s) \le v^*(s)$ is maintained as an invariant. Now at the $(k+1)$-th iteration, the algorithm will update the values at each state s as the following:

$$v_{k+1}(s) = \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \qquad (14.8)$$

This is just the Bellman equation evaluated with the values $v_k(s')$ of each children node.

Example 14.3.3 (Example 14.3.1 revisited)**.** *Say we start the value iteration on the gridworld with all values equal to zero. Now let us compute $v_1(A)$, the value of A after the first iteration. Recall that A has only one action to choose from: moving to A'. Denote this action by a. Therefore,*

$$\begin{aligned} v_1(A) &= p(A' \mid A, a) \cdot \big(r(a \mid A, A') + \gamma v_0(A')\big) \\ &= 1.0 \cdot (10 + 0.9 \cdot 0) = 10 \end{aligned}$$

Problem 14.3.4 (Example 14.3.1 revisited)**.** *Start value iteration with all values equal to zero. What is $v_2((1,3))$, the value of $(1,3)$ after second iteration?*

14.3.3 *Why Does Value Iteration Find an Optimum Policy?*

We prove that the values $v_k(s)$ maintained by the value iteration method converge to the optimal values $v_\pi(s)$ in a finite number of steps. We break this proof down in two parts. We first prove that the invariant $v_k(s) \le v^*(s)$ holds throughout the algorithm. Then we prove that in general, $v_{k+1}(s)$ is a tighter lower bound for $v^*(s)$ than $v_k(s)$.

Proposition 14.3.5. *For each time step $k = 1, 2, \ldots$, and for each state s, the invariant $v_k(s) \le v^*(s)$ holds.*

Proof. Proof by mathematical induction. As discussed earlier, our choice of initial values $v_0(s) = -\frac{R}{1-\gamma}$ satisfies the invariant. Now assume that the invariant holds for some k. Now consider the update rule of the value iteration algorithm:

$$v_{k+1}(s) = \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big)$$

Notice that for any specific policy π and for any next state s', we have

$$\begin{aligned} & p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \\ \le & p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big) \end{aligned}$$

because of the inductive hypothesis that $v_k(s') \leq v^*(s')$. Therefore, if we sum over all state s', we have

$$\begin{aligned}&\sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \\ \leq &\sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big)\end{aligned}$$

Since this inequality holds for every policy π, we have the following:

$$\begin{aligned}v_{k+1}(s) &= \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v_k(s')\big) \\ &\leq \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s)) \cdot \big(r(\pi(s) \mid s, s') + \gamma v^*(s')\big) \quad = v^*(s)\end{aligned}$$

where we apply the Bellman Optimality condition (14.4) in the last equality. This concludes the inductive step, and it suffices for the proof. □

Now to prove that these values $v_k(s)$ eventually converge to $v^*(s)$, we introduce the following definition:

Definition 14.3.6. *The* ***residual*** *at s at the k-th iteration is defined as $\delta_{s,k} = v^*(s) - v_k(s) > 0$.*

Notice that as long as the residuals at the k-th iteration converge to 0, the values $v_k(s)$ also converge to $v^*(s)$. Since the residuals take finite values when the algorithm is initiated, it suffices to prove that the residuals decrease non-trivially in every iteration. [10]

[10] Our exposition of Value Iteration with our particular initialization is new. The usual textbook description requires a slightly more complicated argument.

Proposition 14.3.7. *If the largest residual at iteration k is denoted as $\delta_k = \max_s \delta_{s,k}$, then the largest residual δ_{k+1} at iteration $k+1$ satisfies $\delta_{k+1} \leq \gamma \delta_k$*

Proof. Let a^* be the action at s under the optimum policy π^*. Then by (14.2),

$$v^*(s) = \sum_{s'} p(s' \mid s, a^*)(r(a^* \mid s, s') + \gamma v^*(s')) \tag{14.9}$$

Note that taking the action a^* is always an option at the $(k+1)$-th iteration, so the $v_{k+1}(s)$, the maximum value across all actions has to be greater than or equal to the value computed with the action a^*; that is,

$$\begin{aligned}v_{k+1}(s) &= \max_{\pi} \sum_{s'} p(s' \mid s, \pi(s))(r(\pi(s) \mid s, s') + \gamma v_k(s')) \\ &\geq \sum_{s'} p(s' \mid s, a^*)(r(a^* \mid s, s') + \gamma v_k(s')) \qquad (14.10)\end{aligned}$$

Subtracting (14.10) from (14.9), we get

$$v^*(s) - v_{k+1}(s) \le \gamma \left(\sum_{s'} p(s' \mid s, a^*)(v^*(s') - v_k(s')) \right)$$

By the definition of δ_k, each of $v^*(s') - v_k(s') = \delta_{s',k} \le \delta_k$. Therefore,

$$v^*(s) - v_{k+1}(s) \le \gamma \delta_k \sum_{s'} p(s' \mid s, a^*)$$

Finally note that $\sum_{s'} p(s' \mid s, a^*) = 1$ because p is a probability distribution. □

Theorem 14.3.8. *For each $s \in S$, $v_k(s)$ converges to $v^*(s)$ when $k \to \infty$.*

