13. Deep Reinforcement Learning

In Chapter 4, we introduced the paradigm of reinforcement learning (as distinct from supervised and unsupervised learning), in which an agent (e.g., an algorithm) takes sequential actions within an environment. The environments—whether they be simulated or real world—can be extremely complex and rapidly changing, requiring sophisticated agents that can adapt appropriately in order to succeed at fulfilling their objective. Today, many of the most prolific reinforcement learning agents involve an artificial neural network, making them *deep reinforcement learning* algorithms.

In this chapter, we will

- Cover the essential theory of reinforcement learning in general and, in particular, a deep reinforcement learning model called deep Q-learning
- Use Keras to construct a deep Q-learning network that learns how to excel within simulated, video game environments
- Discuss approaches for optimizing the performance of deep reinforcement learning agents
- Introduce families of deep RL agents beyond deep Q-learning

ESSENTIAL THEORY OF REINFORCEMENT LEARNING

Recall from Chapter 4 (specifically, Figure 4.3) that *reinforcement learning* is a machine learning paradigm involving:

- An *agent* taking an *action* within an *environment* (let's say the action is taken at some timestep t).
- The environment returning two types of information to the agent:
    1. *Reward*: This is a scalar value that provides quantitative feedback on the action that the agent took at timestep t. This could, for example, be 100 points as a reward for acquiring cherries in the video game Pac-Man. The agent's objective is to maximize the rewards it accumulates, and so rewards

are what *reinforce* productive behaviors that the agent discovers under particular environmental conditions.

2. *State*: This is how the environment changes in response to an agent's action. During the forthcoming timestep ($t + 1$), these will be the conditions for the agent to choose an action in.

- Repeating the above two steps in a loop until reaching some terminal state. This terminal state could be reached by, for example, attaining the maximum possible reward, attaining some specific desired outcome (such as a self-driving car reaching its programmed destination), running out of allotted time, using up the maximum number of permitted moves in a game, or the agent dying in a game.

Reinforcement learning problems are sequential decision-making problems. In Chapter 4, we discussed a number of particular examples of these, including:

- Atari video games, such as Pac-Man, Pong, and Breakout
- Autonomous vehicles, such as self-driving cars and aerial drones
- Board games, such as Go, chess, and shogi
- Robot-arm manipulation tasks, such as removing a nail with a hammer

The Cart-Pole Game

In this chapter, we will use OpenAI Gym—a popular library of reinforcement learning environments (examples provided in Figure 4.13)—to train an agent to play Cart-Pole, a classic problem among academics working in the field of control theory. In the Cart-Pole game:

- The objective is to balance a pole on top of a cart. The pole is connected to the cart at a purple dot, which functions as a pin that permits the pole to rotate along the horizontal axis, as illustrated in Figure 13.1. [1]

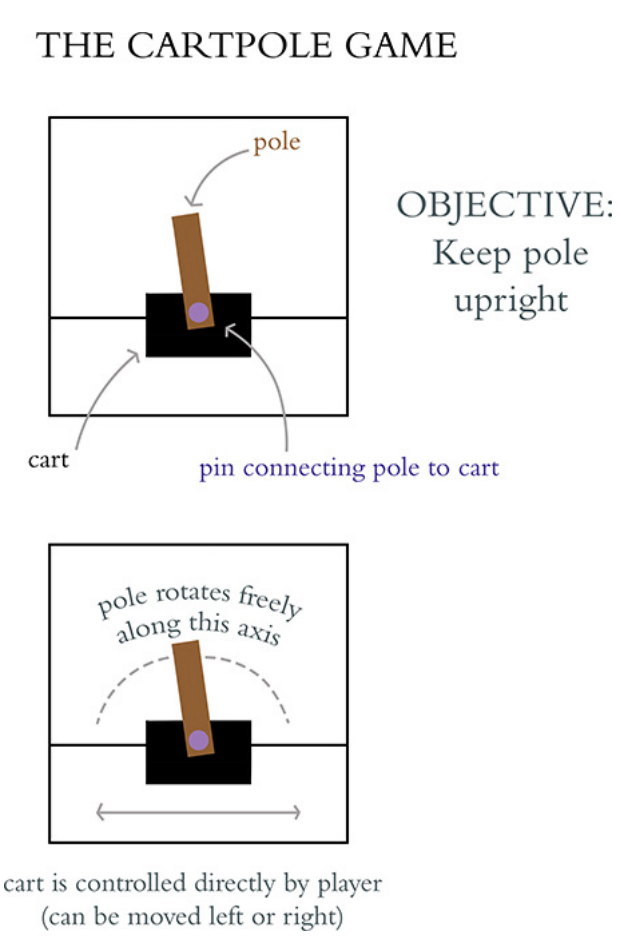

Figure 13.1 The objective of the Cart-Pole game is to keep the brown pole balanced upright on top of the black cart for as long as possible. The player of the game (be it a human or a machine) controls the cart by moving it horizontally to the left or to the right along the black line. The pole moves freely along the axis created by the purple pin.

- The cart itself can only move horizontally, either to the left or to the right. At any given moment—at any given *timestep*—the cart *must* be moved to the left or to the right; it can't remain stationary.
- Each episode of the game begins with the cart positioned at a random point near the center of the screen and with the pole at a random angle near vertical.
- As shown in Figure 13.2, an episode ends when either
  - The pole is no longer balanced on the cart—that is, when the angle of the pole moves too far away from vertical toward horizontal
  - The cart touches the boundaries—the far right or far left of the screen
- In the version of the game that you'll play in this chapter, the maximum number of timesteps in an episode is 200. So, if the episode does not end early (due to losing pole balance or navigating off the screen), then the game will end after 200 timesteps.
- One point of reward is provided for every timestep that the episode lasts, so the maximum possible reward is 200 points.

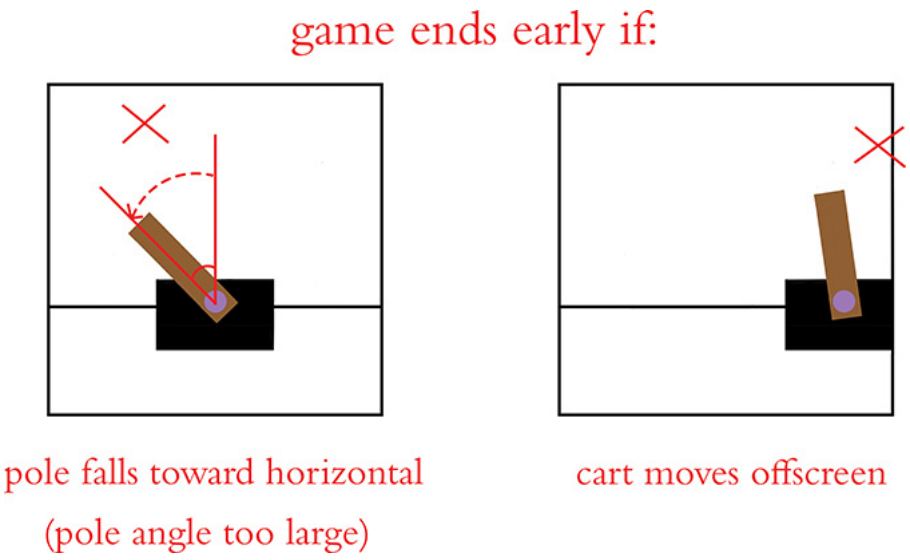

Figure 13.2 The Cart-Pole game ends early if the pole falls toward horizontal or the cart is navigated off-screen.

1 . An actual screen capture of the Cart-Pole game is provided in Figure 4.13a.

The Cart-Pole game is a popular introductory reinforcement learning problem because it's so simple. With a self-driving car, there are effectively an infinite number of possible environmental states: As it moves along a road, its myriad sensors—cameras, radar, sonar, lidar, [2] accelerometers, microphones, and so on—stream in broad swaths of state information from the world around the vehicle, on the order of a gigabyte of data per second. [3] The Cart-Pole game, in stark contrast, has merely four pieces of state information:

1. The position of the cart along the one-dimensional horizontal axis

2. The cart's velocity

3. The angle of the pole

4. The pole's angular velocity

2 . Same principle as sonar, but uses lasers instead of sound.

