O'REILLY®

Reinforcement Learning for Finance

A Python-Based Introduction

Early Release
RAW & UNEDITED

Yves J. Hilpisch

Reinforcement Learning for Finance

A Python-Based Introduction

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Yves J. Hilpisch

Beijing • Boston • Farnham • Sebastopol • Tokyo

Reinforcement Learning for Finance

by Yves J. Hilpisch

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Michelle Smith
- Development Editor: Corbin Collins
- Production Editor: Gregory Hyman
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- November 2024: First Edition

Revision History for the Early Release

- 2024-03-27: First Release

See *https://www.oreilly.com/catalog/errata.csp?isbn=9781098169145* for release details.

property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-16914-5

[LSI]

Brief Table of Contents (*Not Yet Final*)

Part I. The Basics

The first part of the book covers the basics of reinforcement learning and provides background information. It consists of three chapters:

- Chapter 1 focuses on learning through interaction with four major examples: probability matching, Bayesian updating, reinforcement learning (RL), and deep-Q-learning (DQL).
- Chapter 2 introduces concepts from dynamic programming (DP) and discusses DQL as an approach to approximate solutions to DP problems. The major theme is the derivation of optimal policies to maximize a given objective function through taking a sequence of actions and updating the optimal policy iteratively. DQL is illustrated based on the `CartPole` game from the `gymnasium` Python package.
- Chapter 3 develops a first finance environment that allows the DQL agent from Chapter 2 to learn a financial prediction game. Although the environment formally replicates the application programming interface (API) of the `CartPole`, it misses yet some important characteristics to apply RL successfully.

Chapter 1. Learning through Interaction

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *ccollins@oreilly.com*.

> *The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning.*
>
> —Sutton and Barto (2018)

For human beings and animals alike, *learning* is almost as fundamental as breathing. It is something that happens continuously and most often unconsciously. There are different forms of learning. The one most important to the topics covered in this book is based on *interacting with an environment*.

Interaction with an environment provides the learner — or *agent* henceforth — with feedback that can be used to update their knowledge or to refine a skill. In this book, we are mostly interested in learning quantifiable facts about an environment, such as the odds of winning a bet or the reward that an action yields.

"Bayesian Learning" discusses Bayesian learning as an example of learning through interaction. "Reinforcement Learning" presents breakthroughs in artificial intelligence that were made possible through reinforcement learning. It also describes the major building blocks of reinforcement learning. "Deep Q-Learning" explains the two major characteristics of deep Q-learning which is the most important algorithm for the remainder of the book.

Bayesian Learning

Two simple examples can illustrate learning by interacting with an environment: tossing a biased coin and rolling a biased die. The examples are based on the idea that an agent betting repeatedly on the outcome of a biased gamble — and remembering all outcomes — can learn bet-by-bet about a gamble's bias and therewith about the optimal policy for betting. The idea, in that sense, makes use of Bayesian updating. Bayes theorem and Bayesian updating date back to the 18th century (see Bayes and Price (1763)). A modern and Python-based discussion of Bayesian statistics is found in Downey (2021).

Tossing a Biased Coin

Assume the simple game of betting on the outcome of tossing a biased coin. As a benchmark, consider the special case of an unbiased coin first. Agents are allowed to bet for free on the outcome of the coin tosses. An agent might, for example, bet randomly on either heads or tails. If the agent wins, the reward is 1 USD and nothing otherwise. The agent's goal is to maximize the total reward. The following Python code simulates several sequences of 100 bets each:

```
In [1]: import numpy as np
        from numpy.random import default_rng
        rng = default_rng(seed=100)
```

```
          rng = default_rng(seed=100)

In [2]: ssp = [1, 0]  ❶

In [3]: asp = [1, 0]  ❷

In [4]: def epoch():
            tr = 0
            for _ in range(100):
                a = rng.choice(asp)  ❸
                s = rng.choice(ssp)  ❹
                if a == s:
                    tr += 1  ❺
            return tr

In [5]: rl = np.array([epoch() for _ in range(250

        rl[:10]
Out[5]: array([56, 47, 48, 55, 55, 51, 54, 43, 55

In [6]: rl.mean()  ❼
Out[6]: 49.968
```

❶ The state space, `1` for heads and `0` for tails.

❷ The action space, `1` for a bet on heads and `0` for one on tails.

❸ The random bet.

❹ The random coin toss.

❺ The reward for a winning bet.

❻ The simulation of multiple sequences of bets.

❼ The average total reward.

The average total reward in this benchmark case is close to 50. The same result might be achieved by solely betting on either heads or tails.

