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:

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

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,

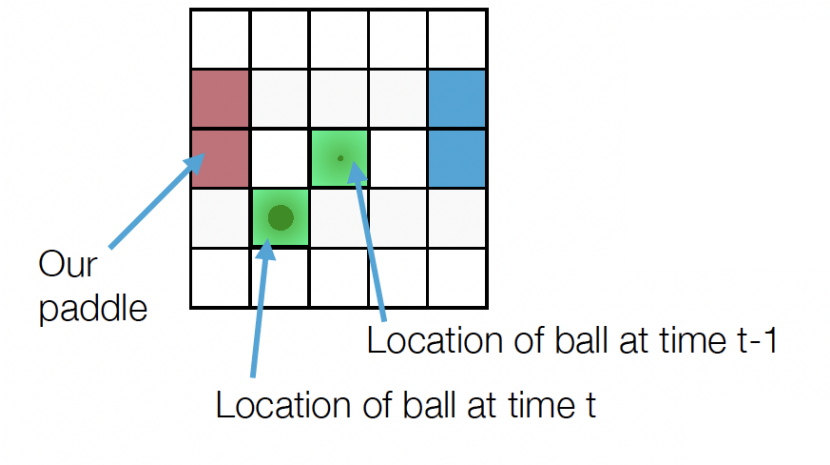

Figure 15.1: The simplified Pico-Pong setup which will be considered in this case study.

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

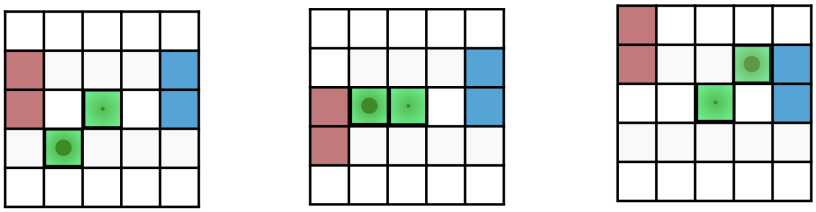

Figure 15.2: Out of these possible states, the third option is never seen.

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.

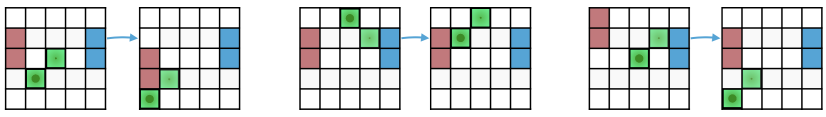

Figure 15.3: Out of these possible transitions, the third option is never seen.

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?*

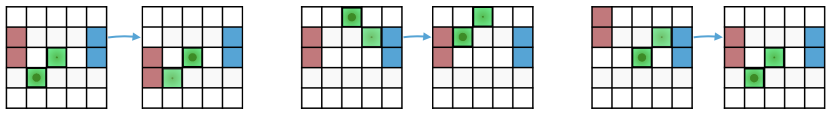

Figure 15.4: Out of these possible transitions, one option is never seen.

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.

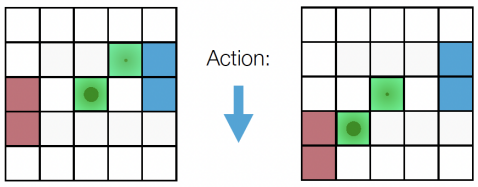

Figure 15.5: Taking action $\downarrow$ results in a reward of $+1$.

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?*

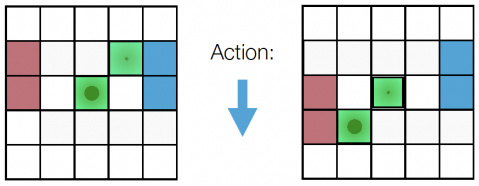

Figure 15.6: What reward will result when taking action $\downarrow$?

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.

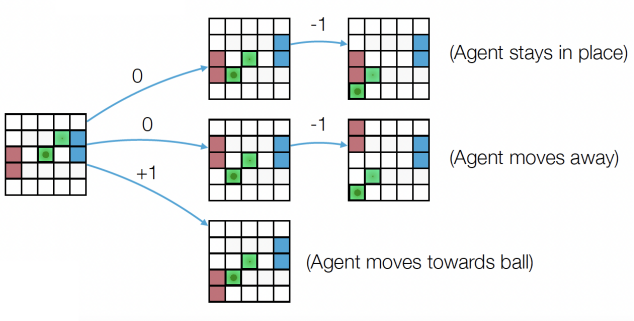

Figure 15.7: An example look-ahead tree for the Pico-Pong model.