3 . `bit.ly/GBpersec`

Likewise, a number of fairly nuanced actions are possible with a self-driving car, such as accelerating, braking, and steering right or left. In the Cart-Pole game, at any given timestep *t*, exactly one action can be taken from only two possible actions: move left or move right.

Markov Decision Processes

Reinforcement learning problems can be defined mathematically as something called a *Markov decision process*. MDPs feature the so-called *Markov property*—an assumption that the current timestep contains

all of the pertinent information about the state of the environment from previous timesteps. With respect to the Cart-Pole game, this means that our agent would elect to move right or left at a given timestep *t* by considering only the attributes of the cart (e.g., its location) and the pole (e.g., its angle) at that particular timestep *t*. [4]

4 . The Markov property is assumed in many financial-trading strategies. As an example, a trading strategy might take into account the price of all the stocks listed on a given exchange at the end of a given trading day, while it does *not* consider the price of the stocks on any previous day.

As summarized in Figure 13.3, the MDP is defined by five components:

1. **S** is the set of all possible *states*. Following set-theory convention, each individual possible state (i.e., a particular combination of cart position, cart velocity, pole angle, and angular velocity) is represented by the lowercase **s**. Even when we consider the relatively simple Cart-Pole game, the number of possible recombinations of its four state dimensions is enormous. To give a couple of coarse examples, the cart could be moving slowly near the far-right of the screen with the pole balanced vertically, or the cart could be moving rapidly toward the left edge of the screen with the pole at a wide angle turning clockwise with pace.

2. **A** is the set of all possible *actions*. In the Cart-Pole game, this set contains only two elements (*left* and *right*); other environments have many more. Each individual possible action is denoted as **a**.

3. **R** is the distribution of *reward* given a *state-action pair*—some particular state paired with some particular action—denoted as (**s**, **a**). It's a distribution in the sense of being a probability distribution: The exact same state-action pair (**s**, **a**) might randomly result in different amounts of reward **r** on different occasions. [5] The details of the reward distribution **R**—its shape, including its mean and variance—are hidden from the agent but can be glimpsed by taking actions within the environment. For example, in Figure 13.1, you can see that the cart is centered within the screen and the pole is angled slightly to the left. [6] We'd expect that pairing the action of moving left with this state **s** would, on average, correspond to a higher expected reward **r** relative to pairing the action of moving right with this state: Moving left in this state **s** should cause the pole to stand more upright, increasing the number of timesteps that the pole is kept balanced for, thereby tending to lead to a higher reward **r**. On the other hand, the move right in this state **s** would increase the probability that the pole would fall toward horizontal, thereby tending toward an early end to the game and a smaller reward **r**.

4. $\mathbb{P}$, like **R**, is also a probability distribution. In this case, it represents the probability of the next state (i.e., $\mathbf{s}_{t+1}$) given a particular state-action pair (**s**, **a**) in the current timestep *t*. Like **R**, the $\mathbb{P}$ distribution is hidden from the agent, but again aspects of it can be inferred by taking actions within the environment. For example, in the Cart-Pole game, it would be relatively straightforward for the agent to learn that the *left* action corresponds directly to the cart moving leftward. [7] More-complex relationships—for example, that the *left* action in the state **s** captured in Figure 13.1 tends to

correspond to a more vertically oriented pole in the next state s_{t+1}—would be more difficult to learn and so would require more gameplay.

5. γ (gamma) is a hyperparameter called the *discount factor* (also known as *decay*). To explain its significance, let's move away from the Cart-Pole game for a moment and back to Pac-Man. The eponymous Pac-Man character explores a two-dimensional surface, gaining reward points for collecting fruit and dying if he gets caught by one of the ghosts that's chasing him. As illustrated by Figure 13.4, when the agent considers the value of a prospective reward, it should value a reward that can be attained immediately (say, 100 points for acquiring cherries that are only one pixel's distance away from Pac-Man) more highly than an equivalent reward that would require more timesteps to attain (100 points for cherries that are a distance of 20 pixels away). Immediate reward is more valuable than some distant reward, because we can't bank on the distant reward: A ghost or some other hazard could get in Pac-Man's way. [8, 9] If we were to set $\gamma = 0.9$, then cherries one timestep away would be considered to be worth 90 points,[10] whereas cherries 20 timesteps away would be considered to be worth only 12.2 points.[11]

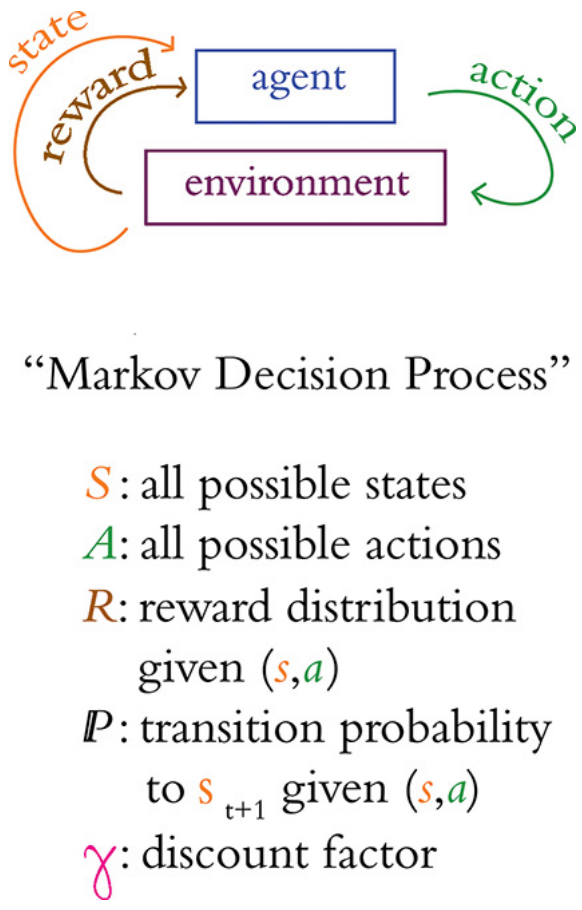

Figure 13.3 The reinforcement learning loop (top; a rehashed version of Figure 4.3, provided again here for convenience) can be considered a Markov decision process, which is defined by the five components S, A, R, $\mathbb{P}$, and γ (bottom).

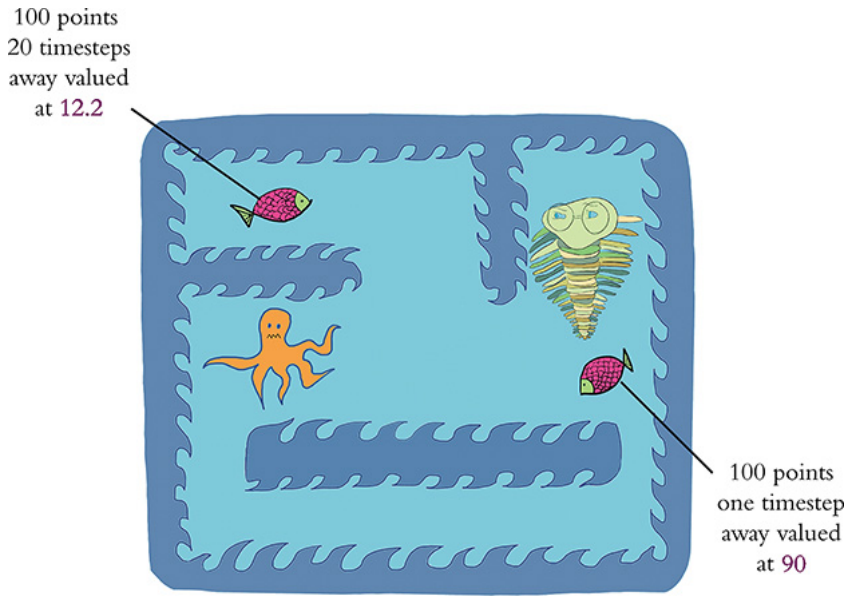

Figure 13.4 Based on the discount factor γ, in a Markov decision process more-distant reward is discounted relative to reward that's more immediately attainable. Using the Atari game Pac-Man to illustrate this concept (a green trilobite sitting in for Mr. Pac-Man himself), with γ = 0.9, cherries (or a fish!) only one timestep away are valued at 90 points, whereas cherries (a fish) 20 timesteps away are valued at 12.2 points. Like the ghosts in the Pac-Man game, the octopus here is roaming around and hoping to kill the poor trilobite. This is why immediately attainable rewards are more valuable than distant ones: There's a higher chance of being killed before reaching the fish that's farther away.