Assume now that the coin is biased in a way that heads prevails in 80% of the coin tosses. Betting solely on heads would yield an average total reward of about 80 for 100 bets. Betting solely on tails would yield an average total reward of about 20. But what about the random betting strategy? The following Python code simulates this case:

```
In [7]: ssp = [1, 1, 1, 1, 0]  ❶

In [8]: asp = [1, 0]  ❷

In [9]: def epoch():
            tr = 0
```

```
            for _ in range(100):
                a = rng.choice(asp)
                s = rng.choice(ssp)
                if a == s:
                    tr += 1
            return tr

In [10]: rl = np.array([epoch() for _ in range(25
         rl[:10]
Out[10]: array([53, 56, 40, 55, 53, 49, 43, 45, 5

In [11]: rl.mean()
Out[11]: 49.924
```

❶ The biased state space.

❷ The same action space as before.

Although the coin is now highly biased the average total reward of the random betting strategy is about the same as in the benchmark case. This might sound counterintuitive. However, the expected win rate is given by $0.8 \cdot 0.5 + 0.2 \cdot 0.5 = 0.5$. In words, when betting on heads the win rate is 80% and when betting on tails it is 20%. Together, the total reward is as before on average. As a consequence, without learning, the agent is not able to capitalize on the bias.

A learning agent on the other hand can gain an edge by basing the betting strategy on the previous outcomes they observe. To this end, it is already enough to record all observed outcomes and to choose randomly from the set of all previous outcomes. In this case, the bias is reflected in the number of times the agent randomly bets on heads as compared to tails. The Python code that follows illustrates this simple learning strategy:

```
In [12]: ssp = [1, 1, 1, 1, 0]

In [13]: def epoch(n):
             tr = 0
             asp = [0, 1]  ❶
             for _ in range(n):
                 a = rng.choice(asp)
                 s = rng.choice(ssp)
                 if a == s:
                     tr += 1
                 asp.append(s)  ❷
             return tr

In [14]: rl = np.array([epoch(100) for _ in range
         rl[:10]
Out[14]: array([71, 65, 67, 69, 68, 72, 68, 68,

In [15]: rl.mean()
Out[15]: 66.78
```

1. The initial action space.
2. The update of the action space with the observed outcome.

With remembering and learning, the agent achieves an average total reward of about 66.8 — a significant improvement over the random strategy without learning. This is close to the expected value of $\left(0.8^2 + 0.2^2\right) \cdot 100 = 68$.

This strategy, while not optimal, is regularly observed in experiments involving human beings — and maybe somewhat surprisingly in animals as well. It is called *probability matching*.

PROBABILITY MATCHING

Koehler and James (2014) report results from studies analyzing probability matching, utility maximization, and other types of decision strategies.[1] The studies include a total of 1,557 university students.[2] The researchers find that probability matching is the most frequent strategy chosen or a close second to the utility maximizing strategy.

The researchers also find that the utility maximizing strategy is chosen in general by the "most cognitively able participants". They approximate cognitive ability through Scholastic Aptitude Test (SAT) scores, Mathematics Experience Composite scores, and the Number of University Statistics courses taken.

As is often the case in decision-making, human beings might need formal training and experience to overcome urges and behaviors that feel natural to achieve optimal results.

On the other hand, the agent can do better by simply betting on the most likely outcome as derived from past results. The following Python code implements this strategy.

```
In [16]: from collections import Counter


In [17]: ssp = [1, 1, 1, 1, 0]

In [18]: def epoch(n):
             tr = 0
             asp = [0, 1]  ❶
             for _ in range(n):
```

```
                    c = Counter(asp)  ❷
                    a = c.most_common()[0][0]  ❸
                    s = rng.choice(ssp)
                    if a == s:
                        tr += 1
                    asp.append(s)  ❹
                return tr

In [19]: rl = np.array([epoch(100) for _ in range
         rl[:10]
Out[19]: array([81, 70, 74, 77, 82, 74, 81, 80, 7

In [20]: rl.mean()
Out[20]: 78.828
```

- ❶ The initial action space.
- ❷ The frequencies of the action space elements.
- ❸ The action is chosen with the highest frequency.
- ❹ The update of the action space with the observed outcome.

In this case, the gambler achieves an average total reward of 78.5 which is close to the theoretical optimum of 80. In this context, this strategy seems to be the optimal one.

Rolling a Biased Die

As another example, consider a biased die. For this die, the probability for the outcome “4” shall be five times as likely as for any other number of the six-sided die. The following Python code simulates sequences of 600 bets on the outcome of the die, where a correct bet is rewarded with 1 USD and nothing otherwise.

```
In [21]: ssp = [1, 2, 3, 4, 4, 4, 4, 4, 5, 6]  ❶

In [22]: asp = [1, 2, 3, 4, 5, 6]  ❷

In [23]: def epoch():
             tr = 0
             for _ in range(600):
                 a = rng.choice(asp)
                 s = rng.choice(ssp)
                 if a == s:
                     tr += 1
             return tr

In [24]: rl = np.array([epoch() for _ in range(25
         rl[:10]
Out[24]: array([ 92,  96, 106,  99,  96, 107, 101

In [25]: rl.mean()
```

```
Out[25]: 101.22
```

❶ The biased state space.

