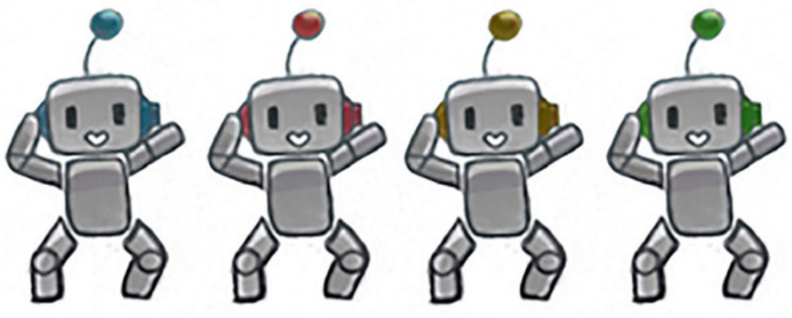
PREV
Chapter 7. A peek into autoencoders

NEXT
Chapter 9. Convolutional neural networks

Chapter 8. Reinforcement learning



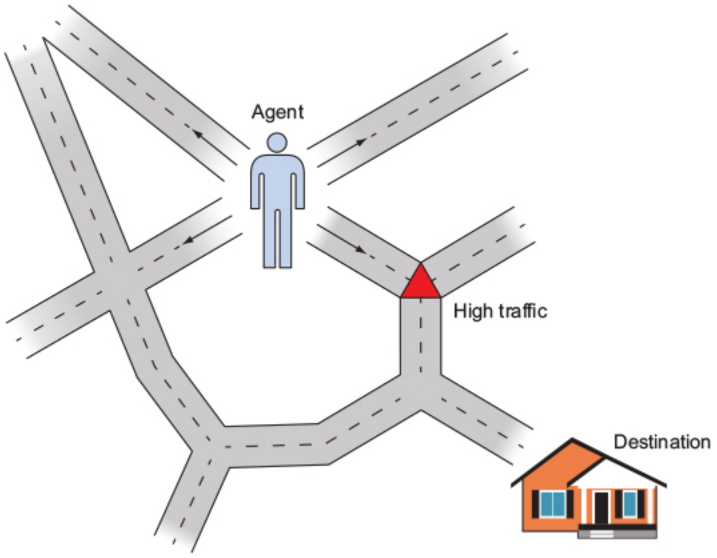
This chapter covers

- Defining reinforcement learning
- Implementing reinforcement learning

Humans learn from past experiences (or, at least they *should*). You didn't get so charming by accident. Years of positive compliments as well as negative criticism have all helped shape who you are today. This chapter is about designing a machine-learning system driven by criticisms and rewards.

You learn what makes people happy, for example, by interacting with friends, family, or even strangers, and you figure out how to ride a bike by trying out various muscle movements until riding just clicks. When you perform actions, you're sometimes rewarded immediately. For example, finding a good restaurant nearby might yield instant gratification. Other times, the reward doesn't appear right away, such as traveling a long distance to find an exceptional place to eat. Reinforcement learning is about making the right actions, given any state—such as in figure 8.1, which shows a person making decisions to arrive at their destination.

Figure 8.1. A person navigating to reach a destination in the midst of traffic and unexpected situations is a problem setup for reinforcement learning.



Moreover, suppose on your drive from home to work, you always choose the same route. But one day your curiosity takes over, and you decide to try a different path in hopes of a shorter commute. This dilemma of trying out new routes or sticking to the best-known route is an example of *exploration versus exploitation*.

Note

Why is the trade-off between trying new things and sticking with old ones called exploration versus exploitation? Exploration makes sense, but you can think of exploitation as exploiting your knowledge of the status quo by sticking with what you know.

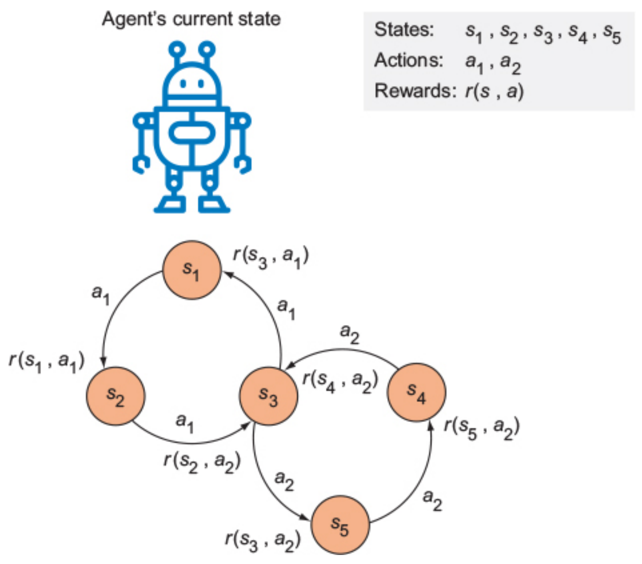
All these examples can be unified under a general formulation: performing an action in a scenario can yield a reward. A more technical term for scenario is *state*. And we call the collection of all possible states a *state space*. Performing an action causes the state to change. But the question is, what series of actions yields the highest expected rewards?