5 . Although this is true in reinforcement learning in general, the Cart-Pole game in particular is a relatively simple environment that is fully deterministic. In the Cart-Pole game, the exact same state-action pair (**s**, **a**) will in fact result in the same reward every time. For the purposes of illustrating the principles of reinforcement learning in general, we use examples in this section that imply the Cart-Pole game is less deterministic than it really is.

6 . For the sake of simplicity, let's ignore cart velocity and pole angular velocity for this example, because we can't infer these state aspects from this static image.

7 . As with all of the other artificial neural networks in this book, the ANNs within deep reinforcement learning agents are initialized with random starting parameters. This means that, prior to any learning (via, say, playing episodes of the Cart-Pole game), the agent has no awareness of even the simplest relationships between some state-action pair (**s**, **a**) and the next state $\mathbf{s}_{t+1}$. For example, although it may be intuitive and obvious to a human player of the Cart-Pole game that the action *left* should cause the cart to move leftward, *nothing* is "intuitive" or "obvious" to a randomly initialized neural net, and so all relationships must be learned through gameplay.

8 . The γ discount factor is analogous to the *discounted cash flow* calculations that are common in accounting: Prospective income a year from now is discounted relative to income expected today.

9 . Later in this chapter, we introduce concepts called value functions (V) and Q-value functions (Q). Both V and Q incorporate γ because it prevents them from becoming unbounded (and thus computationally impossible) in games with an infinite number of possible future timesteps.

10. $100 \times \gamma^t = 100 \times 0.9^1 = 90$

11. $100 \times \gamma^t = 100 \times 0.9^{20} = 12.16$

The Optimal Policy

The ultimate objective with an MDP is to find a function that enables an agent to take an appropriate action **a** (from the set of all possible actions **A**) when it encounters any particular state **s** from the set of all possible environmental states **S**. In other words, we'd like our agent to learn a function that enables it to *map* **S** to **A**. As shown in Figure 13.5, such a function is denoted by π and we call it the *policy function*.

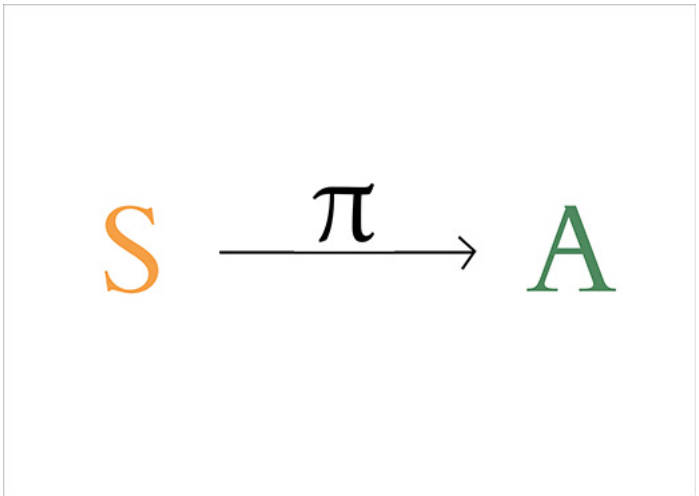

Figure 13.5 The policy function π enables an agent to map any state s (from the set of all possible states S) to an action a from the set of all possible actions A.

The high-level idea of the policy function π, using vernacular language, is this: Regardless of the particular circumstance the agent finds itself in, what is the *policy* it should follow that will enable it to maximize its reward? For a more concrete definition of this reward-maximization idea, you are welcome to pore over this:

$$J\left(\pi^*\right) = \max_\pi J(\pi) = \max_\pi \mathbb{E}\left[\sum_{t>0} \gamma^t r_t\right] \quad (13.1)$$

In this equation:

- $J(\pi)$ is called an *objective function*. This is a function that we can apply machine learning techniques to in order to maximize reward.[12]
- π represents *any* policy function that maps **S** to **A**.

- **π*** represents a particular, *optimal* policy (out of all the potential **π** policies) for mapping **S** to **A**. That is, **π*** is a function that—fed any state **s**—will return an action **a** that will lead to the agent attaining the *max*-imum possible *discounted future reward*.

- *Expected discounted future reward* is defined by $\mathbb{E}\left[\sum_{t>0}\gamma^t r_t\right]$ where $\mathbb{E}$ stands for *expectation* and $\sum_{t>0}\gamma^t r_t$ stands for the *discounted future reward*.

- To calculate the discounted future reward $\sum_{t>0}\gamma^t r_t$, over all future timesteps (i.e., $t > 0$), we do the following.

  - Multiply the reward that can be attained in any given future timestep ($\mathbf{r}_t$) by the discount factor of that timestep (γ^t).

  - Accumulate these individual discounted future rewards ($\gamma^t\mathbf{r}_t$) by summing them all up (using $\sum$).

12. The cost functions (a.k.a. loss functions) referred to throughout this book are examples of objective functions. Whereas cost functions return some cost value **C**, the objective function $J(\boldsymbol{\pi})$ returns some reward value **r**. With cost functions, our objective is to *minimize* cost, so we apply gradient *descent* to them (as depicted by the valley-descending trilobite back in Figure 8.2). With the function $J(\boldsymbol{\pi})$, in contrast, our objective is to *maximize* reward, and so we technically apply gradient *ascent* to it (conjuring up Figure 8.2 imagery, imagine a trilobite hiking to identify the peak of a mountain) even though the mathematics are the same as with gradient descent.

ESSENTIAL THEORY OF DEEP Q-LEARNING NETWORKS

In the preceding section, we defined reinforcement learning as a Markov decision process. At the end of the section, we indicated that as part of an MDP, we'd like our agent—when it encounters any given state **s** at any given timestep *t*—to follow some optimal policy **π*** that will enable it to select an action **a** that maximizes the discounted future reward it can obtain. The issue is that—even with a rather simple reinforcement learning problem like the Cart-Pole game—it is computationally intractable (or, at least, extremely computationally inefficient) to definitively calculate the maximum cumulative discounted future reward, $max\left(\Sigma_{t>0}\gamma^t r_t\right)$. Because of all the possible future states **S** and all the possible actions **A** that could be taken in those future states, there are *way* too many possible future outcomes to take into consideration. Thus, as a computational shortcut, we'll describe the *Q-learning* approach for *estimating* what the optimal action **a** in a given situation might be.

Value Functions

The story of Q-learning is most easily described by beginning with an explanation of *value functions*. The value function is defined by $V^{\pi}(\mathbf{s})$. It provides us with an indication of how *valuable* a given state **s**

is if our agent follows its policy π from that state onward.

As a simple example, consider yet again the state $\mathbf{s}$ captured in Figure 13.1.[13] Assuming our agent already has some reasonably sensible policy π for balancing the pole, then the cumulative discounted future reward that we'd expect it to obtain in this state is probably fairly large because the pole is near vertical. The value $V^{\pi}(\mathbf{s})$, then, of this particular state $\mathbf{s}$ is high.

13. As we did earlier in this chapter, let's consider cart position and pole position only, because we can't speculate on cart velocity or pole angular velocity from this still image.

On the other hand, if we imagine a state $\mathbf{s}_h$ where the pole angle is approaching horizontal, the value of it—$V^{\pi}(\mathbf{s}_h)$—is lower, because our agent has already lost control of the pole and so the episode is likely to terminate within the next few timesteps.

Q-Value Functions

The *Q-value function*[14] builds on the value function by taking into account not only state: It considers the utility of a particular action when that action is paired with a given state—that is, it rehashes our old friend, the state-action pair symbolized by $(\mathbf{s}, \mathbf{a})$. Thus, where the value function is defined by $V^{\pi}(\mathbf{s})$, the Q-value function is defined by $Q^{\pi}(\mathbf{s}, \mathbf{a})$.

14. The "Q" in Q-value stands for *quality* but you seldom hear practitioners calling these "quality-value functions."

Let's return once more to Figure 13.1. Pairing the action *left* (let's call this $\mathbf{a}_L$) with this state $\mathbf{s}$ and then following a pole-balancing policy π from there should generally correspond to a high cumulative discounted future reward. Therefore, the Q-value of this state-action pair $(\mathbf{s}, \mathbf{a}_L)$ is high.

