CHAPTER 3
Financial Q-Learning

> Today's algorithmic trading programs are relatively simple and make only limited use of AI. This is sure to change.
>
> —Murray Shanahan (2015)

The previous chapter shows that a deep Q-learning (DQL) agent can learn to play the game of *CartPole* quite well. What about financial applications? As this chapter shows, the agent can also learn to play a financial game that is about predicting the future movement in a financial market. To this end, this chapter implements a `Finance` environment that mimics the behavior of the `CartPole` environment and trains the DQL agent from the previous chapter based on the requirements of the `Finance` environment.

This chapter is brief, but it illustrates an important point: with the appropriate environment, DQL can be applied to financial problems basically in the same way as it is applied to games and in other domains. "Finance Environment" on page 37 develops step-by-step the `Finance` class that mimics the behavior of the `CartPole` class. "DQL Agent" on page 43 slightly adjusts the `DQLAgent` class from "CartPole as an Example" on page 26. The adjustments are made to reflect the new context. The DQL agent can learn to predict future market movements with a significant margin over the baseline accuracy of 50%. "Where the Analogy Fails" on page 45 finally discusses the major issues of the modeling approach and the `Finance` class when compared, for example, to a gaming environment such as the *CartPole* game.

Finance Environment

The goal in this section is to implement a `Finance` environment as a prediction game. The environment uses static historical financial time series data to generate the states of the environment and the value to be predicted by the DQL agent. The state is given

by four floating-point numbers representing the four most recent data points in the time series—such as normalized price or return values. The value to be predicted is either 0 or 1. Here, 0 means that the financial time series value drops to a lower level ("market goes down") and 1 means that the time series value rises to a higher level ("market goes up").

To get started, the following Python class implements the behavior of the `env.action_space` object for the generation of random actions. The DQL agent relies on this capability in the context of exploration:

```
In [1]: import os
        import random

In [2]: random.seed(100)
        os.environ['PYTHONHASHSEED'] = '0'

In [3]: class ActionSpace:
            def sample(self):
                return random.randint(0, 1)

In [4]: action_space = ActionSpace()

In [5]: [action_space.sample() for _ in range(10)]
Out[5]: [0, 1, 1, 0, 1, 1, 1, 0, 0, 0]
```

The `Finance` class, which is at the core of this chapter, implements the idea of the prediction game as described previously. It starts with the definition of important parameters and objects:

```
In [6]: import numpy as np
        import pandas as pd

In [7]: class Finance:
            url = 'https://certificate.tpq.io/rl4finance.csv'  ❶
            def __init__(self, symbol, feature,
                         min_accuracy=0.485, n_features=4):
                self.symbol = symbol  ❷
                self.feature = feature  ❸
                self.n_features = n_features  ❹
                self.action_space = ActionSpace()  ❺
                self.min_accuracy = min_accuracy  ❻
                self._get_data()  ❼
                self._prepare_data()  ❽
            def _get_data(self):
                self.raw = pd.read_csv(self.url,
                               index_col=0, parse_dates=True)  ❼
```

❶ The URL for the data set to be used (which can be replaced)

❷ The symbol for the time series to be used for the prediction game

❸ The type of feature to be used to define the state of the environment

❹ The number of feature values to be provided to the agent

❺ The `ActionSpace` object that is used for random action sampling

❻ The minimum prediction accuracy required for the agent to continue with the prediction game

❼ The retrieval of the financial time series data from the remote source

❽ The method call for the data preparation

The data set used in this class allows the selection of the following financial instruments:

```
AAPL.O | Apple Stock
MSFT.O | Microsoft Stock
INTC.O | Intel Stock
AMZN.O | Amazon Stock
GS.N   | Goldman Sachs Stock
SPY    | SPDR S&P 500 ETF Trust
.SPX   | S&P 500 Index
.VIX   | VIX Volatility Index
EUR=   | EUR/USD Exchange Rate
XAU=   | Gold Price
GDX    | VanEck Vectors Gold Miners ETF
GLD    | SPDR Gold Trust
```