8.1. FORMAL NOTIONS

Whereas supervised and unsupervised learning appear at opposite ends of the spectrum, *reinforcement learning* (RL) exists somewhere in the middle. It’s not supervised learning, because the training data comes from the algorithm deciding between exploration and exploitation. And it’s not unsupervised, because the algorithm receives feedback from the environment. As long as you’re in a situation where performing an action in a state produces a reward, you can use reinforcement learning to discover a good sequence of actions to take that maximize expected rewards.

You may notice that reinforcement-learning lingo involves anthropomorphizing the algorithm into taking *actions* in *situations* to *receive rewards*. The algorithm is often referred to as an *agent* that *acts with* the environment. It shouldn’t be a surprise that much of reinforcement-learning theory is applied in robotics. Figure 8.2 demonstrates the interplay between states, actions, and rewards.

Figure 8.2. Actions are represented by arrows, and states are represented by circles. Performing an action on a state produces a reward. If you start at state *s*1, you can perform action *a*1 to obtain a reward *r*(*s*1, *a*1).



A robot performs actions to change states. But how does it decide which action to take? The next section introduces a new concept, called a *policy*, to answer this question.

Do humans use reinforcement learning?

Reinforcement learning seems like the best way to explain how to perform the next action based on the current situation. Perhaps humans behave the same way biologically. But let's not get ahead of ourselves; consider the following example.

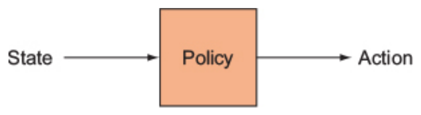
Sometimes, humans act without thinking. If I'm thirsty, I might instinctively grab a cup of water to quench my thirst. I don't iterate through all possible joint motions in my head and choose the optimal one after thorough calculations.

Most important, the actions we make aren't characterized solely by our observations at each moment. Otherwise, we're no smarter than bacteria, which act deterministically given their environment. There seems to be a lot more going on, and a simple RL model might not fully explain human behavior.

8.1.1. Policy

Everyone cleans their room differently. Some people start by making their bed. I prefer cleaning my room clockwise so I don't miss a corner. Have you ever seen a robotic vacuum cleaner, such as a Roomba? Someone programmed a strategy it can follow to clean any room. In reinforcement-learning lingo, the way an agent decides which action to take is called a *policy*: it's the set of actions that determines the next state (see figure 8.3).

Figure 8.3. A policy suggests which action to take, given a state.



The goal of reinforcement learning is to discover a good policy. A common way to create that policy is by observing the long-term consequences of actions at each state. The *reward* is the measure of the outcome of taking an action. The best possible policy is called the *optimal policy*, and it's often the Holy Grail of reinforcement learning. The optimal policy tells you the optimal action, given any state—but it may not provide the highest reward at the moment.

If you measure the reward by looking at the immediate consequence—the state of things after taking the action—it's easy to calculate. This is called the *greedy strategy*—but it's not always a good idea to "greedily" choose the action with the best *immediate* reward. For example, when cleaning your room, you might make your bed first, because the room looks neater with the bed made. But if another goal is to wash your sheets, making the bed first may not be the best overall strategy. You need to look at the results of the next few actions, and the eventual end state, to come up with the optimal approach. Similarly, in chess, grabbing your opponent's queen may maximize the points for the pieces on the board—but if it puts you in checkmate five moves later, it isn't the best possible move.

You can also arbitrarily choose an action: this is a *random policy*. If you come up with a policy to solve a reinforcement-learning problem, it's often a good idea to double-check that your learned policy performs better than both the random and greedy policies.