Proof. By Proposition 14.3.5 and Proposition 14.3.7,

$$|v^*(s) - v_k(s)| = v^*(s) - v_k(s) \le \delta_k \le \gamma^k \delta_0$$

which converges to 0 when k goes to infinity. □

14.3.4 Retrieving Optimal Policy from the v^*'s

One important thing to note is that the value iteration method finds the optimal *value* of each state, not the optimal *policy*. So we need an extra step to retrieve the optimal policy from the output of the value iteration algorithm. This can be done by considering the Bellman Optimality condition. For each state s, define $\pi^*(s) = a^*$ such that

$$a^* = \arg\max_{a \in A_s} \sum_{s'} p(s' \mid s, a)(r(a \mid s, s') + \gamma v^*(s')) \qquad \text{(14.5 revisited)}$$

where $v^*(s)$ is the value that the value iteration algorithm converges to. If there are multiple actions a that satisfy the equation above, arbitrarily choose an action.

Example 14.3.9 (Example 14.3.1 revisited)**.** *Say we ran the value iteration algorithm on the Gridworld. The output of the algorithm (the optimal values of each state) is given in Table 14.1.*

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Table 14.1: Optimal values $v^*(s)$ of the Gridworld.

Consider the state $A' = (5, 2)$. There are four actions to take: left-/right/up/down. Each action would yield the following values when evaluat-

ing the Bellman equation:

$$\begin{aligned}
v_{left}(A') &= 0 + 0.9 \times 14.4 = 13.0 \\
v_{right}(A') &= 0 + 0.9 \times 14.4 = 13.0 \\
v_{up}(A') &= 0 + 0.9 \times 17.8 = 16.0 \\
v_{down}(A') &= -1 + 0.9 \times 16.0 = 13.4
\end{aligned}$$

The only action that maximizes the value is the action "go up." Therefore, we can conclude that the optimal policy π^* will adopt the action "go up" for the state A'.

Problem 14.3.10 (Example 14.3.1 revisited)**.** *Verify that an optimal policy can assign either the action "go up" or the action "go left" for the state* $(5, 3)$.

15
Reinforcement Learning in Unknown Environment

In the previous Chapter 14, we established the principles of reinforcement learning using a Markov Decision Process (MDP) with set of states S, set of actions A, transition probabilities $p(s' \mid a, s)$, and the rewards $r(a \mid s, s')$. We saw a method (value iteration) to find the optimal policy that will maximize the expected reward for every state. The main assumption of the chapter was that the agent has access to the full description of the MDP — the set of states, the set of actions, the transition probabilities, rewards, etc.

But what can we do when some of the parameters of the MDP are not available to the agent in advance — specifically, the transition probabilities and the rewards. Instead, the agent makes actions and observes the new state and the reward it just received. Using such experiences it begins to learns the reward and transition structure, and then to translate this incremental knowledge into improved actions.

The above scenario describes most real-life agents: the system designer does not know a full description of the probabilities and transitions. For instance, think of the sets of possible states and transitions in the MuJoCo animals and walkers that we saw. Even with a small number of joints, the total set of scenarios is too vast. Thus the designer can set up an intuitive reward structure and let the learner figure out from experience (which is painless since it involves a simulation).

Settings where agent must determine (or "figure out") the MDP through *experience*, specifically by taking actions and observing the effects, is called the "model-free" setting of RL. This chapter will introduce basic concepts, including the famous Q-learning algorithm.

In many settings today, the underlying MDP is too large for the agent to reconstruct completely, and the agent uses deep neural networks to represent its knowledge of the environment and its own policy.

15.1 Model-Free Reinforcement Learning

In model-free RL, we know the set of states S and the set of actions A, but the transition probabilities and rewards are unknown. The agent now needs to explore the environment to estimate the transition probabilities and rewards. Suppose the agent is originally in state s_1, chooses to take an action a, and ends up in state s_2. The agent immediately observes some reward $r(a \mid s_1, s_2)$, but we need more information to figure out $p(s_2, |s_1, a)$.

One way we can estimate the transition probabilities is through Maximum Likelihood Principle. This concept has been used before when considering estimating unigram probabilities in Chapter 8. In model-free RL, an agent can keep track of the number of times they took action a at state s_1 and ended up in state s_2 — denote this as $\#(s_1, a, s_2)$. Then the estimate of the transition probability $p(s'|s, a)$ is:

$$p(s_2|s_1, a) = \frac{\#(s_1, a, s_2)}{\sum\limits_{s'} \#(s_1, a, s')} \tag{15.1}$$

The Central Limit Theorem (see Chapter 18) guarantees that estimates will improve with more observations and quickly converge to underlying state-action transition probabilities and rewards.

15.1.1 Groundhog Day

Groundhog Day is an early movie about a "time loop" and the title has even become an everyday term. The film tracks cynical TV weatherman Phil Connors (Bill Murray) who is tasked with going to the small town of Punxsutawney and filming its annual Groundhog Day celebration. He ends up reliving the same day over and over again, and becomes temporarily trapped. Along the way, he attempts to court his producer Rita Hanson (Andie MacDowell), and is only released from the time loop after a concerted effort to improve his character.

Sounds philosophically deep! On the internet you can find various interpretations of the movie: *Buddhist* interpretation ("many reincarnations ending in Nirvana") and *psychoanalysis* ("revisiting of the same events over and over again to reach closure"). The RL interpretation is that Phil is in an model-free RL environment, [1] revisiting the same events of the day over and over again and figuring out his optimal actions.