A key method of the `Finance` class is the one for preparing the data for both the state description (features) and the prediction itself (labels). The state data is provided in normalized form, which is known to improve the performance of deep neural networks (DNNs). From the implementation, it is obvious that the financial time series data is used in a static, nonrandom way. When the environment is reset to the initial state, it is always the same initial state:

```
In [8]: class Finance(Finance):
            def _prepare_data(self):
                self.data = pd.DataFrame(self.raw[self.symbol]).dropna() ❶
                self.data['r'] = np.log(self.data / self.data.shift(1)) ❷
                self.data['d'] = np.where(self.data['r'] > 0, 1, 0) ❸
                self.data.dropna(inplace=True) ❹
                self.data_ = (self.data - self.data.mean()) / self.data.std() ❺
            def reset(self):
                self.bar = self.n_features ❻
                self.treward = 0 ❼
                state = self.data_[self.feature].iloc[
                    self.bar - self.n_features:self.bar].values ❽
                return state, {}
```

❶ Selects the relevant time series data from the `DataFrame` object

❷ Generates a log return time series from the price time series

❸ Generates the binary, directional data to be predicted from the log returns

❹ Gets rid of all those rows in the `DataFrame` object that contain `NaN` ("not a number") values

❺ Applies Gaussian normalization to the data

❻ Sets the current bar (position in the time series) to the value for the number of feature values

❼ Resets the total reward value to zero

❽ Generates the initial state object to be returned by the method

The following Python code finally implements the `.step()` method, which moves the environment from one state to the next or signals that the game is terminated. One key idea is to check for the current prediction accuracy of the agent and to compare it to a minimum required accuracy. The purpose is to avoid a situation where the agent simply plays along even if its current performance is much worse than, say, that of a random agent:

```
In [9]: class Finance(Finance):
            def step(self, action):
                if action == self.data['d'].iloc[self.bar]: ❶
                    correct = True
                else:
                    correct = False
                reward = 1 if correct else 0 ❷
                self.treward += reward ❸
                self.bar += 1 ❹
                self.accuracy = self.treward / (self.bar - self.n_features)  ❺
                if self.bar >= len(self.data): ❻
                    done = True
                elif reward == 1: ❼
                    done = False
                elif (self.accuracy < self.min_accuracy) and (self.bar > 15): ❽
                    done = True
                else:
                    done = False
                next_state = self.data_[self.feature].iloc[
                    self.bar - self.n_features:self.bar].values ❾
                return next_state, reward, done, False, {}
```

❶ Checks whether the prediction ("action") is correct.

❷ Assigns a reward of `+1` or `0`, depending on correctness.

❸ Increases the total reward accordingly.

❹ The bar value is increased to move the environment forward on the time series.

❺ The current accuracy is calculated.

❻ Checks whether the end of the data set is reached.

❼ Checks whether the prediction is correct.

❽ Checks whether the current accuracy is above the minimum required accuracy.

❾ Generates the next state object to be returned by the method.

This completes the `Finance` class and allows the instantiation of objects based on the class, as in the following Python code. The code also lists the available symbols in the financial data set used. It further illustrates that either normalized price or log returns data can be used to describe the state of the environment:

```
In [10]: fin = Finance(symbol='EUR=', feature='EUR=')  ❶

In [11]: list(fin.raw.columns)  ❷
Out[11]: ['AAPL.O',
          'MSFT.O',
          'INTC.O',
          'AMZN.O',
          'GS.N',
          '.SPX',
          '.VIX',
          'SPY',
          'EUR=',
          'XAU=',
          'GDX',
          'GLD']

In [12]: fin.reset()
         # four lagged, normalized price points
Out[12]: (array([2.74844931, 2.64643904, 2.69560062, 2.68085214]), {})

In [13]: fin.action_space.sample()
Out[13]: 1

In [14]: fin.step(fin.action_space.sample())
Out[14]: (array([2.64643904, 2.69560062, 2.68085214, 2.63046153]), 0, False,
          False, {})
```

```
In [15]: fin = Finance('EUR=', 'r')  ❸

In [16]: fin.reset()
         # four lagged, normalized log returns
Out[16]: (array([-1.19130476, -1.21344494,  0.61099805, -0.16094865]), {})
```