Limitations of (Markovian) reinforcement learning

Most RL formulations assume that the best action to take can be figured out from knowing the current state, instead of considering the longer-term history of states and actions that got you there. This approach of making decisions based on the current state is called *Markovian*, and the general framework is often referred to as the *Markov decision process* (MDP).

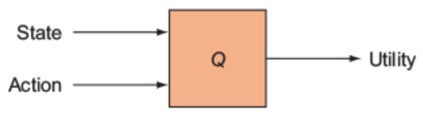
Such situations in which the state sufficiently captures what to do next can be modeled with RL algorithms discussed in this chapter. But most real-world situations aren't Markovian and therefore need a more realistic approach, such as a hierarchical representation of states and actions. In a grossly oversimplified sense, hierarchical models are like context-free grammars, whereas MDPs are like finite-state machines. The expressive leap of modeling a problem as an MDP to something more hierarchical can dramatically improve the effectiveness of the planning algorithm.

8.1.2. Utility

The long-term reward is called a *utility*. If you know the utility of performing an action at a state, learning the policy is easy using reinforcement learning. For example, to decide which action to take, you select the action that produces the highest utility. The hard part, as you might have guessed, is uncovering these utility values.

The utility of performing an action a at a state s is written as a function $Q(s, a)$, called the *utility function*, shown in figure 8.4.

Figure 8.4. Given a state and the action taken, applying a utility function *Q* predicts the expected and the total rewards: the immediate reward (next state) plus rewards gained later by following an optimal policy.



Exercise 8.1

If you were given the utility function *Q*(*s*, *a*), how could you use it to derive a policy function?

Answer

Policy(*s*) = argmax_a *Q*(*s*, *a*)

An elegant way to calculate the utility of a particular state-action pair (*s*, *a*) is by recursively considering the utilities of future actions. The utility of your current action is influenced not only by the immediate reward but also by the next best action, as shown in the next formula. In the formula, *s*' denotes the next state, and *a*' denotes the next action. The reward of taking action *a* in state *s* is denoted by *r*(*s*, *a*):

$$Q(s, a) = r(s, a) + \gamma \max Q(s', a')$$

Here, γ is a hyperparameter that you get to choose, called the *discount factor*. If γ is 0, the agent chooses the action that maximizes the immediate reward. Higher values of γ will make the agent put more importance on considering long-term consequences. You can read the formula as "the value of this action is the immediate reward provided by taking this action, added to the discount factor times the best thing that can happen after that."

Looking ahead at future rewards is one type of hyperparameter you can play with, but there's also another. In some applications of reinforcement learning, newly available information might be more important than historical records, or vice versa. For example, if a robot is expected to learn to solve tasks quickly but not necessarily optimally, you might want to set a faster learning rate. Or if a robot is allowed more time to explore and exploit, you might tune down the learning rate. Let's call the learning rate α, and change the utility function as follows (notice that when α = 1, both equations are identical).

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r(s, a) + \gamma \max Q(s', a') - Q(s, a))$$

Reinforcement learning can be solved if you know the Q-function: *Q*(*s*, *a*). Conveniently for us, *neural networks* (chapter 7) are a way to approximate functions, given enough training data. TensorFlow is the perfect tool to deal with neural networks because it comes with many essential algorithms to simplify neural network implementation.

8.2. APPLYING REINFORCEMENT LEARNING

Application of reinforcement learning requires defining a way to retrieve rewards after an action is taken from a state. A stock-market trader fits these requirements easily, because buying and selling a stock changes the state of the trader (cash on hand), and each action generates a reward (or loss).

The states in this situation are a vector containing information about the current budget, the current number of stocks, and a recent history of stock prices (the last 200 stock prices). Each state is a 202-dimensional vector.

Exercise 8.2

What are some possible disadvantages of using reinforcement learning for buying and selling stocks?