In comparison, let's consider pairing the action *right* (we can call it $\mathbf{a}_R$) with the state $\mathbf{s}$ from Figure 13.1 and then following a pole-balancing policy π from there. Although this might not turn out to be an egregious error, the cumulative discounted future reward would nevertheless probably be somewhat lower relative to taking the *left* action. In this state $\mathbf{s}$, the *left* action should generally cause the pole to become more vertically oriented (enabling the pole to be better controlled and better balanced), whereas the rightward action should generally cause it to become somewhat more horizontally oriented—thus, *less* controlled, and the episode somewhat more likely to end early. All in all, we would expect the Q-value of $(\mathbf{s}, \mathbf{a}_L)$ to be higher than the Q-value of $(\mathbf{s}, \mathbf{a}_R)$.

Estimating an Optimal Q-Value

When our agent confronts some state $\mathbf{s}$, we would then like it to be able to calculate the *optimal Q-value*, denoted as $Q^*(\mathbf{s}, \mathbf{a})$. We could consider all possible actions, and the action with the highest Q-value—the highest cumulative discounted future reward—would be the best choice.

In the same way that it is computationally intractable to definitively calculate the optimal policy **π***(Equation 13.1) even with relatively simple reinforcement learning problems, so too is it typically computationally intractable to definitively calculate an optimal Q-value, *Q**(**s**, **a**). With the approach of deep Q-learning (as introduced in Chapter 4; see Figure 4.5), however, we can leverage an artificial neural network to *estimate* what the optimal Q-value might be. These deep Q-learning networks (DQNs for short) rely on this equation:

$$Q^*(s, a) \approx Q(s, a; \theta) \quad (13.2)$$

In this equation:

- The optimal Q-value (*Q**(**s**, **a**)) is being *approximated*.
- The Q-value approximation function incorporates neural network model parameters (denoted by the Greek letter theta, **θ**) in addition to its usual state **s** and action **a** inputs. These parameters are the usual artificial neuron weights and biases that we have become familiar with since Chapter 6.

In the context of the Cart-Pole game, a DQN agent armed with Equation 13.2 can, upon encountering a particular state **s**, calculate whether pairing an action **a** (*left* or *right*) with this state corresponds to a higher predicted cumulative discounted future reward. If, say, *left* is predicted to be associated with a higher cumulative discounted future reward, then this is the action that should be taken. In the next section, we'll code up a DQN agent that incorporates a Keras-built dense neural net to illustrate hands-on how this is done.

For a thorough introduction to the theory of reinforcement learning, including deep Q-learning networks, we recommend the recent edition of Richard Sutton (Figure 13.6) and Andrew Barto's *Reinforcement Learning: An Introduction*,[15] which is available free of charge at `bit.ly/SuttonBarto`.

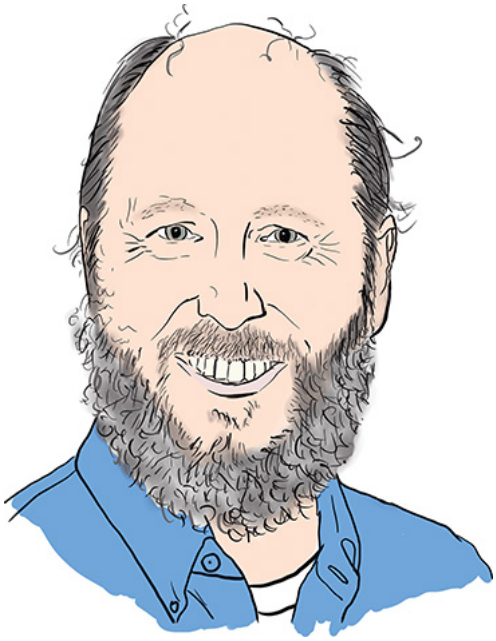

Figure 13.6 The biggest star in the field of reinforcement learning, Richard Sutton has long been a computer science professor at the University of Alberta. He is more recently also a distinguished research scientist at Google DeepMind.

15. Sutton, R., & Barto, A. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). Cambridge, MA: MIT Press.

DEFINING A DQN AGENT

Our code for defining a DQN agent that learns how to act in an environment—in this particular case, it happens to be the Cart-Pole game from the OpenAI Gym library of environments—is provided within our *Cartpole DQN* Jupyter notebook.[16] Its dependencies are as follows:

16. Our DQN agent is based directly on Keon Kim's, which is available at his GitHub repository at `bit.ly/keonDQN`.

Click here to view code image

```
import random
import gym
import numpy as np
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
import os
```

The most significant new addition to the list is `gym`, the Open AI Gym itself. As usual, we discuss each dependency in more detail as we apply it.

The hyperparameters that we set at the top of the notebook are provided in Example 13.1.

Example 13.1 Cart-Pole DQN hyperparameters

Click here to view code image

```
env = gym.make('CartPole-v0')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n
batch_size = 32
n_episodes = 1000
output_dir = 'model_output/cartpole/'
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
```

Let's look at this code line by line:

- We use the Open AI Gym `make()` method to specify the particular environment that we'd like our agent to interact with. The environment we choose is version zero (`v0`) of the Cart-Pole game, and we assign it to the variable `env`. On your own time, you're welcome to select an alternative Open AI Gym environment, such as one of those presented in Figure 4.13.

- From the environment, we extract two parameters:

  1. `state_size`: the number of types of state information, which for the Cart-Pole game is 4 (recall that these are cart position, cart velocity, pole angle, and pole angular velocity).

  2. `action_size`: the number of possible actions, which for Cart-Pole is 2 (*left* and *right*).

- We set our mini-batch size for training our neural net to `32`.

- We set the number of episodes (rounds of the game) to `1000`. As you'll soon see, this is about the right number of episodes it will take for our agent to excel regularly at the Cart-Pole game. For more-complex environments, you'd likely need to increase this hyperparameter so that the agent has more rounds of gameplay to learn in.

- We define a unique directory name (`'model_output/cartpole/'`) into which we'll output our neural network's parameters at regular intervals. If the directory doesn't yet exist, we use `os.makedirs()` to make it.

The rather large chunk of code for creating a DQN agent Python class—called `DQNAgent`—is provided in Example 13.2.

Example 13.2 A deep Q-learning agent

Click here to view code image

```
class DQNAgent:
        def __init__(self, state_size, action_size):
            self.state_size = state_size
            self.action_size = action_size
            self.memory = deque(maxlen=2000)
            self.gamma = 0.95
            self.epsilon = 1.0
            self.epsilon_decay = 0.995
            self.epsilon_min = 0.01
            self.learning_rate = 0.001
            self.model = self._build_model()

        def _build_model(self):
            model = Sequential()
            model.add(Dense(32, activation='relu',
                            input_dim=self.state_size))
            model.add(Dense(32, activation='relu'))
            model.add(Dense(self.action_size, activation='linear'))
            model.compile(loss='mse',
                          optimizer=Adam(lr=self.learning_rate))
            return model

        def remember(self, state, action, reward, next_state, done):
            self.memory.append((state, action,
                                reward, next_state, done))

        def train(self, batch_size):
            minibatch = random.sample(self.memory, batch_size)
            for state, action, reward, next_state, done in minibatch:
                target = reward # if done
                if not done:

                    target = (reward +
                              self.gamma *
                              np.amax(self.model.predict(next_state)[0]))
                target_f = self.model.predict(state)
                target_f[0][action] = target
                self.model.fit(state, target_f, epochs=1, verbose=0)
            if self.epsilon >  self.epsilon_min:
                self.epsilon *= self.epsilon_decay

        def act(self, state):
            if np.random.rand() <= self.epsilon:
                return random.randrange(self.action_size)
            act_values = self.model.predict(state)
            return np.argmax(act_values[0])

        def save(self, name):
            self.model.save_weights(name)

        def load(self, name):
            self.model.load_weights(name)
```

Initialization Parameters

We begin Example 13.2 by initializing the class with a number of parameters:

- `state_size` and `action_size` are environment-specific, but in the case of the Cart-Pole game are 4 and 2, respectively, as mentioned earlier.

- `memory` is for storing *memories* that can subsequently be *replayed* in order to train our DQN's neural net. The memories are stored as elements of a data structure called a *deque* (pronounced "deck"), which is the same as a list except that—because we specified `maxlen=2000`—it only retains the 2,000 most recent memories. That is, whenever we attempt to append a 2,001st element onto the deque, its first element is removed, always leaving us with a list that contains no more than 2,000 elements.