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.*

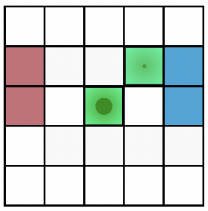

Figure 15.8: A sample state in the game play of Pico Pong.

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)*

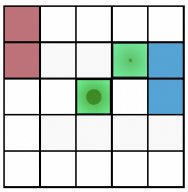

Figure 15.9: A sample state in the game play of Pico Pong.

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:

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

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 $0.4 + 0.6\eta$.

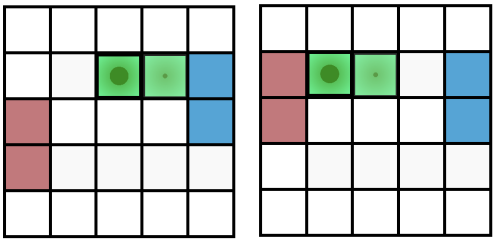

Figure 15.10: The diagram representing two states in a game of Pico-Pong.

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.

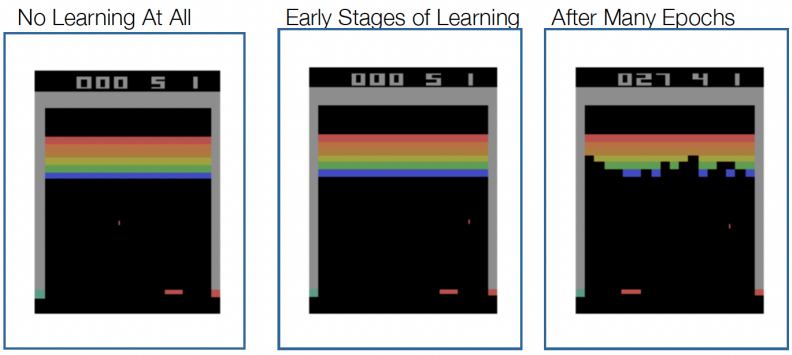

Figure 15.11: An application of Q-learning to the famous Atari game *Breakout*.

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.

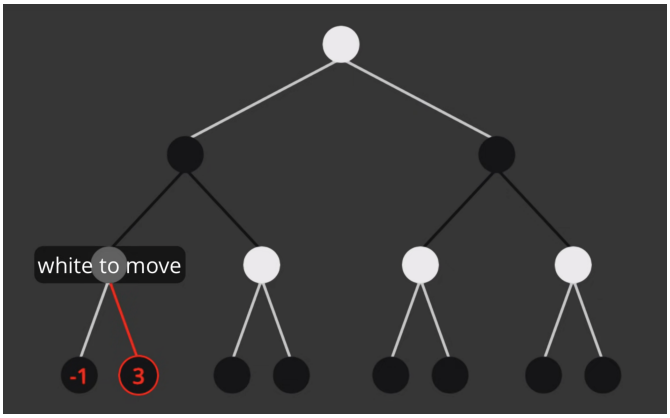

Figure 15.12: An example look-ahead game tree for chess with depth 3. White will choose the right option.

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*.

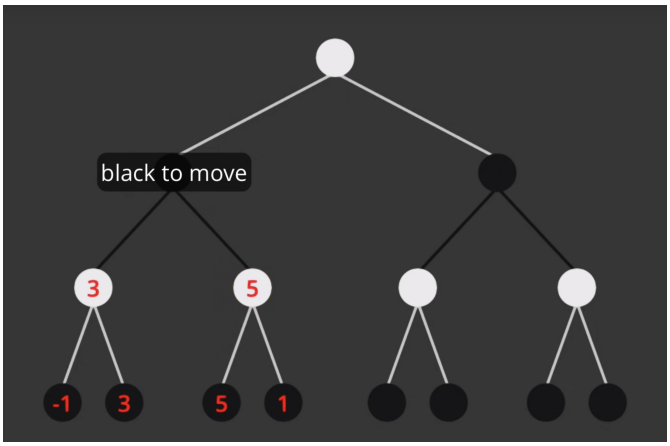

Figure 15.13: Black will choose the left option.

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.