Answer

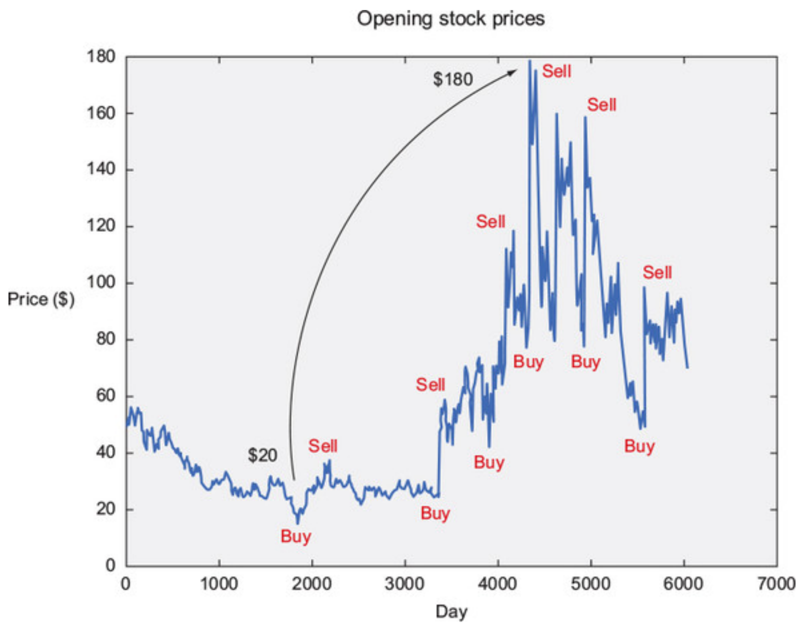
By performing actions on the market, such as buying or selling shares, you could end up influencing the market, causing it to change dramatically from your training data.

For simplicity, there are only three actions—buy, sell, and hold:

- Buying a stock at the current stock price decreases the budget while incrementing the current stock count.
- Selling a stock trades it in for money at the current share price.
- Holding does neither. This action waits a single time period and yields no reward.

Figure 8.5 demonstrates one possible policy, given stock market data.

Figure 8.5. Ideally, our algorithm should buy low and sell high. Doing so just once, as shown here, might yield a reward of around $160. But the real profit rolls in when you buy and sell more frequently. Ever heard the term *high-frequency trading*? It's about buying low and selling high as frequently as possible to maximize profits within a period of time.



The goal is to learn a policy that gains the maximum net worth from trading in a stock market. Wouldn't that be cool? Let's do it!

8.3. IMPLEMENTING REINFORCEMENT LEARNING

To gather stock prices, you'll use the `yahoo_finance` library in Python. You can install it using `pip` or follow the official guide (https://pypi.python.org/pypi/yahoo-finance). The command to install it using `pip` is as follows:

```
$ pip install yahoo-finance
```

With that installed, let's import all the relevant libraries.

Listing 8.1. Importing relevant libraries

```
from yahoo_finance import Share              1
from matplotlib import pyplot as plt         2
import numpy as np                           3
import tensorflow as tf                      3
import random
```

- *1* **For obtaining stock-price raw data**
- *2* **For plotting stock prices**
- *3* **For numeric manipulation and machine learning**

Create a helper function to get stock prices by using the `yahoo_finance` library. The library requires three pieces of information: share symbol, start date, and end date. When you pick each of the three values, you'll get a list of numbers representing the share prices in that period by day.

If you choose a start and end date too far apart, it'll take some time to fetch that data. It might be a good idea to save (that is, cache) the data to disk so you can load it locally next time. See the following listing for how to use the library and cache the data.

Listing 8.2. Helper function to get prices

```
def get_prices(share_symbol, start_date, end_date,
               cache_filename='stock_prices.npy'):
    try:                                                                   1
        stock_prices = np.load(cache_filename)
    except IOError:
        share = Share(share_symbol)                                        2
        stock_hist = share.get_historical(start_date, end_date)
        stock_prices = [stock_price['Open'] for stock_price in stock_hist] 3
        np.save(cache_filename, stock_prices)                              4

    return stock_prices.astype(float)
```

- *1* **Tries to load the data from file if it has already been computed**
- *2* **Retrieves stock prices from the library**
- *3* **Extracts only relevant info from the raw data**

- *4* **Caches the result**

Just for a sanity check, it's a good idea to visualize the stock-price data. Create a plot, and save it to disk.

Listing 8.3. Helper function to plot the stock prices

```
def plot_prices(prices):
    plt.title('Opening stock prices')
    plt.xlabel('day')
    plt.ylabel('price ($)')
    plt.plot(prices)
    plt.savefig('prices.png')
    plt.show()
```