[1] Specifically a model-free RL environment with an ability to *reset* to an initial state. This happens for example with a robot vacuum that periodically returns to its charging station. After charging, it starts exploring the MDP from the initial state again.

15.2 Atari Pong (1972): A Case Study

In 1972, the classic game of Pong was released by Atari. This was the first commercially successful video game, and had a major cultural impact on the perception of video games by the general public. The rules of the game are simple: each player controls a virtual paddle which can move vertically in order to rally a ball back and forth (one participant may be a computer AI). If a player misses the ball, the other player wins a point. We can consider the total number of points accumulated by a player to be their *reward* so far. While technology and video games have become far more advanced in the present, it is still useful to analyze Pong today. This is because it is a simple example of a *physics-based* system, similar to (but far less advanced than) the MuJoCo stick figure simulations discussed in Chapter 13. It thus provides a useful case study to demonstrate how an agent can learn basic principles of physics through random exploration and estimation of transition probabilities.

Let's apply some simplifications in the interest of brevity. We define the pong table to be 5×5 pixels in size, the ball to have a size of 1 pixel, and the paddles to be 2 pixels in height. We define the state at a time t as the locations of the two paddles at time t, and the locations of the ball at time t and time $t-1$. [2]

[2] Storing the location of the ball at time $t-1$ and time t allows us to calculate the difference between the two locations and thus gives an estimate for the velocity.

We additionally restrict attention to the problem of tracking and returning the ball, also known as "Pico Pong." Thus, we define the game to *begin* when the opponent hits the ball. The agent gets a reward of $+1$ if they manage to hit the ball, -1 if they miss, and 0 if the ball is still in play. As soon as the agent either hits the ball or misses, we define that the game *ends*. Of course, these additional rules of the game are not available to the agent playing the game. The agent needs to "learn" these rules by observing the possible states, transitions, and corresponding rewards.

In general, these simplifications remove complications of modeling the opponent and makes the MDP acyclic; an explanatory diagram is shown in Figure 15.1. Throughout this section, we will build intuition about different aspects of our Pico-Pong model through some examples.'

15.2.1 Pico-Pong Modeling: States

Suppose the agent is playing the random paddle movements. Consider the possible states of the game shown in Figure 15.2. We note that out of the three, the third option is never seen. By the definition of the game, the ball can never move away from the agent. Of course, the agent is oblivious to this fact at first, but once the game proceeds,

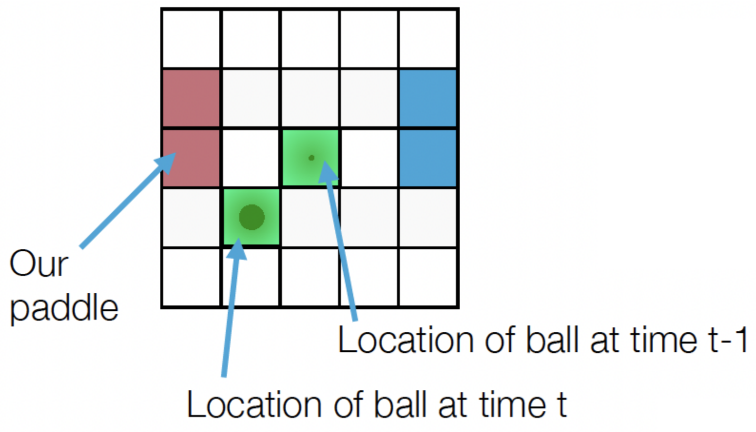


Figure 15.1: The simplified Pico-Pong setup which will be considered in this case study.

the agent will be able to implicitly "learn" that the ball can never move away from them.

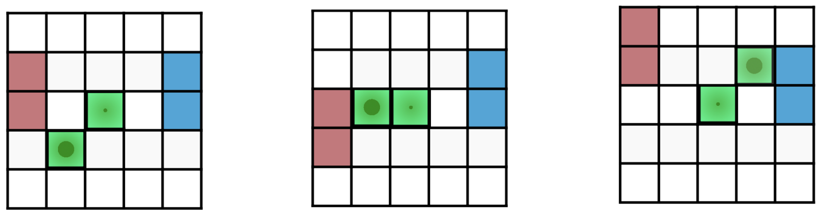


Figure 15.2: Out of these possible states, the third option is never seen.

15.2.2 *Pico-Pong Modeling: Transitions*

Let us now add another restriction to the game that the ball always moves at a speed of 1 pixel every time step (*i.e.*, moves to one of the 8 adjacent pixels) and in a straight linear path unless being bounced against the top/bottom wall. Consider the possible transitions shown in Figure 15.3. We note that out of the three, the third option is never seen. By the restriction of the game, the ball cannot move 2 pixels in one time step. The agent thus implicitly "learns" that the ball moves at a constant speed of 1 pixel per time step.

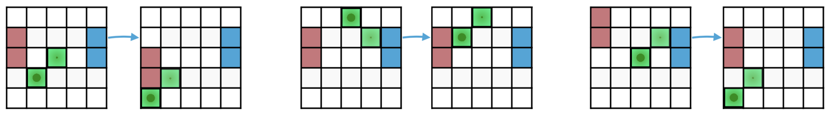


Figure 15.3: Out of these possible transitions, the third option is never seen.