- `gamma` is the discount factor (a.k.a. decay rate) γ that we introduced earlier in this chapter (see Figure 13.4). This agent hyperparameter discounts prospective rewards in future timesteps. Effective γ values typically approach 1 (for example, 0.9, 0.95, 0.98, and 0.99). The closer to 1, the less we're discounting future reward.[17] Tuning the hyperparameters of reinforcement learning models such as γ can be a fiddly process; near the end of this chapter, we discuss a tool called SLM Lab for carrying it out effectively.

- `epsilon`—symbolized by the Greek letter $\in$—is another reinforcement learning hyperparameter called *exploration rate*. It represents the proportion of our agent's actions that are random (enabling it to *explore* the impact of such actions on the next state $\mathbf{s}_{t+1}$ and the reward $\mathbf{r}$ returned by the environment) relative to how often we allow its actions to *exploit* the existing "knowledge" its neural net has accumulated through gameplay. Prior to having played any episodes, agents have no gameplay experience to exploit, so it is the most common practice to start it off exploring 100 percent of the time; this is why we set `epsilon = 1.0`.

- As the agent gains gameplay experience, we very slowly *decay* its exploration rate so that it can gradually exploit the information it has learned (hopefully enabling it to attain more reward, as illustrated in Figure 13.7). That is, at the end of each episode the agent plays, we multiply its $\in$ by `epsilon_decay`. Common options for this hyperparameter are 0.990, 0.995, and 0.999.[18]

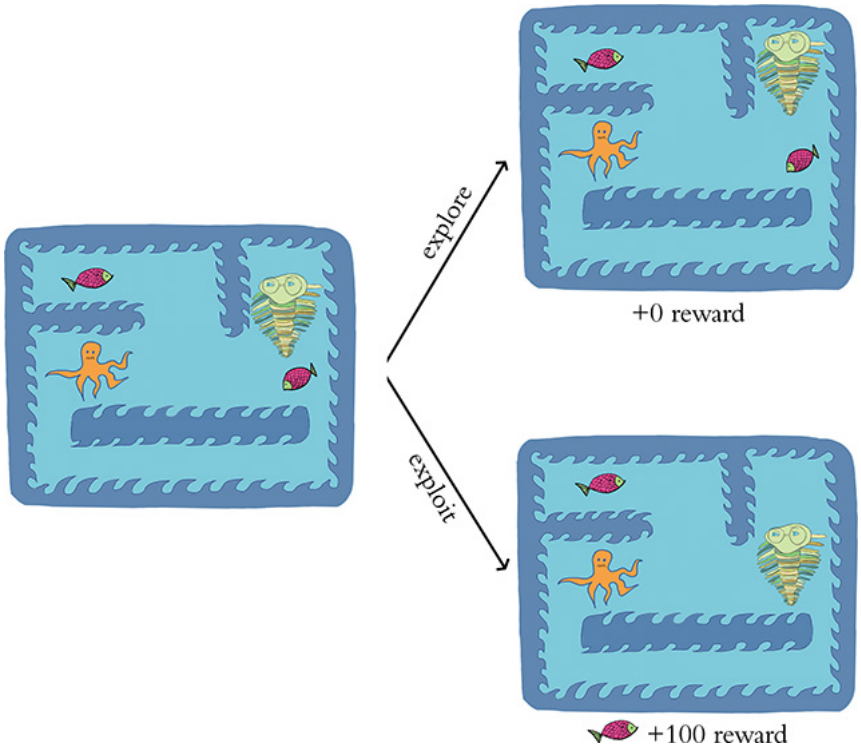

Figure 13.7 As in Figure 13.4, here we use the Pac-Man environment (with a green trilobite representing a DQN agent in place of the Mr. Pac-Man character) to illustrate a reinforcement learning concept. In this case, the concept is exploratory versus exploitative actions. The higher the hyperparameter ∈ (epsilon) in a given episode, the more likely the agent is to be in its exploratory mode, in which it takes purely random actions: By chance, an agent in this mode might navigate in the opposite direction of a fish that would have provided an immediate reward of 100 points. The alternative to the exploratory mode is the exploitative mode. Assuming the DQN agent's neural net parameters have already benefited from some previous gameplay experience, in its exploitative mode the agent's policy should be to acquire reward that is immediately available to it.

- `epsilon_min` is a floor (a minimum) on how low the exploration rate ∈ can decay to. This hyperparameter is typically set to a near-zero value such as 0.001, 0.01, or 0.02. We set it equal to `0.01`, meaning that after ∈ has decayed to 0.01 (as it will in our case by the 911th episode), our agent will explore on only 1 percent of the actions it takes—exploiting its gameplay experience the other 99 percent of the time.[19]

- `learning_rate` is the same stochastic gradient descent hyperparameter that we covered in Chapter 8.

- Finally, `_build_model()`—by the inclusion of its leading underscore—is being suggested as a *private* method. This means that this method is recommended for use "internally" only—that is, solely by instances of the class `DQNAgent`.

17. Indeed, if you were to set $\gamma = 1$ (which we don't recommend) you wouldn't be discounting future reward at all.

18. Analogous to setting $\gamma = 1$, setting epsilon_decay = 1 would mean ϵ would not be decayed at all—that is, exploring at a continuous rate. This would be an unusual choice for this hyperparameter.

19. If at this stage this exploration rate concept is somewhat unclear, it should become clearer as we examine our agent's episode-by-episode results later on.

Building the Agent's Neural Network Model

The `_build_model()` method of Example 13.2 is dedicated to constructing and compiling a Keras-specified neural network that maps an environment's state **s** to the agent's Q-value for each available action **a**. Once trained via gameplay, the agent will then be able to use the predicted Q-values to select the particular action it should take, given a particular environmental state it encounters. Within the method, there is nothing you haven't seen before in this book:

- We specify a sequential model.
- We add to the model the following layers of neurons.
  - The first hidden layer is dense, consisting of 32 ReLU neurons. Using the `input_dim` argument, we specify the shape of the network's input layer, which is the dimensionality of the environment's state information **s**. In the case of the Cart-Pole environment, this value is an array of length 4, with one element each for cart position, cart velocity, pole angle, and pole angular velocity.[20]
  - The second hidden layer is also dense, with 32 ReLU neurons. As mentioned earlier, we'll explore hyperparameter selection—including how we home in on a particular model architecture—by discussing the SLM Lab tool later on in this chapter.
  - The output layer has dimensionality corresponding to the number of possible actions.[21] In the case of the Cart-Pole game, this is an array of length 2, with one element for *left* and the other for *right*. As with a regression model (see Example 9.8), with DQNs the **z** values are output directly from the neural net instead of being converted into a probability between 0 and 1. To do this, we specify the `linear` activation function instead of the sigmoid or softmax functions that have otherwise dominated this book.
- As indicated when we compiled our regression model (Example 9.9), mean squared error is an appropriate choice of cost function when we use linear activation in the output layer, so we set the `compile()` method's `loss` argument to `mse`. We return to our routine optimizer choice, `Adam`.

20. In environments other than Cart-Pole, the state information might be much more complex. For example, with an Atari video game environment like Pac-Man, state **s** would consist of pixels on a screen, which would be a two- or three-dimensional input (for monochromatic or full-color,

respectively). In a case such as this, a better choice of first hidden layer would be a convolutional layer such as Conv2D (see Chapter 10).

21. Any previous models in this book with only two outcomes (as in Chapters 11 and 12) used a single sigmoid neuron. Here, we specify separate neurons for each of the outcomes, because we would like our code to generalize beyond the Cart-Pole game. While Cart-Pole has only two actions, many environments have more than two.