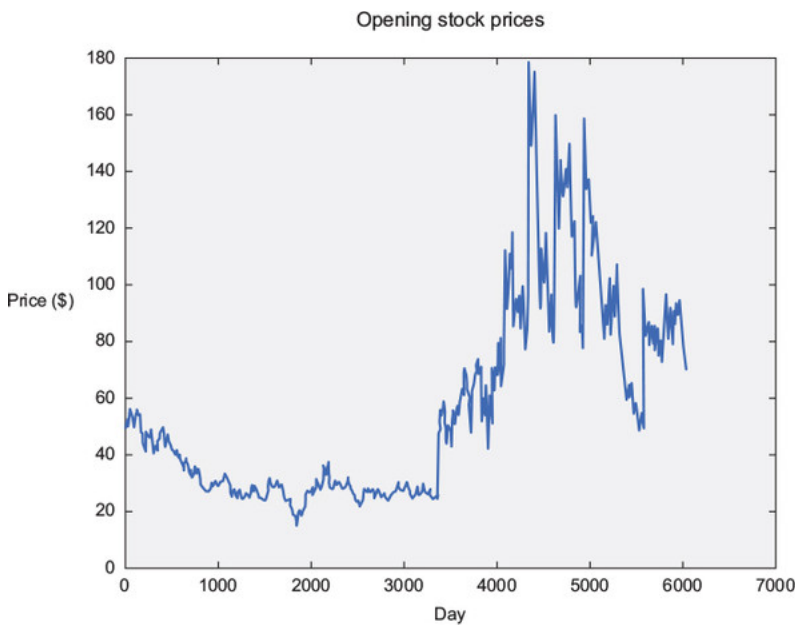
You can grab some data and visualize it by using the following listing.

Listing 8.4. Get data and visualize it

```
if __name__ == '__main__':
  prices = get_prices('MSFT', '1992-07-22', '2016-07-22')
  plot_prices(prices)
```

Figure 8.6 shows the chart produced from running listing 8.4.

Figure 8.6. This chart summarizes the opening stock prices of Microsoft (MSFT) from 7/22/1992 to 7/22/2016. Wouldn't it have been nice to buy around day 3000 and sell around day 5000? Let's see if our code can learn to buy, sell, and hold to make optimal gain.



Most reinforcement-learning algorithms follow similar implementation patterns. As a result, it's a good idea to create a class with the relevant methods to reference later, such as an abstract class or interface. See the following listing for an example and figure 8.7 for an illustration. Reinforcement learning needs two operations well defined: how to select an action, and how to improve the utility Q-function.

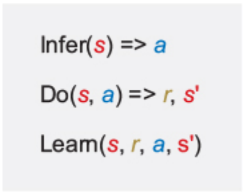
Listing 8.5. Defining a superclass for all decision policies

```
class DecisionPolicy:
    def select_action(self, current_state):                 1
```

```
        pass

    def update_q(self, state, action, reward, next_state):  2
        pass
```

- **1 Given a state, the decision policy will calculate the next action to take.**
- **2 Improve the Q-function from a new experience of taking an action.**

Figure 8.7. Most reinforcement-learning algorithms boil down to just three main steps: infer, do, and learn. During the first step, the algorithm selects the best action (*a*), given a state (*s*), using the knowledge it has so far. Next, it does the action to find out the reward (*r*) as well as the next state (*s*'). Then it improves its understanding of the world by using the newly acquired knowledge (*s*, *r*, *a*, *s*').



Next, let's inherit from this superclass to implement a policy where decisions are made at random, otherwise known as a *random decision policy*. You need to define only the `select_action` method, which will randomly pick an action without even looking at the state. The following listing shows how to implement it.

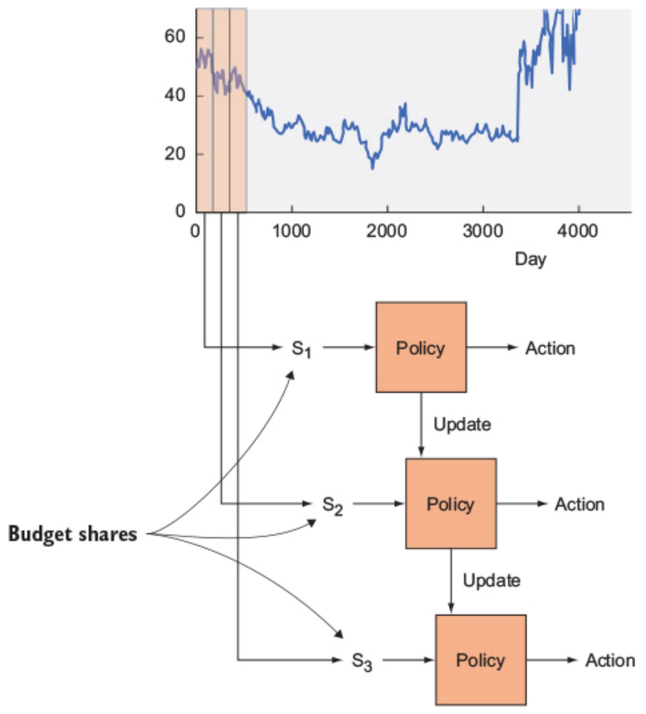
Listing 8.6. Implementing a random decision policy

```
class RandomDecisionPolicy(DecisionPolicy):           1
    def __init__(self, actions):
        self.actions = actions

    def select_action(self, current_state):           2
        action = random.choice(self.actions)
        return action
```