Problem 15.2.1. *Suppose the agent is playing randomly and the ball is traveling at a speed of* 1 *pixel per step. Which of the transitions in Figure 15.4 is never seen, and why?*

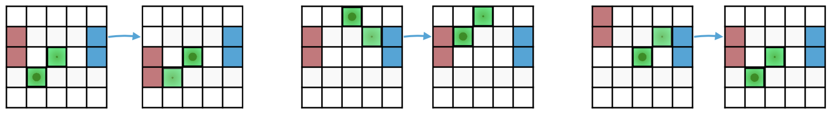


Figure 15.4: Out of these possible transitions, one option is never seen.

15.2.3 *Pico-Pong Modeling: Rewards*

Suppose the agent is playing randomly and the ball is traveling at a speed of 1 pixel per step. Consider the action in Figure 15.5. We note that the associated reward will be $+1$ because in the resulting state the agent has "hit" the ball. The agent thus implicitly learns that if the ball is 1 pixel away horizontally, it should move towards it to obtain a positive reward.

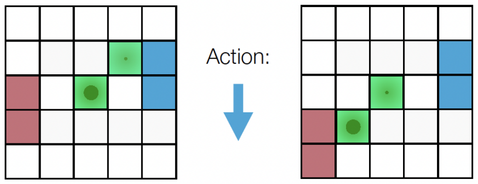


Figure 15.5: Taking action ↓ results in a reward of $+1$.

Problem 15.2.2. *Suppose the agent is playing randomly and the ball is traveling at a speed of* 1 *pixel per step. What reward is achieved given the current state and chosen action in Figure 15.6, and why?*

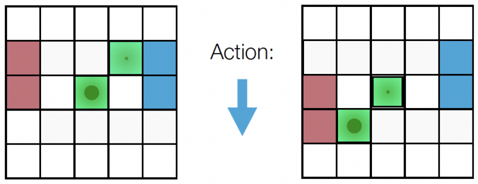


Figure 15.6: What reward will result when taking action ↓?

15.2.4 *Playing Optimally in the Learned MDP*

After allowing the agent to *explore* enough, the agent has "learnt" some information about the underlying MDP of the Pico-Pong model. First thing the agent can learn is that, out of all possible states, there is a subset of states that never appear in the game (*e.g.*, ball moving away from the agent or ball moving too fast). The agent will be able to ignore these states, while learning how to play optimally in states that *did* occur while exploring.

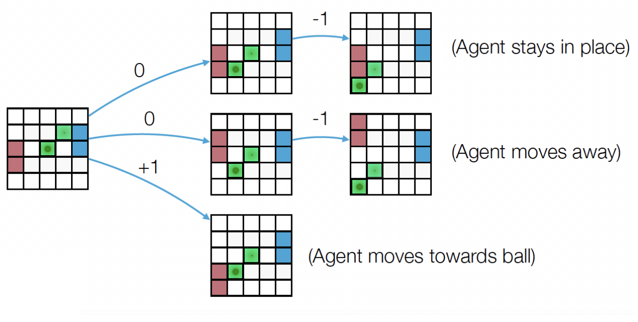


Figure 15.7: An example look-ahead tree for the Pico-Pong model.

Also, the agent has now "learnt" the transition probabilities and rewards of the MDP. Using these estimates, the agent is able to build up a representation of the MDP. Since the underlying MDP for the simplified Pico-Pong model is acyclic, the optimal policy can be determined using a simple look-ahead tree. An example diagram is shown in Figure 15.7.

We provide a specific example to aid the exposition. Suppose an agent finds themselves in the state shown in Figure 15.8. Since the path of the ball is already determined, the next possible state is uniquely determined by the choice of the action — "go down" or "stay in place" or "go up." If the agent chooses to "go down," the game will end with a reward of $+1$. If the agent chooses to "stay in place" or "go up," the game continues for another time step, but no matter the choice of action on that step, the game will end with a reward of -1. Therefore, the agent will learn that the optimal policy will assign the action of "go down" in the state shown in Figure 15.8.

Problem 15.2.3. *Draw out the look-ahead tree from the state shown in Figure 15.8.*

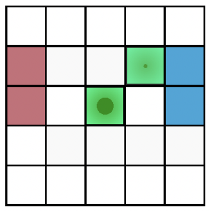


Figure 15.8: A sample state in the game play of Pico Pong.

Problem 15.2.4. *Suppose we start from the state shown in Figure 15.9. Assuming optimal play, what is the expected reward for the agent? (Hint: consider if the agent be able to reach the ball in time)*

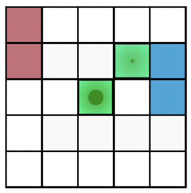


Figure 15.9: A sample state in the game play of Pico Pong.

Impressive! The agent has learnt how to return the ball in Pico-Pong by first building up the MDP and its transitions/rewards through repeated observations, and then computing the optimum policy for the constructed MDP through a look-ahead tree. [3]

[3] How would you extend these ideas to design a rudimentary ping pong robot which can track and return the ball?