Remembering Gameplay

At any given timestep *t*—that is, during any given iteration of the reinforcement learning loop (refer back to Figure 13.3)—the DQN agent's `remember()` method is run in order to append a memory to the end of its `memory` deque. Each memory in this deque consists of five pieces of information about timestep *t*:

1. The `state` $\mathbf{s}_t$ that the agent encountered
2. The `action` $\mathbf{a}_t$ that the agent took
3. The `reward` $\mathbf{r}_t$ that the environment returned to the agent
4. The `next_state` $\mathbf{s}_{t+1}$ that the environment also returned to the agent
5. A Boolean flag `done` that is `true` if timestep *t* was the final iteration of the episode, and `false` otherwise

Training via Memory Replay

The DQN agent's neural net model is trained by *replaying memories* of gameplay, as shown within the `train()` method of Example 13.2. The process begins by randomly sampling a `minibatch` of 32 (as per the agent's `batch_size` parameter) memories from the `memory` deque (which holds up to 2,000 memories). Sampling a small subset of memories from a much larger set of the agent's experiences makes model-training more efficient: If we were instead to use, say, the 32 most recent memories to train our model, many of the states across those memories would be very similar. To illustrate this point, consider a timestep *t* where the cart is at some particular location and the pole is near vertical. The adjacent timesteps (e.g., $t - 1$, $t + 1$, $t + 2$) are also likely to be at nearly the same location with the pole in a near-vertical orientation. By sampling from across a broad range of memories instead of temporally proximal ones, the model will be provided with a richer cornucopia of experiences to learn from during each round of training.

For each of the 32 sampled memories, we carry out a round of model training as follows: If `done` is `True`—that is, if the memory was of the final timestep of an episode—then we know definitively that

the highest possible reward that could be attained from this timestep is equal to the `reward` $\mathbf{r}_t$. Thus, we can just set our `target` reward equal to `reward`.

Otherwise (i.e., if `done` is `False`) then we try to estimate what the `target` reward—the maximum discounted future reward—might be. We perform this estimation by starting with the known `reward` $\mathbf{r}_t$ and adding to it the discounted[22] maximum future Q-value. Possible future Q-values are estimated by passing the next (i.e., *future*) state $\mathbf{s}_{t+1}$ into the model's `predict()` method. Doing this in the context of the Cart-Pole game returns two outputs: one output for the action *left* and the other for the action *right*. Whichever of these two outputs is higher (as determined by the NumPy `amax` function) is the maximum predicted future Q-value.

22. That is, multiplied by `gamma`, the discount factor γ.

Whether `target` is known definitively (because the timestep was the final one in an episode) or it's estimated using the maximum future Q-value calculation, we continue onward within the `train()` method's `for` loop:

- We run the `predict()` method again, passing in the *current* state $\mathbf{s}_t$. As before, in the context of the Cart-Pole game this returns two outputs: one for the *left* action and one for the *right*. We store these two outputs in the variable `target_f`.

- Whichever `action` $\mathbf{a}_t$ the agent actually took in this memory, we use `target_f[0][action] = target` to replace that `target_f` output with the `target` reward.[23]

- We train our model by calling the `fit()` method.

  - The model input is the current `state` $\mathbf{s}_t$ and its output is `target_f`, which incorporates our approximation of the maximum future discounted reward. By tuning the model's parameters (represented by $\boldsymbol{\theta}$ in Equation 13.2), we thus improve its capacity to accurately predict the action that is more likely to be associated with maximizing future reward in any given state.

  - In many reinforcement learning problems, `epochs` can be set to `1`. Instead of recycling an existing training dataset multiple times, we can cheaply engage in more episodes of the Cart-Pole game (for example) to generate as many fresh training data as we fancy.

  - We set `verbose=0` because we don't need any model-fitting outputs at this stage to monitor the progress of model training. As we demonstrate shortly, we'll instead monitor agent performance on an episode-by-episode basis.

23. We do this because we can only train the Q-value estimate based on actions that were actually taken by the agent: We estimated `target` based on `next_state` $\mathbf{s}_{t+1}$ and we only know what $\mathbf{s}_{t+1}$ was for the `action` $\mathbf{a}_t$ that was actually taken by the agent at timestep *t*. We don't know what next state $\mathbf{s}_{t+1}$ the environment might have returned had the agent taken a different action than it actually took.

Selecting an Action to Take

To select a particular action $\mathbf{a}_t$ to take at a given timestep *t*, we use the agent's `act()` method. Within this method, the NumPy `rand` function is used to sample a random value between 0 and 1 that we'll call υ. In conjunction with our agent's `epsilon`, `epsilon_decay`, and `epsilon_min` hyperparameters, this υ value will determine for us whether the agent takes an exploratory action or an exploitative one:[24]

- If the random value *υ* is less than or equal to the exploration rate $\in$, then a random exploratory action is selected using the `randrange` function. In early episodes, when $\in$ is high, most of the actions will be exploratory. In later episodes, as $\in$ decays further and further (according to the `epsilon_decay` hyperparameter), the agent will take fewer and fewer exploratory actions.

- Otherwise—that is, if the random value υ is greater than $\in$—the agent selects an action that exploits the "knowledge" the model has learned via memory replay. To exploit this knowledge, the `state` $\mathbf{s}_t$ is passed in to the model's `predict()` method, which returns an activation output[25] for each of the possible actions the agent could theoretically take. We use the NumPy `argmax` function to select the action $\mathbf{a}_t$ associated with the largest activation output.

24. We introduced the exploratory and exploitative modes of action when discussing the initialization parameters for our DQNAgent class earlier, and they're illustrated playfully in Figure 13.7.

25. Recall that the activation is linear, and thus the output is *not* a probability; instead, it is the discounted future reward for that action.

Saving and Loading Model Parameters

Finally, the `save()` and `load()` methods are one-liners that enable us to save and load the parameters of the model. Particularly with respect to complex environments, agent performance can be flaky: For long stretches, the agent may perform very well in a given environment, and then later appear to lose its capabilities entirely. Because of this flakiness, it's wise to save our model parameters at regular intervals. Then, if the agent's performance drops off in later episodes, the higher-performing parameters from some earlier episode can be loaded back up.

INTERACTING WITH AN OPENAI GYM ENVIRONMENT

Having created our DQN agent class, we can initialize an instance of the class—which we name `agent`—with this line of code:

Click here to view code image

```
agent = DQNAgent(state_size, action_size)
```

The code in Example 13.3 enables our `agent` to interact with an OpenAI Gym environment, which in our particular case is the Cart-Pole game.

Example 13.3 DQN agent interacting with an OpenAI Gym environment

Click here to view code image

```
    for e in range(n_episodes):
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        done = False
        time = 0
        while not done:
    #         env.render()
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            reward = reward if not done else -10
            next_state = np.reshape(next_state, [1, state_size])
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            if done:
               print("episode: {}/{}, score: {}, e: {:.2}"
                     .format(e, n_episodes-1, time, agent.epsilon))
            time += 1
        if len(agent.memory) > batch_size:
            agent.train(batch_size)
        if e % 50 == 0:
            agent.save(output_dir + "weights_"
                       + '{:04d}'.format(e) + ".hdf5")
```

Recalling that we had set the hyperparameter `n_episodes` to `1000`, Example 13.3 consists of a big `for` loop that allows our agent to engage in these 1,000 rounds of game-play. Each episode of gameplay is counted by the variable `e` and involves:

- We use `env.reset()` to begin the episode with a random `state` $\mathbf{s}_t$. For the purposes of passing `state` into our Keras neural network in the orientation the model is expecting, we use `reshape` to convert it from a column into a row.[26]

- Nested within our thousand-episode loop is a `while` loop that iterates over the timesteps of a given episode. Until the episode ends (i.e., until `done` equals `True`), in each timestep *t* (represented by the variable `time`), we do the following.

    - The `env.render()` line is commented out because if you are running this code via a Jupyter notebook within a Docker container, this line will cause an error. If, however, you happen to be running the code via some other means (e.g., in a Jupyter notebook *without* using Docker) then you can try uncommenting this line. If an error isn't thrown, then a pop-up window should appear that *renders* the environment graphically. This enables you to watch your DQN agent as it plays

the Cart-Pole game in real time, episode by episode. It's fun to watch, but it's by no means essential: It certainly has no impact on how the agent learns!

- We pass the `state` $\mathbf{s}_t$ into the agent's `act()` method, and this returns the agent's `action` $\mathbf{a}_t$, which is either `0` (representing *left*) or `1` (*right*).

- The action $\mathbf{a}_t$ is provided to the environment's `step()` method, which returns the `next_state` $\mathbf{s}_{t+1}$, the current `reward` $\mathbf{r}_t$, and an update to the Boolean flag `done`.