- **1 Inherits from DecisionPolicy to implement its functions**
- **2 Randomly chooses the next action**

In listing 8.7, you assume a policy is given to you (such as the one from listing 8.6) and run it on the real-world stock-price data. This function takes care of exploration and exploitation at each interval of time. Figure 8.8 illustrates the algorithm from listing 8.7.

Figure 8.8. A rolling window of a certain size iterates through the stock prices, as shown by the chart segmented to form states S_1, S_2, and S_3. The policy suggests an action to take: you may either choose to exploit it or randomly explore another action. As you get rewards for performing an action, you can update the policy function over time.



Listing 8.7. Using a given policy to make decisions, and returning the performance

```
def run_simulation(policy, initial_budget, initial_num_stocks, prices, hist):
    budget = initial_budget                                                 1
    num_stocks = initial_num_stocks                                         1
    share_value = 0                                                         1
    transitions = list()
    for i in range(len(prices) - hist - 1):
        if i % 1000 == 0:
            print('progress {:.2f}%'.format(float(100*i) / (len(prices) -
     hist - 1)))
        current_state = np.asmatrix(np.hstack((prices[i:i+hist], budget,
     num_stocks)))                                                          2
        current_portfolio = budget + num_stocks * share_value               3
        action = policy.select_action(current_state, i)                     4
        share_value = float(prices[i + hist])
        if action == 'Buy' and budget >= share_value:                       5
            budget -= share_value
            num_stocks += 1
        elif action == 'Sell' and num_stocks > 0:                           5
            budget += share_value
            num_stocks -= 1
        else:                                                               5
            action = 'Hold'
        new_portfolio = budget + num_stocks * share_value                   6
        reward = new_portfolio - current_portfolio                          7
        next_state = np.asmatrix(np.hstack((prices[i+1:i+hist+1], budget,
     num_stocks)))
        transitions.append((current_state, action, reward, next_state))
        policy.update_q(current_state, action, reward, next_state)          8

    portfolio = budget + num_stocks * share_value                           9
    return portfolio
```

- ***1*** **Initializes values that depend on computing the net worth of a portfolio**
- ***2*** **The state is a hist + 2 dimensional vector. You’ll force it to be a NumPy matrix.**

- ***3*** **Calculates the portfolio value**
- ***4*** **Selects an action from the current policy**
- ***5*** **Updates portfolio values based on action**
- ***6*** **Computes a new portfolio value after taking action**
- ***7*** **Computes the reward from taking an action at a state**
- ***8*** **Updates the policy after experiencing a new action**
- ***9*** **Computes the final portfolio worth**

To obtain a more robust measurement of success, let's run the simulation a couple of times and average the results. Doing so may take a while to complete (perhaps 5 minutes), but your results will be more reliable.

Listing 8.8. Running multiple simulations to calculate an average performance

```
def run_simulations(policy, budget, num_stocks, prices, hist):
    num_tries = 10                                                         1
    final_portfolios = list()                                              2
    for i in range(num_tries):
        final_portfolio = run_simulation(policy, budget, num_stocks, prices,
     hist)                                                                 3
        final_portfolios.append(final_portfolio)
        print('Final portfolio: ${}'.format(final_portfolio))
    plt.title('Final Portfolio Value')
    plt.xlabel('Simulation #')
    plt.ylabel('Net worth')
    plt.plot(final_portfolios)
    plt.show()
```

- ***1*** **Decides the number of times to rerun the simulations**
- ***2*** **Stores the portfolio worth of each run in this array**
- ***3*** **Runs this simulation**

In the `main` function, append the following lines to define the decision policy and run simulations to see how it performs.