15.3 *Q-learning*

15.3.1 *Exploration vs. Exploitation*

Let us analyze the case study with Pico-Pong more deeply. We can separate the process of learning into two different stages — *exploration* and *exploitation*:

- *Exploration:* This pertains to what the agent did in the first phase. Random paddle movements were used to help build up previously unknown knowledge of the MDP — transition probabilities and rewards.
- *Exploitation:* This pertains to what the agent did in the second phase. Specifically, the agent used the learnt MDP to play optimally.

In general, an RL environment is more complicated than Pico Pong, and there is no clear-cut boundary of when an agent has explored "sufficiently." It is best to combine the two stages (*i.e.*, exploration and exploitation) into one and "learn as you go." Also, it is difficult to balance between these two processes, and how to find the correct trade-off between exploration and exploitation is a recurring topic in RL.

15.3.2 *Q-function*

We now introduce the *Q-function*, an important concept that helps tie together concepts of exploration and exploitation when considering general MDPs with discounted rewards.

Definition 15.3.1 (Q-function). *We define the **Q-function** $Q : S \times A \to \mathbb{R}$ as a table which assigns a real value $Q(s, a)$ to each pair (s, a) where $s \in S$ and $a \in A$.*

Intuitively, the value $Q(s,a)$ is the current *estimate* of the *expected discounted reward when we take action a from state s*. In other words, it is the estimate of the value $v_\pi(s)$ if π is any policy that will assign the action a to state s. Using the currently stored values of the Q-function, we can define a canonical policy π_Q. For each state s, the policy will assign the action a that maximizes the $Q(s,a)$ value; that is,

$$\pi_Q(s) = \arg\max_a Q(s,a)$$

Since the agent only has access to the estimate values $Q(s,a)$, but not the actual value function v, this is the most optimal policy to the agent's knowledge. Therefore, if the agent choose to take an exploitation step, they will take an action prescribed by the policy π_Q with respect to the currently maintained Q-function.

Instead of relying on the currently stored Q-function, we can also choose to take an exploration step. Every time we take an exploration step and receive additional information about the RL environment, we update the values of the Q-function accordingly. The goal of the Q-learning is to learn the *optimal Q-function*, which approximates the optimal policy π^* and the optimal value function v^* as closely as possible. We formalize the notion as follows:

Definition 15.3.2 (Optimal Q-function)**.** *The* ***optimal Q-function*** *is a Q-function that satisfies the following two conditions:*

- *The corresponding canonical policy π_Q is an optimal policy for the MDP.*
- *The Q-function satisfies the following condition:*

$$Q(s,a) = \sum_{s';\, a} p(s' \mid s,a)(r(a \mid s,s') + \gamma \max_b Q(s',b)) \tag{15.2}$$

The first condition of Definition 15.3.2 states that for a fixed state s, the action a that maximizes $Q(s,a)$ is $a = \pi^*(s)$. This condition only cares about the relative ordering of the values of $Q(s,a)$ — as long as $Q(s,\pi^*(s))$ is the maximum value among all $Q(s,a)$, then it is fine. This condition guarantees that the action we take in the exploitation step is an optimal action.

The second condition is formally stating that the values of the Q-function are estimates of the expected reward when we take aciton a from state s. It also suggests that Q-function needs to "behave like" a value function v_π for some policy π. However, whereas a similar condition for a value function v_π only needs to hold for one particular action (*i.e.*, $a = \pi(s)$) given a state s, this condition for a Q-function should hold for any arbitrary action a. Note that for an optimal Q-function, the term $\max_b Q(s',b)$ in (15.2) is equivalent to $v_{\pi_Q}(s')$.

15.3.3 *Q-learning*

Now that we have defined the Q-function and the optimal Q-function, it is time for us to study how to learn the optimal Q-function. This process is called *Q-learning*. The basic idea is to probabilistically choose between exploration or exploitation: we define some probability $\epsilon \in [0, 1]$ such that we choose a random action a with probability ϵ (exploration) or choose the action a according to the current canonical policy π_Q with probability $1 - \epsilon$ (exploitation). If we choose the exploration option, we use its outcome to update the $Q(s, a)$ table. But how should we define the update rule?

Let's take a step back and consider a (plausibly?) real life scenario. You are a reporter for the Daily Princetonian at Princeton, and want to estimate the average wealth of alumni at a Princeton Reunions event. The alumni, understandably vexed by such a request, strike a compromise that you are only allowed to ask *one* alum about their net worth. Can you get an estimate of the average? Well, you could pick an alum at random and ask them their net worth! [4]

[4] The expectation gives the right average. But typically the answer would be far from the true average; especially if Jeff Bezos happens to be attending the reunion.

With this intuition, we return to the world of Q-learning. Suppose you start at some state s_t, take an action a_t, receive a reward of r_t, and arrive at state s_{t+1}. We call this process an *experience*. Now, when we update the current estimate of $Q(s_t, a_t)$, we ideally want to mimic the behavior of the optimal Q-function in (15.2) and update it to:

$$Q(s_t, a_t) = \sum_{s';\, a_t} p(s' \mid s_t, a_t) \cdot (r(a_t \mid s_t, s') + \gamma \max_b Q(s', b)) \tag{15.3}$$