❷ The uninformed action space.

Without learning, the random betting strategy yields an average total reward of about 100. With perfect information about the biased die, the agent could expect an average total reward of about 300 because it would win about 50% of the 600 bets.

With probability matching, the agent will not achieve a perfect outcome — as was the case with the biased coin. However, the agent can improve the average total reward by more than 75% as the following Python code shows:

```
In [26]: def epoch():
             tr = 0
             asp = [1, 2, 3, 4, 5, 6]  ❶
             for _ in range(600):
                 a = rng.choice(asp)
                 s = rng.choice(ssp)
                 if a == s:
                     tr += 1
                 asp.append(s)  ❷
             return tr
```

```
In [27]: rl = np.array([epoch() for _ in range(25
         rl[:10]
Out[27]: array([182, 174, 162, 157, 184, 167, 190

In [28]: rl.mean()
Out[28]: 176.296
```

❶ The initial action space.

❷ The update of the action space.

The average total reward increases to about 177, which is not that far from the expected value of that strategy of $\left(0.5^2 + 0.1^2 \cdot 5\right) \cdot 600 = 180$.

As with the biased coin tossing game, the agent again can do better by simply choosing the action with the highest frequency in the updated action space, as the following Python code confirms. The average total reward of 299 is pretty close to the theoretical maximum of 300:

```
In [29]: def epoch():
             tr = 0
             asp = [1, 2, 3, 4, 5, 6]  ❶
             for _ in range(600):
                 c = Counter(asp)  ❷
```

```
                    a = c.most_common()[0][0]  ❸
                    s = rng.choice(ssp)
                    if a == s:
                        tr += 1
                    asp.append(s)  ❹
                return tr

In [30]: rl = np.array([epoch() for _ in range(25
         rl[:10]
Out[30]: array([305, 288, 312, 306, 318, 302, 304

In [31]: rl.mean()
Out[31]: 297.204
```

❶ The initial action space.

❷ The frequencies of the action space elements.

❸ The action is chosen with the highest frequency.

❹ The update of the action space with the observed outcome.

Bayesian Updating

The Python code and simulation approach in the previous sub-sections make for a simple way to implement the learning of an agent through playing a potentially biased game. In other words, by interacting with the betting environment, the agent can update their estimates for the relevant probabilities.

The procedure can therefore be interpreted as *Bayesian updating* of probabilities — to find out, for example, the bias of a coin.[3] The following discussion illustrates this insight based on the coin-tossing game.

Assume that the probability for heads (`h`) is $P(h) = \alpha$ and that the probability for tails (`t`) accordingly is $P(t) = 1 - \alpha$. The coin flips are assumed to be identically and independently distributed (i.i.d) according to the binomial distribution. Assume that an experiment yields f_h times heads and f_t times tails. In that case, we get $P\left(E|\alpha\right) = \alpha^{f_h} \cdot (1-\alpha)^{f_t}$ as the probability that the experiment yields the assumed observations. E represents the event that f_h times heads and f_t times tails is observed.

One approach to deriving an appropriate value for α given the results from the experiment is *maximum likelihood estimation* (MLE). The goal of MLE is to find a value α that maximizes $P(E|\alpha)$. The problem to solve as follows:

$$
\begin{aligned}
\alpha^{MLE} &= \underset{\alpha}{\operatorname{argmax}} P\left(E|\alpha\right) \\
&= \underset{\alpha}{\operatorname{argmax}} \ln P\left(E|\alpha\right) \\
&= \underset{\alpha}{\operatorname{argmax}} \ln \left(\alpha^{f_h} \cdot (1-\alpha)^{f_t}\right) \\
&= \underset{\alpha}{\operatorname{argmax}} f_h \ln \alpha + f_t \ln \left(1-\alpha\right)
\end{aligned}
$$

With this, one derives the optimal estimator by taking the first derivative with respect to α and setting it equal to zero:

$$
\begin{aligned}
\tfrac{d}{d\alpha} P\left(E|\alpha\right) &= 0 \\
f_h \tfrac{d}{d\alpha} \ln \alpha + f_t \tfrac{d}{d\alpha} \ln \left(1-\alpha\right) &= 0 \\
\tfrac{f_h}{\alpha} - \tfrac{f_t}{1-\alpha} &= 0
\end{aligned}
$$

Simple manipulations yield the following maximum likelihood estimator:

$$
\alpha^{MLE} = \frac{f_h}{f_h + f_t}
$$

α^{MLE} is the frequency of heads over the total number of flips in the experiment. This is what has been learned flip-by-flip through the simulation approach, that is, through an agent betting on the outcomes of coin flips one after the other and remembering previous outcomes.