- If the episode is done (i.e., `done` equals `true`), then we set `reward` to a negative value (`-10`). This provides a strong disincentive to the agent to end an episode early by losing control of balancing the pole or navigating off the screen. If the episode is not done (i.e., `done` is `False`), then `reward` is `+1` for each additional timestep of gameplay.

- In the same way that we needed to reorient `state` to be a row at the start of the episode, we use `reshape` to reorient `next_state` to a row here.

- We use our agent's `remember()` method to save all the aspects of this timestep (the state $\mathbf{s}_t$, the action $\mathbf{a}_t$ that was taken, the reward $\mathbf{r}_t$, the next state $\mathbf{s}_{t+1}$, and the flag `done`) to memory.

- We set `state` equal to `next_state` in preparation for the next iteration of the loop, which will be timestep $t + 1$.

- If the episode ends, then we print summary metrics on the episode (see Figures 13.8 and 13.9 for example outputs).

Figure 13.8 The performance of our DQN agent during its first 10 episodes playing the Cart-Pole game. Its scores are low (keeping the game going for between 10 and 42 timesteps), and its exploration rate $\in$ is high (starting at 100 percent and decaying to 96 percent by the 10th episode).

```
episode: 990/999, score: 199, e: 0.01
episode: 991/999, score: 199, e: 0.01
episode: 992/999, score: 199, e: 0.01
episode: 993/999, score: 199, e: 0.01
episode: 994/999, score: 199, e: 0.01
episode: 995/999, score: 199, e: 0.01
episode: 996/999, score: 199, e: 0.01
episode: 997/999, score: 199, e: 0.01
episode: 998/999, score: 199, e: 0.01
episode: 999/999, score: 199, e: 0.01
```

Figure 13.9 The performance of our DQN agent during its final 10 episodes playing the Cart-Pole game. It scores the maximum (199 timesteps) across all 10 episodes. The exploration rate $\in$ had already decayed to its minimum of 1 percent, so the agent is in exploitative mode for ~99 percent of its actions.

  - Add `1` to our timestep counter `time`.

- If the length of the agent's memory deque is larger than our batch size, then we use the agent's `train()` method to train its neural net parameters by replaying its memories of gameplay.[27]

- Every `50` episodes, we use the agent's `save()` method to store the neural net model's parameters.

26. We previously performed this transposition for the same reason back in Example 9.11.

27. You can optionally move this training step up so that it's inside the `while` loop. Each episode will take *a lot* longer because you'll be training the agent much more often, but your agent will tend to solve the Cart-Pole game in far fewer episodes.

As shown in Figure 13.8, during our agent's first 10 episodes of the Cart-Pole game, the scores were low. It didn't manage to keep the game going for more than 42 timesteps (i.e., a score of 41). During these initial episodes, the exploration rate $\in$ began at 100 percent. By the 10th episode, $\in$ had decayed to 96 percent, meaning that the agent was in exploitative mode (refer back to Figure 13.7) on about 4 percent of timesteps. At this early stage of training, however, most of these exploitative actions would probably have been effectively random anyway.

As shown in Figure 13.9, by the 991st episode our agent had mastered the Cart-Pole game. It attained a perfect score of 199 in all of the final 10 episodes by keeping the game going for 200 timesteps in each one. By the 911th episode,[28] the exploration rate $\in$ had reached its minimum of 1 percent so during all of these final episodes the agent is in exploitative mode in about 99 percent of timesteps. From the perfect performance in these final episodes, it's clear that these exploitative actions were guided by a neural net well trained by its gameplay experience from previous episodes.

28. Not shown here, but can be seen in our *Cartpole DQN* Jupyter notebook.

As mentioned earlier in this chapter, deep reinforcement learning agents often display finicky behavior. When you train your DQN agent to play the Cart-Pole game, you might find that it performs very well during some later episodes (attaining many consecutive 200-timestep episodes around, say, the 850th or 900th episode) but then it performs poorly around the final (1,000th) episode. If this ends up being the case, you can use the `load()` method to restore model parameters from an earlier, higher-performing phase.

HYPERPARAMETER OPTIMIZATION WITH SLM LAB

At a number of points in this chapter, in one breath we'd introduce a hyperparameter and then in the next breath we'd indicate that we'd later introduce a tool called SLM Lab for tuning that hyperparameter.[29] Well, that moment has arrived!

29. "SLM" is an abbreviation of *strange loop machine*, with the *strange loop* concept being related to ideas about the experience of human consciousness. See Hofstadter, R. (1979). *Gödel, Escher, Bach*. New York: Basic Books.

SLM Lab is a deep reinforcement learning framework developed by Wah Loon Keng and Laura Graesser, who are California-based software engineers (at the mobile-gaming firm MZ and within the Google Brain team, respectively). The framework is available at github.com/kengz/SLM-Lab (http://github.com/kengz/SLM-Lab) and has a broad range of implementations and functionality related to deep reinforcement learning:

- It enables the use of many types of deep reinforcement learning agents, including DQN and others (forthcoming in this chapter).
- It provides modular agent components, allowing you to dream up your own novel categories of deep RL agents.
- You can straightforwardly drop agents into environments from a number of different environment libraries, such as OpenAI Gym and Unity (see Chapter 4).
- Agents can be trained in multiple environments simultaneously. For example, a single DQN agent can at the same time solve the OpenAI Gym Cart-Pole game and the Unity ball-balancing game Ball2D.
- You can benchmark your agent's performance in a given environment against others' efforts.

Critically, for our purposes, the SLM Lab also provides a painless way to experiment with various agent hyperparameters to assess their impact on an agent's performance in a given environment. Consider, for

example, the *experiment graph* shown in Figure 13.10. In this particular experiment, a DQN agent was trained to play the Cart-Pole game during a number of distinct *trials*. Each trial is an instance of an agent with particular, distinct hyperparameters trained for many episodes. Some of the hyperparameters varied between trials were as follows.

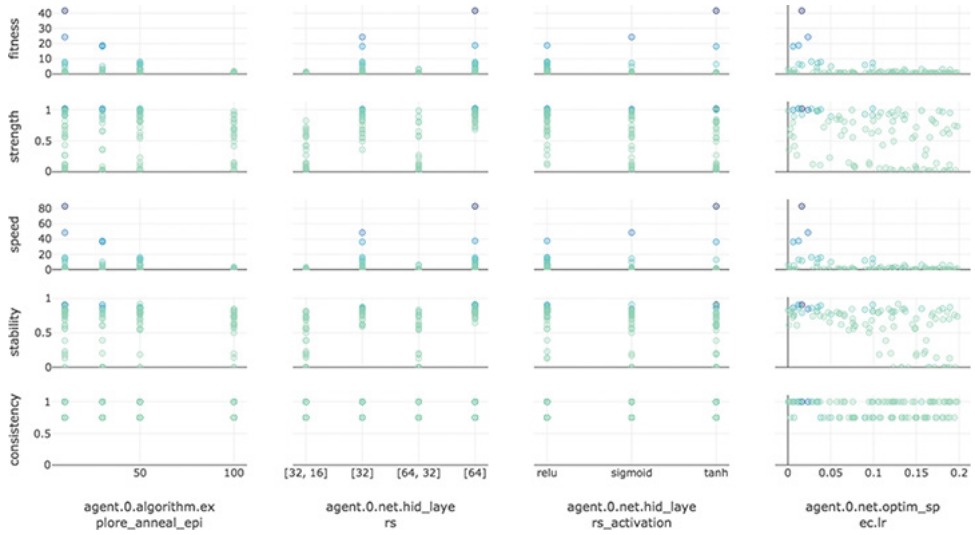

Figure 13.10 An experiment run with SLM Lab, investigating the impact of various hyperparameters (e.g., hidden-layer architecture, activation function, learning rate) on the performance of a DQN agent within the Cart-Pole environment

- Dense net model architecture
  - `[32]`: a single hidden layer, with 32 neurons
  - `[64]`: also a single hidden layer, this time with 64 neurons
  - `[32, 16]`: two hidden layers; the first with 32 neurons and the second with 16
  - `[64, 32]`: also with two hidden layers, this time with 64 neurons in the first hidden layer and 32 in the second
- Activation function across all hidden layers
  - Sigmoid
  - Tanh
  - ReLU
- Optimizer learning rate (η), which ranged from zero up to 0.2
- Exploration rate ($\in$) *annealing*, which ranged from 0 to 100[30]

30. *Annealing* is an alternative to ∈ decay that serves the same purpose. With the `epsilon` and `epsilon_min` hyper-parameters set to fixed values (say, 1.0 and 0.01, respectively), variations in annealing will adjust `epsilon_decay` such that an ∈ of 0.01 will be reached by a specified episode. If, for example, annealing is set to 25 then ∈ will decay at a rate such that it lowers uniformly from 1.0 in the first episode to 0.01 after 25 episodes. If annealing is set to 50 then ∈ will decay at a rate such that it lowers uniformly from 1.0 in the first episode to 0.01 after 50 episodes.