❶ Specifies that the feature type is *normalized prices*

❷ Shows the available symbols in the data set used

❸ Specifies that the feature type is *normalized returns*

To illustrate the interaction with the `Finance` environment, a random agent can again be considered. The total rewards that the agent achieves are, of course, quite low. They are below 20 on average. This needs to be compared with the length of the data set, which has more than 2,500 data points. In other words, a total reward of 2,500 or more is possible:

```
In [17]: class RandomAgent:
             def __init__(self):
                 self.env = Finance('EUR=', 'r')
             def play(self, episodes=1):
                 self.trewards = list()
                 for e in range(episodes):
                     self.env.reset()
                     for step in range(1, 100):
                         a = self.env.action_space.sample()
                         state, reward, done, trunc, info = self.env.step(a)
                         if done:
                             self.trewards.append(step)
                             break

In [18]: ra = RandomAgent()

In [19]: ra.play(15)

In [20]: ra.trewards
Out[20]: [17, 13, 17, 12, 12, 12, 13, 23, 31, 13, 12, 15]

In [21]: round(sum(ra.trewards) / len(ra.trewards), 2)  ❶
Out[21]: 15.83

In [22]: len(fin.data)  ❷
Out[22]: 2607
```

❶ Average reward for the random agent

❷ Length of the data set, which equals roughly the maximum total reward

DQL Agent

Equipped with the `Finance` environment, it is straightforward to let the DQL agent (the `DQLAgent` class from "The DQL Agent" on page 29) play the financial prediction game.

The following Python code takes care of the required imports and configurations:

```
In [23]: import os
         import random
         import warnings
         import numpy as np
         import tensorflow as tf
         from tensorflow import keras
         from collections import deque
         from keras.layers import Dense
         from keras.models import Sequential

In [24]: warnings.simplefilter('ignore')
         os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

In [25]: from tensorflow.python.framework.ops import disable_eager_execution
         disable_eager_execution()

In [26]: opt = keras.optimizers.legacy.Adam(learning_rate=0.0001)
```

For the sake of completeness, the following code shows the `DQLAgent` class as a whole. It is basically the same code as in "The DQL Agent" on page 29, with some minor adjustments for the context of this chapter:

```
In [27]: class DQLAgent:
             def __init__(self, symbol, feature, min_accuracy, n_features=4):
                 self.epsilon = 1.0
                 self.epsilon_decay = 0.9975
                 self.epsilon_min = 0.1
                 self.memory = deque(maxlen=2000)
                 self.batch_size = 32
                 self.gamma = 0.5
                 self.trewards = list()
                 self.max_treward = 0
                 self.n_features = n_features
                 self._create_model()
                 self.env = Finance(symbol, feature,
                                    min_accuracy, n_features)  ❶
             def _create_model(self):
                 self.model = Sequential()
                 self.model.add(Dense(24, activation='relu',
                                      input_dim=self.n_features))
                 self.model.add(Dense(24, activation='relu'))
                 self.model.add(Dense(2, activation='linear'))
                 self.model.compile(loss='mse', optimizer=opt)
             def act(self, state):
```

```
        if random.random() < self.epsilon:
            return self.env.action_space.sample()
        return np.argmax(self.model.predict(state)[0])
    def replay(self):
        batch = random.sample(self.memory, self.batch_size)
        for state, action, next_state, reward, done in batch:
            if not done:
                reward += self.gamma * np.amax(
                    self.model.predict(next_state)[0])
            target = self.model.predict(state)
            target[0, action] = reward
            self.model.fit(state, target, epochs=1, verbose=False)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
    def learn(self, episodes):
        for e in range(1, episodes + 1):
            state, _ = self.env.reset()
            state = np.reshape(state, [1, self.n_features])
            for f in range(1, 5000):
                action = self.act(state)
                next_state, reward, done, trunc, _ = \
                    self.env.step(action)
                next_state = np.reshape(next_state,
                                        [1, self.n_features])
                self.memory.append(
                    [state, action, next_state, reward, done])
                state = next_state
                if done:
                    self.trewards.append(f)
                    self.max_treward = max(self.max_treward, f)
                    templ = f'episode={e:4d} | treward={f:4d}'
                    templ += f' | max={self.max_treward:4d}'
                    print(templ, end='\r')
                    break
            if len(self.memory) > self.batch_size:
                self.replay()
        print()
    def test(self, episodes):
        ma = self.env.min_accuracy  ❷
        self.env.min_accuracy = 0.5  ❸
        for e in range(1, episodes + 1):
            state, _ = self.env.reset()
            state = np.reshape(state, [1, self.n_features])
            for f in range(1, 5001):
                action = np.argmax(self.model.predict(state)[0])
                state, reward, done, trunc, _ = self.env.step(action)
                state = np.reshape(state, [1, self.n_features])
                if done:
                    tmpl = f'total reward={f} | '
                    tmpl += f'accuracy={self.env.accuracy:.3f}'
                    print(tmpl)
```

```
                break
        self.env.min_accuracy = ma  ❷
```