Notice that this is the weight average of the expected reward $r(a_t \mid s_t, s') + \gamma \max_b Q(s', b)$ over all possible next state s' given the action a_t. But in practice, the agent only has the ability to take a *single* experience; they lack the ability to "rest" and try all states s' according to the transition probability $p(s' \mid s_t, a_t)$. We thus must consider an alternative idea — we define the *estimate* for $Q(s_t, a_t)$ according to the experience at time step t as

$$Q'_t = r_t + \max_b Q(s_{t+1}, b)$$

This estimate can be calculated using the observed reward r_t and looking up the Q values of the state s_{t+1} on the Q-function table. Note that the expectation of Q'_t is exactly the right hand side of (15.3). That is,

$$\mathbb{E}[Q'_t] = \sum_{s';\, a_t} p(s' \mid s_t, a_t) \cdot (r(a_t \mid s_t, s') + \gamma \max_b Q(s', b))$$

This is because the agent took a transition to state s_{t+1} with probability $p(s_{t+1} \mid s_t, a_t)$ (of course, the agent does not know this value). This is thus analogous to the single-sample estimate of average alumni wealth at the Princeton Reunions event. We can now define the following update rule of the Q-learning process:

$$\begin{aligned} Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \eta(Q'_t - Q(s_t, a_t)) \\ &= (1-\eta)Q(s_t, a_t) + \eta Q'_t \end{aligned} \tag{15.4}$$

for some learning rate $\eta > 0$. You can understand this update rule in two different ways. First, we are gently nudging the value of $Q(s_t, a_t)$ towards the estimate Q'_t from the most recent experience. We can alternatively think of the updated value of $Q(s_t, a_t)$ as the weighted average of the previous value of $Q(s_t, a_t)$ and the estimate Q'_t. In either approach, the most important thing to note is that we combine both the previous Q value and the new estimate to compute the updated Q value. This is because the new estimate is just a single sample that can be far off from the actual expectation, and also because after enough iterations, we can assume the previous Q value to contain information from past experience.

Example 15.3.3. *Let's return to our adventures in Pico-Pong and consider the situation in Figure 15.10. Denote the state in the left diagram as s_t and the state in the right as s_{t+1}. Suppose the current value of $Q(s_t, a) = 0.4$ with $a = \uparrow$. Assuming that $Q(s_{t+1}, a) = 0$ for all a, we can compute the estimate Q'_t from this experience as*

$$Q'_t = r_t + \max_b Q(s_{t+1}, b) = 1$$

Then the Q value will be increased to $0.4 + 0.6\eta$.

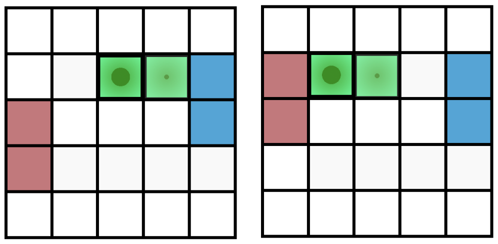


Figure 15.10: The diagram representing two states in a game of Pico-Pong.

15.3.4 Deep Q-learning

Note that the update rule in (15.4) looks similar to the Gradient Descent algorithm. They are both iterative processes which incorporate a learning rate η. In fact, you can consider the Q-learning update rule to be trying to minimize the squared difference between $Q(s_t, a_t)$ and Q'_t. The similarity between the Q-learning update rule and the Gradient Descent algorithm allows us to utilize deep neural network to learn the optimal Q-function. Such a network is called the *Deep Q Network (DQN).*

In a DQN, the Q-function can be represented by the parameters $\mathbf{W}$ of the network. We emphasize this by denoting the Q-function as $Q_{\mathbf{W}}(s, a)$. Now instead of directly updating the Q-function as in the update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta(Q'_t - Q(s_t, a_t)) \qquad \text{(15.4 revisited)}$$

we instead update the parameters $\mathbf{W}$ such that the Q-function is updated accordingly.

First consider the case that $Q'_t > Q(s_t, a_t)$. That is, the estimated Q-value is larger than the currently stored value. Then the update rule (15.4) will increase the value of $Q(s_t, a_t)$. To mimic this behavior, we want to find an update rule for $\mathbf{W}$ that will increase the Q-value. This is given as:

$$\mathbf{W} \leftarrow \mathbf{W} + \beta \cdot \nabla_{\mathbf{W}} Q_{\mathbf{W}}(s_t, s_a)$$

for some learning rate $\beta > 0$.

Problem 15.3.4. *Suppose $Q'_t < Q(s_t, a_t)$. How should we design the weight updates?*

One final thing to note is a technique called *experience replay*. Experiencing the environment can be expensive (*i.e.*, computation time, machine wear, etc.). Therefore, it is customary to keep a history of old experiences and their rewards, and periodically take a random sample out of the old experiences to update the Q values. In particular, experience replay ensures that DQNs are efficient and avoid "catastrophic forgetting." [5]

[5] Catastrophic forgetting is a phenomenon where a neural network, after being exposed to new information, "forgets" information it had learned earlier

15.4 *Applications of Reinforcement Learning*

15.4.1 *Q-learning for* Breakout *(1978)*

We previously considered using reinforcement learning for *Pong*. We can also use it for another famous Atari game called *Breakout*. One particular design uses a CNN to process the screen and uses the "score" as a reward. As shown in Figure 15.11, the model becomes quite successful after several epochs.