Listing 8.9. Defining the decision policy

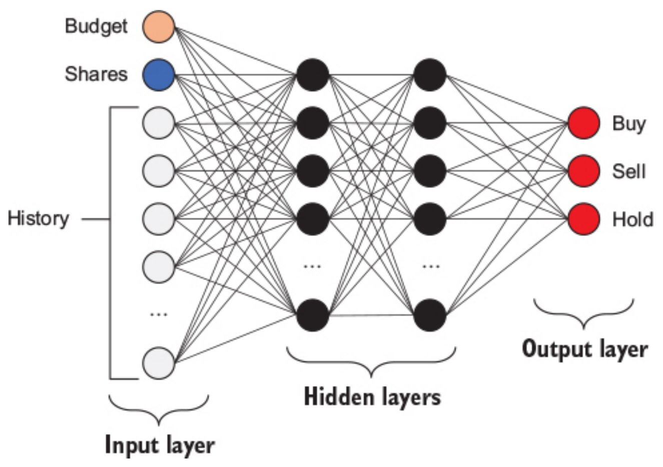
```
if __name__ == '__main__':
    prices = get_prices('MSFT', '1992-07-22', '2016-07-22')
    plot_prices(prices)
    actions = ['Buy', 'Sell', 'Hold']                                1
    hist = 3
    policy = RandomDecisionPolicy(actions)                           2
    budget = 100000.0                                                3
    num_stocks = 0                                                   4
    run_simulations(policy, budget, num_stocks, prices, hist)        5
```

- ***1*** **Defines the list of actions the agent can take**
- ***2*** **Initializes a random decision policy**
- ***3*** **Sets the initial amount of money available to use**
- ***4*** **Sets the number of stocks already owned**

- **5 Runs simulations multiple times to compute the expected value of your final net worth**

Now that you have a baseline to compare your results, let's implement a neural network approach to learn the Q-function. The decision policy is often called the *Q-learning decision policy*. listing 8.10 introduces a new hyperparameter, `epsilon`, to keep the solution from getting "stuck" when applying the same action over and over. The lower its value, the more often it will randomly explore new actions. The Q-function is defined by the function depicted in figure 8.9.

Figure 8.9. The input is the state space vector, with three outputs: one for each output's Q-value.



Exercise 8.3

What are other possible factors that your state-space representation ignores that can affect the stock prices? How could you factor them into the simulation?

Answer

Stock prices depend on a variety of factors, including general market trends, breaking news, and specific industry trends. Each of these, once quantified, could be applied as additional dimensions to the model.

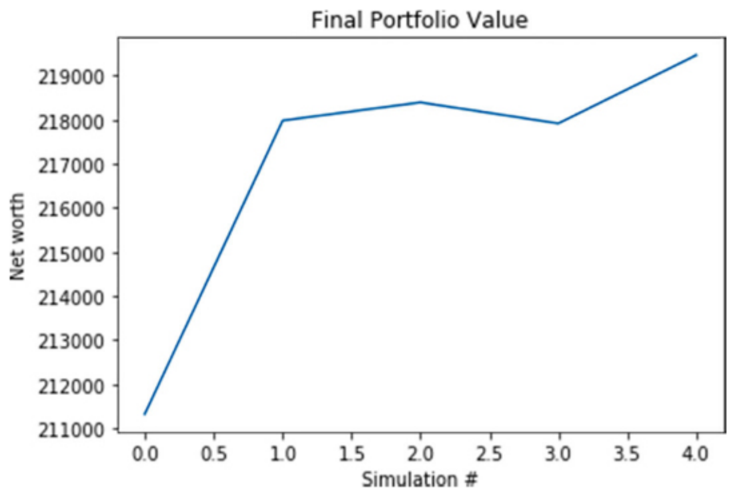
Listing 8.10. Implementing a more intelligent decision policy

```
class QLearningDecisionPolicy(DecisionPolicy):
    def __init__(self, actions, input_dim):
        self.epsilon = 0.95                                              1
        self.gamma = 0.3                                                 1
        self.actions = actions
        output_dim = len(actions)
        h1_dim = 20                                                      2

        self.x = tf.placeholder(tf.float32, [None, input_dim])           3
        self.y = tf.placeholder(tf.float32, [output_dim])                3
        W1 = tf.Variable(tf.random_normal([input_dim, h1_dim]))          4
        b1 = tf.Variable(tf.constant(0.1, shape=[h1_dim]))               4
        h1 = tf.nn.relu(tf.matmul(self.x, W1) + b1)                      4
        W2 = tf.Variable(tf.random_normal([h1_dim, output_dim]))         4
        b2 = tf.Variable(tf.constant(0.1, shape=[output_dim]))           4
        self.q = tf.nn.relu(tf.matmul(h1, W2) + b2)                      5
        loss = tf.square(self.y - self.q)                                6
        self.train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)   7
        self.sess = tf.Session()                                         7
        self.sess.run(tf.global_variables_initializer())                 8

    def select_action(self, current_state, step):
        threshold = min(self.epsilon, step / 1000.)
        if random.random() < threshold:                                  9
               # Exploit best option with probability epsilon
            action_q_vals = self.sess.run(self.q, feed_dict={self.x: current_
            action_idx = np.argmax(action_q_vals)
            action = self.actions[action_idx]
        else:                                                            1
            # Explore random option with probability 1 - epsilon
            action = self.actions[random.randint(0, len(self.actions) - 1)]
        return action

    def update_q(self, state, action, reward, next_state):               11
        action_q_vals = self.sess.run(self.q, feed_dict={self.x: state})
        next_action_q_vals = self.sess.run(self.q, feed_dict={self.x:
     next_state})
        next_action_idx = np.argmax(next_action_q_vals)
        current_action_idx = self.actions.index(action)
        action_q_vals[0, current_action_idx] = reward + self.gamma
   * next_action_q_vals[0, next_action_idx]
        action_q_vals = np.squeeze(np.asarray(action_q_vals))
        self.sess.run(self.train_op, feed_dict={self.x: state, self.y:
     action_q_vals})
```