❶ Defines the `Finance` environment object as a instance attribute

❷ Captures and resets the original minimum accuracy for the `Finance` environment

❸ Redefines the minimum accuracy for testing purposes

As the following Python code shows, the `DQLAgent` learns to predict the next market movement with an accuracy of significantly above 50%:

```
In [28]: random.seed(250)
         tf.random.set_seed(250)

In [29]: agent = DQLAgent('EUR=', 'r', 0.495, 4)

In [30]: %time agent.learn(250)
         episode= 250 | treward=  12 | max=2603
         CPU times: user 18.6 s, sys: 3.15 s, total: 21.8 s
         Wall time: 18.2 s

In [31]: agent.test(5)  ❶
         total reward=2603 | accuracy=0.525
         total reward=2603 | accuracy=0.525
         total reward=2603 | accuracy=0.525
         total reward=2603 | accuracy=0.525
         total reward=2603 | accuracy=0.525
```

❶ Test results are all the same, given the static data set.

Where the Analogy Fails

The `Finance` environment as introduced in "Finance Environment" on page 37 has one major goal: to exactly replicate the API of the `CartPole` environment. This goal is relatively easily achieved, allowing the DQL agent from the previous chapter to learn the financial prediction game. This is an accomplishment and insight in and of itself: a DQL agent can learn to play different games—even a large number of them.

However, the `Finance` environment brings two major, intertwined drawbacks with it: limited data and no impact of actions. This section discusses them in some detail.

Limited Data

The first drawback is that the environment is based on a static, deterministic data set. Whenever the environment is reset, it starts at the same initial state and moves step-by-step through the same states afterward, independent of the action (prediction) of

the DQL agent. This is in stark contrast to the `CartPole` environment, which by default generates a random initial state. Given the random initial state, the whole set of ensuing states can be considered to be a sequence of random states since they always differ, given a new random initial state.

Here, it is important to note that the transition from one state to another is deterministic. However, all sequences of states will differ due to the initial state being random. In a certain sense, the sequence of states as a whole inherits its randomness from the initial state.

Working with static data sets severely limits the training data. Although the data set has more than 2,500 data points, it is just *one* data set. The situation is as if a reinforcement learning (RL) agent were learning to play chess based only on a single historical game, which it could go through over and over again. It is also comparable to a student preparing for an upcoming mathematics exam with only one mathematics problem available to study. Too little data is not only a problem in RL, but obviously in machine learning and deep learning in general.

Another thought should be outlined here as well. Even if one adds other historical financial time series to the training data set or if one uses, say, historical intraday data instead of end-of-day data, the problem of limited financial data persists. It might not be as severe as in the context of the `Finance` environment, but the problem still plays an important role.

Too Little Data

The success or failure of a DQL agent often depends on the availability of large amounts of or even infinite data. When playing board games such as chess, for example, the available data (experiences made) is practically infinite because an agent can play a very large number of games against itself. Financial data in and of itself is limited by definition.