SLM Lab provides a number of metrics for evaluating model performance (some of which can be seen along the vertical axis of Figure 13.10):

- *Strength*: This is a measure of the cumulative reward attained by the agent.
- *Speed*: This is how quickly (i.e., over how many episodes) the agent was able to reach its strength.
- *Stability*: After the agent solved how to perform well in the environment, this is a measure of how well it retained its solution over subsequent episodes.
- *Consistency*: This is a metric of how reproducible the performance of the agent was across trials that had identical hyperparameter settings.
- *Fitness*: An overall summary metric that takes into account the above four metrics simultaneously. Using the fitness metric in the experiment captured by Figure 13.10, it appears that the following hyperparameter settings are optimal for this DQN agent playing the Cart-Pole game:
  - A single-hidden-layer neural net architecture, with 64 neurons in that single layer outperforming the 32-neuron model.
  - The tanh activation function for the hidden layer neurons.
  - A low learning rate (η) of ~0.02.
  - Trials with an exploration rate (∈) that anneals over 10 episodes outperform trials that anneal over 50 or 100 episodes.

Details of running SLM Lab are beyond the scope of our book, but the library is well documented at `kengz.gitbooks.io/slm-lab`.[31]

31. At the time of this writing, SLM Lab installation is straightforward only on Unix-based systems, including macOS.

AGENTS BEYOND DQN

In the world of deep reinforcement learning, deep Q-learning networks like the one we built in this chapter are relatively simple. To their credit, not only are DQNs (comparatively) simple, but—relative

to many other deep RL agents—they also make efficient use of the training samples that are available to them. That said, DQN agents do have drawbacks. Most notable are:

1. If the possible number of state-action pairs is large in a given environment, then the Q-function can become extremely complicated, and so it becomes intractable to estimate the optimal Q-value, Q^*.

2. Even in situations where finding Q^* is computationally tractable, DQNs are not great at exploring relative to some other approaches, and so a DQN may not converge on Q^* anyway.

Thus, even though DQNs are sample efficient, they aren't applicable to solving all problems.

To wrap up this deep reinforcement learning chapter, let's briefly introduce the types of agents beyond DQNs. The main categories of deep RL agents, as shown in Figure 13.11, are:

- *Value optimization*: These include DQN agents and their derivatives (e.g., *double DQN*, *dueling QN*) as well as other types of agents that solve reinforcement learning problems by optimizing value functions (including Q-value functions).

Figure 13.11 The broad categories of deep reinforcement learning agents

- *Imitation learning*: The agents in this category (e.g., *behavioral cloning* and *conditional imitation learning* algorithms) are designed to mimic behaviors that are taught to them through demonstration, by—for example—showing them how to place dinner plates on a dish rack or how to pour water into a cup. Although imitation learning is a fascinating approach, its range of applications is relatively small and we don't discuss it further in this book.

- *Model optimization*: Agents in this category learn to predict future states based on (**s**, **a**) at a given timestep. An example of one such algorithm is Monte Carlo tree search (MCTS), which we introduced with respect to AlphaGo in Chapter 4.

- *Policy optimization*: Agents in this category learn policies *directly*, that is, they directly learn the policy function π shown in Figure 13.5. We'll cover these in further detail in the next section.

Policy Gradients and the REINFORCE Algorithm

Recall from Figure 13.5 that the purpose of a reinforcement learning agent is to learn some policy function π that maps the state space $\mathbf{S}$ to the action space $\mathbf{A}$. With DQNs, and indeed with any other value optimization agent, π is learned indirectly by estimating a value function such as the optimal Q-value, Q^*. With policy optimization agents, π is learned *directly* instead.

Policy gradient (PG) algorithms, which can perform gradient *ascent*[32] on π directly, are exemplified by a particularly well-known reinforcement learning algorithm called REINFORCE.[33] The advantage of PG algorithms like REINFORCE is that they are likely to converge on a fairly optimal solution,[34] so they're more widely applicable than value optimization algorithms like DQN. The trade-off is that PGs have *low consistency*. That is, they have higher variance in their performance relative to value optimization approaches like DQN, and so PGs tend to require a larger number of training samples.

32. Because PG algorithms *maximize* reward (instead of, say, minimizing cost), they perform gradient *ascent* and not gradient descent. For more on this, see Footnote 12 in this chapter.

33. Williams, R. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*, 229–56.

34. PG agents tend to converge on at least an optimal local solution, although some particular PG methods have been demonstrated to identify the optimal *global* solution to a problem. See Fazel, K., et al. (2018). Global convergence of policy gradient methods for the linear quadratic regulator. *arXiv: 1801.05039*.

The Actor-Critic Algorithm

As suggested by Figure 13.11, the *actor-critic* algorithm is an RL agent that combines the value optimization and policy optimization approaches. More specifically, as depicted in Figure 13.12, the actor-critic combines the Q-learning and PG algorithms. At a high level, the resulting algorithm involves a loop that alternates between:

- *Actor*: a PG algorithm that decides on an action to take.

- *Critic*: a Q-learning algorithm that critiques the action that the actor selected, providing feedback on how to adjust. It can take advantage of efficiency tricks in Q-learning, such as memory replay.

Figure 13.12 The actor-critic algorithm combines the policy gradient approach to reinforcement learning (playing the role of actor) with the Q-learning approach (playing the role of critic).

In a broad sense, the actor-critic algorithm is reminiscent of the generative adversarial networks of Chapter 12. GANs have a generator network in a loop with a discriminator network, with the former creating fake images that are evaluated by the latter. The actor-critic algorithm has an actor in a loop with a critic, with the former taking actions that are evaluated by the latter.

The advantage of the actor-critic algorithm is that it can solve a broader range of problems than DQN, while it has lower variance in performance relative to REINFORCE. That said, because of the presence of the PG algorithm within it, the actor-critic is still somewhat sample inefficient.

While implementing REINFORCE and the actor-critic algorithm are beyond the scope of this book, you can use SLM Lab to apply them yourself, as well as to examine their underlying code.

SUMMARY

In this chapter, we covered the essential theory of reinforcement learning, including Markov decision processes. We leveraged that information to build a deep Q-learning agent that solved the Cart-Pole environment. To wrap up, we introduced deep RL algorithms beyond DQN such as REINFORCE and actor-critic. We also described SLM Lab—a deep RL framework with existing algorithm implementations as well as tools for optimizing agent hyperparameters.

This chapter brings an end to Part III of this book, which provided hands-on applications of machine vision (Chapter 10), natural language processing (Chapter 11), art-generating models (Chapter 12), and sequential decision-making agents. In Part IV, the final part of the book, we will provide you with loose guidance on adapting these applications to your own projects and inclinations.

KEY CONCEPTS

Listed here are the key concepts from across this book. The final concept—covered in the current chapter—is highlighted in purple.

- parameters:
  - weight **w**
  - bias **b**
- activation **a**
- artificial neurons:
  - sigmoid
  - tanh
  - ReLU
  - linear
- input layer
- hidden layer
- output layer
- layer types:
  - dense (fully connected)
  - softmax
  - convolutional
  - de-convolutional
  - max-pooling
  - upsampling
  - flatten
  - embedding

    - RNN
    - (bidirectional-)LSTM
    - concatenate
- cost (loss) functions:
    - quadratic (mean squared error)
    - cross-entropy
- forward propagation
- backpropagation
- unstable (especially vanishing) gradients
- Glorot weight initialization
- batch normalization
- dropout
- optimizers:
    - stochastic gradient descent
    - Adam
- optimizer hyperparameters:
    - learning rate η
    - batch size
- word2vec
- GAN components:
    - discriminator network
    - generator network

- adversarial network

- deep Q-learning

PREV
12. Generative Adversarial Networks

NEXT
IV: You and AI