- **_1_ Sets the hyperparameters from the Q-function**
- **_2_ Sets the number of hidden nodes in the neural networks**
- **_3_ Defines the input and output tensors**
- **_4_ Designs the neural network architecture**
- **_5_ Defines the op to compute the utility**
- **_6_ Sets the loss as the square error**
- **_7_ Uses an optimizer to update model parameters to minimize the loss**
- **_8_ Sets up the session, and initializes variables**
- **_9_ Exploits the best option with probability epsilon**
- **_10_ Explores a random option with probability 1 - epsilon**
- **_11_ Updates the Q-function by updating its model parameters**

The resulting output when running the entire script is shown in figure 8.10.

Figure 8.10. The algorithm learns a good policy to trade Microsoft stocks.



8.4. EXPLORING OTHER APPLICATIONS OF REINFORCEMENT LEARNING

Reinforcement learning is used more often than you might expect. It's too easy to forget that it exists when you've learned supervised- and unsupervised-learning methods. But the following examples will open your eyes to successful uses of RL by Google:

- *Game playing*—In February 2015, Google developed a reinforcement-learning system called Deep RL to learn how to play arcade video games from the Atari 2600 console. Unlike most RL solutions, this algorithm had a high-dimensional input: it perceived the raw frame-by-frame images of the video game. That way, the same algorithm could work with any video game without much reprogramming or reconfiguring.
- *More game playing*—In January 2016, Google released a paper about an AI agent capable of winning the board game Go. The game is known to be unpredictable because of the enormous number of possible configurations (even more than chess!), but this algorithm using RL could beat top human Go players. The latest version, AlphaGo Zero, was released in late 2017 and was able to beat the earlier version consistently—100 games to 0—in only 40 days of training. It will be considerably better than that by the time you read this.
- *Robotics and control*—In March 2016, Google demonstrated a way for a robot to learn by many examples how to grab an object. Google collected more than 800,000 grasp attempts by using multiple robots and developed a model to grasp arbitrary objects. Impressively, the robots were capable of grasping an object with the help of camera input alone. Learning the simple concept of grasping an object required aggregating the knowledge of many robots spending many days in brute-force attempts until enough patterns were detected. Clearly, there's a long way to go for robots to be able to generalize, but it's an interesting start, nonetheless.

Note

Now that you've applied reinforcement learning to the stock market, it's time for you to drop out of school or quit your job and start gaming the system. Turns out this is your payoff, dear reader, for making it this far into the book! Just kidding—the actual stock market is a much more complicated beast, but the techniques used in this chapter generalize to many situations.

8.5. SUMMARY

- Reinforcement learning is a natural tool for problems that can be framed by states that change due to actions taken by an agent to discover rewards.
- Implementing the reinforcement-learning algorithm requires three primary steps: infer the best action from the current state, perform the action, and learn from the results.
- Q-learning is an approach to solving reinforcement learning whereby you develop an algorithm to approximate the utility function (Q-function). After a good enough approximation is found, you can start inferring the best actions to take from each state.