No Impact

In RL with DQL agents, it is often assumed or expected that the next state of an environment depends on the action chosen by the agent, at least to some extent. In chess, it is clear that the next board position depends on the move of the player or the DQL agent trying to learn the game. In *CartPole*, the agent influences all four parameters of the next state—cart position, cart velocity, pole angle, and angular velocity—by pushing the cart to the left or right.

In *The Book of Why*, Pearl and Mackenzie (2018) explain that there are three layers from which one can learn and formulate causal relationships. The first layer is data that can be observed, processed, and analyzed. For example, analyzing data might

lead one to insights concerning the correlation between two related quantities. But, as is often pointed out, correlation is not necessarily causation.

To get deeper insights into what might really *cause* a phenomenon or an observation, one needs the other two layers. The second layer is about *interventions*. In the real world, one can in general expect that an action has some impact. Whether I exercise regularly or not should make a difference in the evolution of my weight and health, for example. This is comparable to the `CartPole` environment, in which every action has a direct impact.

In the `Finance` environment, the next state is completely independent of the prediction (action) of the DQL agent. In this context, it might be acceptable that way, because, after all, what impact shall a prediction of a DQL agent (or a human analyst) have on the evolution of the EUR/USD exchange rate or the Apple share price? In finance, it is routinely assumed that agents are "infinitesimally small" and therefore cannot impact financial markets through trading or other financial decisions.

In reality, of course, large financial institutions often have a significant influence on financial markets, for example, when executing a large order or block trade. In such a context, feedback effects of actions would be highly relevant for the learning of optimal execution strategies, for instance.

Going one level higher and recalling what RL is about at its core, it should also be clear that the consequences of actions should play an important role. How should "reinforcement" otherwise be happening if the consequences of actions have no effect? The situation is comparable to a student receiving the same feedback from their parents no matter whether they get an A or D grade on a mathematics exam. For a comprehensive discussion about the role the consequences of actions play for human beings and animals alike, see the book *The Science of Consequences* by Schneider (2012).

The third layer is about *counterfactuals*. This implies that an agent possesses the capabilities to imagine hypothetical states for an environment and to hypothetically simulate the impact that a hypothetical action might have. This probably cannot be expected entirely from a DQL agent as discussed in this book. It might be something for which an artificial general intelligence (AGI) might be required.[1] On a simpler level, one could interpret the simulation of a hypothetical future action that is optimal as coming up with a counterfactual. The DQL agent does not, however, hypothesize about possible states that it has not experienced before.

1 For the definitions of different types of AI, see, for example, Hilpisch (2020, Chapter 2).

No Impact

In this book, it is usually assumed that a DQL agent's actions have no direct effect on the next state. A state is given, and, independent of which action the agent chooses, the next state is revealed to the agent. This holds for static historical data sets or those generated in Part II based on adding noise, leveraging simulation techniques, or using generative adversarial networks.

Conclusions

This chapter develops a simple financial environment that allows the DQL agent from the previous chapter (with some minor adjustments) to learn a financial prediction game. The environment is based on real historical financial price data. The DQL agent learns to predict the future movement of the market (the price of the financial instrument chosen) with an accuracy that is significantly above the 50% baseline level.

While the financial environment developed in this chapter mimics the major elements of the API as provided by the `CartPole` environment, it lacks two important elements: the training data set is limited to a single, static time series only, and the actions of the DQL agents do not impact the state of the environment.

Part II focuses on the major problem of limited financial data and introduces data augmentation approaches that allow you to generate a basically unlimited number of financial time series.

References

- Hilpisch, Yves. *Artificial Intelligence in Finance: A Python-Based Guide*. Sebastopol, CA: O'Reilly, 2020.
- Pearl, Judea, and Dana Mackkenzie. *The Book of Why: The New Science of Cause and Effect*. New York: Basic Books, 2018.
- Shanahan, Murray. *The Technological Singularity*. Cambridge and London: MIT Press, 2015.
- Schneider, Susan M. *The Science of Consequences: How They Affect Genes, Change the Brain, and Impact Our World*. Amherst, MA: Prometheus Books, 2012.