15.4.2 *Self-help Apps*

Self-help apps are designed to aid in recovery of the user from addiction, trauma heart disease, etc. A typical design involves an RL algorithm which determines the next advice/suggestion based upon reversals, achieved milestones, etc. so far. These can be a helpful supplement to expensive therapy/consultation.

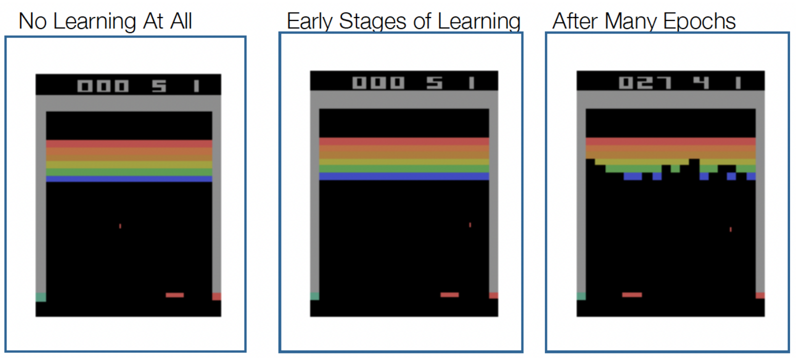


Figure 15.11: An application of Q-learning to the famous Atari game *Breakout*.

15.4.3 *Content Recommendation*

At reputable websites, we might imagine that there exists a page creation system designed to capture the "reward" of user engagement. We can use MDP techniques to model this situation. Specifically, we can define s_0 as the outside link which brought the user to the landing page and/or the past history of the user on the site. If the user clicks on a link, a new page is created and we can define s_1 as a concatenation of s_0 and the new link. If the user again clicks on a link, another new page is created and we can define s_2 as the concatenation of s_1 and the new link.

15.5 *Deep Reinforcement Learning*

Deep Reinforcement Learning is a subfield of machine learning that combines the methods of Deep Learning and Reinforcement Learning that we have discussed earlier. [6] The goal of it is to create an artificial agent with human-level intelligence (AGI). In general, Reinforcement Learning defines the objective and Deep Learning gives mechanism for optimizing that objective. Deep RL method combines the problem given by the RL with the solution given by the DL. In the cited source video, RL expert David Silver made three broad conjectures related to this topic.

[6] Source: `https://www.youtube.com/watch?v=x5Q79XCxMVc`

1. RL is enough to formalize the problem of intelligence
2. Deep neural networks can represent and learn any computable function
3. Deep RL can solve the problem of intelligence

Many Deep RL models are trained to play games (*e.g.*, chess, Go) because it is easy to evaluate progress. By letting them compete against humans, we can easily compare them to human-level intelligence. As an example, Google Deepmind trained a Deep RL model called DQN to play 49 arcade games. [7] The computer is not given the explicit set of rules; instead, given only the pixels and game score as input, it learns by using deep reinforcement learning to maximize its score. Amazingly, on about half of the games, the model played at least at human level of intelligence!

[7] For the full paper, visit `https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf`

15.5.1 *Chess: A Case Study*

Founders of AI considered chess to be the epitome of human intelligence. In principle, the best next move can be calculated via a look-ahead tree (similar to Figure 13.5 from the cake-eating example). Since chess is a two-player game, we can use an algorithm called the *min-max search* on the look-ahead game tree. [8]

[8] Source: `https://www.youtube.com/watch?v=l-hh51ncgDI`

Usually, RL agents are playing against the nature that causes them to take random transitions according to the MDP's transition probabilities. But in chess, the agent plays against an opponent that is trying to make you take the largest possible loss (the largest possible gain for the opponent). That is why we need a min-max evaluation of the look-ahead tree.

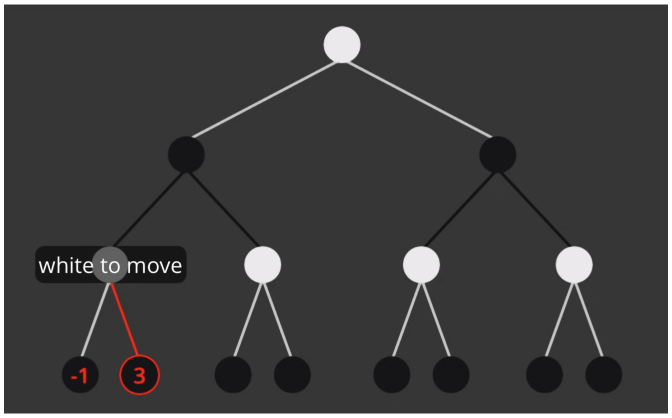


Figure 15.12: An example look-ahead game tree for chess with depth 3. White will choose the right option.

In Figure 15.12, the numbers at the leaf nodes represent a static evaluation of how good the game configuration is for white. This is an approximation for the actual value of the node. An example metric in chess would be the difference in the number of pieces (# white − # black). These numbers are evaluated either when the game terminates or when the algorithm has reached the specified number of steps to look ahead. If the game ever reaches the specified node, the white has two options to choose from: if white chooses the left child node, it will end up with reward of -1; whereas if it chooses the right child node, the reward will be 3. Then to maximize reward, the best move of white will be to choose 3. [9]

[9] For those who are familiar with chess or game theory in general, this is known as the *best response*.

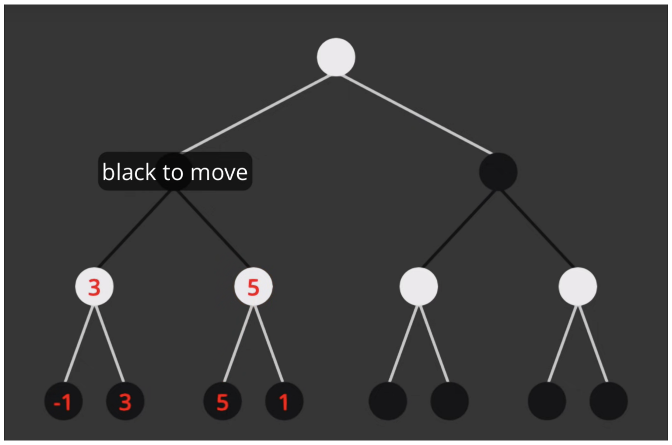


Figure 15.13: Black will choose the left option.

In Figure 15.13, it is now black's turn to choose. Note that the reward for black is the opposite of the reward for white, so black wants to *minimize* the value on the tree. Therefore, black will want to choose the left child node.

So whenever we are at a configuration, we can create a look-ahead tree for a reasonable number of steps and try to calculate the best move. But the size of a game tree is astronomical, so it is computationally infeasible to search all levels of the tree. [10]

[10] There is an optimization method called the alpha-beta pruning. Consult the video referenced above for an implementation on the game of chess.

15.5.2 *AlphaGo: A Case Study*

Go is a game invented in China around 500 BC. It is played by 2 players on a 19×19 grid. Players take turns placing stones on the grid, and if any set of stones is entirely surrounded by opponent stones, the enclosed stones are taken away from the board and awarded to the opponent as points. Even though the rules are very simple, no computer could beat a good human amateur at Go until 2015. [11]

[11] In comparison, IBM's Deep Blue model beat the world chess champion Kasparov in 1997.

How can we utilize RL concepts to play this game? In general, we can create a Deep Policy Net (DPN) to learn W, which is a function that takes state s as an input and outputs a probability distribution $p_W(a \mid s)$ over the next possible actions from s. AlphaGo is an example of a DPN engineered by the Google Deepmind lab. It takes the current board position as the input and uses ConvNet to learn the internal weights, and outputs the value given by a softmax function. In its initial setup, the DPN was trained using a big dataset of past games. [12]

[12] Source: `https://www.youtube.com/watch?v=Wujy7OzvdJk`

To be more specific, AlphaGo used supervised learning from human data to learn the optimal policy (action to take at each game setting). In other words, it used convolutional layers to replicate the moves of professional players as closely as possible. Since the CNN is just mimicking human players, it cannot beat human champions.

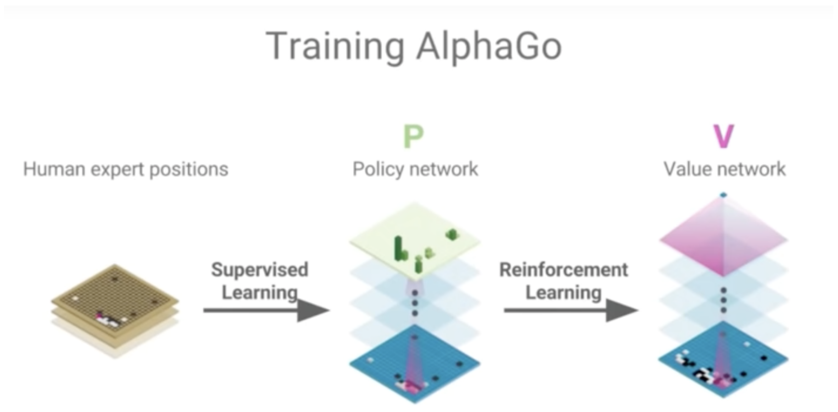


Figure 15.14: The diagram representing the process of training AlphaGo.

However, it can be used to search the full game tree more efficiently than the alpha-beta search. Formally, this method is called the Monte Carlo Tree Search, where the CNN is used to decide the order in which to explore the tree. After the policy network was sufficiently trained, reinforcement learning was used to train the value network for position evaluation. Given a board setting, the network was trained to estimate the value (*i.e.*, likelihood of winning) of that setting.

AlphaGo Zero is a newer version of the model that does not depend on any human data or features. In this model, policy and value networks are combined into one neural network, and the model does not use any randomized Monte-Carlo simulations. It learns solely by self-play reinforcement learning and uses neural network (resnet) to evaluate its performance. Within 3 days of training, AlphaGo Zero surpassed an earlier version of AlphaGo that beat Lee Se Dol, the holder of 8 world titles; within 21 days, it surpassed the version that beat Ke Jie, the world champion. Interestingly enough, AlphaGo Zero adopted some opening patterns commonly played by human players, but it also discarded some common human patterns and it also discovered patterns unknown to humans.

The newest version of AlphaGo is called the AlphaZero. It is a model that can be trained to play not just Go but simultaneously Chess and Shogi (Japanese chess). After just a few hours of training, AlphaZero surpassed the previous computer world champions (*Stockfish* in Chess, *Elmo* in Shogi, and *AlphaGo Zero* in Go). Just as AlphaGo Zero did, AlphaZero was able to dynamically adopt or discard known openings in chess.