In other words, the agent has implemented Bayesian updating incrementally and bet-by-bet to arrive, after enough bets, to a

numerical estimator $\widehat{\alpha}$ close to α^{MLE}, that is, $\widehat{\alpha} \approx \alpha^{MLE}$.

Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning (ML) algorithm that relies on the interaction of an agent with an environment. This aspect is similar to the agent playing a potentially biased game and learning about relevant probabilities. However, RL algorithms are more general and capable in that an agent can learn from high-dimensional input to accomplish complex tasks.

While the mode of learning, *interaction* or *trial and error*, differs from other ML methods, the goals are nevertheless the same. Mitchell (1997) defines ML as follows:

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*
>
> —Mitchell (1997)

This section provides some general background to RL while the next chapter introduces more technical details. Sutton and

Barto (2018) provide a comprehensive overview of RL approaches and algorithms. On a high level, they describe RL as follows:

> *Reinforcement learning is about learning from interaction how to behave in order to achieve a goal. The reinforcement learning agent and its environment interact over a sequence of discrete time steps.*

REINFORCEMENT LEARNING

Most books on ML focus on supervised and unsupervised learning algorithms, but RL is the learning approach that comes closest to how human beings and animals learn. Namely, through repeated interaction with their environment and receiving positive (reinforcing) or negative (punishing) feedback. Such a sequential approach is much closer to human learning than the simultaneous learning from a generally very large number of labeled or unlabeled examples.

Major Breakthroughs

In artificial intelligence (AI) research and practice, two types of algorithms have seen a meteoric rise over the last ten years: *deep neural networks* (DNNs) and *reinforcement learning* (RL).[4] While DNNs have seen their own success stories in many

different application areas, they also play an integral role in modern RL algorithms, such as *Q-learning*.[5]

The book by Gerrish (2018) recounts several major success stories — and sometimes also failures — of AI over recent decades. In almost all of them, DNNs play a central role and RL algorithms sometimes are also a core part of the story. Among those successes are playing Atari 2600 games, chess, and Go on superhuman levels. These are discussed in what follows.

Concerning RL, and Q-learning in particular, the company DeepMind has achieved several noteworthy breakthroughs. In Mnih et al. (2013) and Mnih et al. (2015), the company reports how a so-called deep Q-learning (DQL) agent can learn to play Atari 2600 console[6] games on a superhuman level through interacting with a game-playing API. Bellemare et al. (2013) provide an overview of this popular API for the training of RL agents.

While mastering Atari games is impressive for an RL agent, and was celebrated by the AI researcher and retro gamer communities alike, the breakthrough concerning popular board games, such as Go and chess, gained the highest public attention and admiration.

In 2014, researcher and philosopher Nick Bostrom predicted in his popular book *Superintelligence* that it might take another 10 years for AI researchers to come up with an AI agent that plays the game of Go on a superhuman level:

> *Go-playing programs have been improving at a rate of about 1 dan/year in recent years. If this rate of improvement continues, they might beat the human world champion in about a decade.*

However, DeepMind researchers were able to successfully leverage the DQL techniques developed for playing Atari games and to come up with a DQL agent, called AlphaGo, that first beat the European champion in Go in 2015 and later in early 2016 even the world champion.[7] The details are documented in Silver et al. (2017). They summarize:

> *A long-standing goal of artificial intelligence is an algorithm that learns, tabula rasa, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play.*

DeepMind was able to generalize the approach of AlphaGo, which primarily relies on DQL agents playing a large number of games against themselves ("self-playing"), to the board games of chess and shogi. DeepMind calls this generalized agent AlphaZero. What is most impressive about AlphaZero is that only a few hours of self-playing chess for training are enough to reach not only a superhuman level but also a level well above any other computer engine, such as Stockfish. The paper by Silver et al. (2018) provides the details and summarizes:

> *In this paper, we generalize this approach into a single AlphaZero algorithm that can achieve superhuman performance in many challenging games. Starting from random play and given no domain knowledge except the game rules, AlphaZero convincingly defeated a world champion program in the games of chess and shogi (Japanese chess), as well as Go.*

The paper also provides the following training times:

> *Training lasted for approximately 9 hours in chess, 12 hours in shogi, and 13 days in Go …*

The dominance of AlphaZero over Stockfish in chess is not only remarkable given the short training time. It is also remarkable

because AlphaZero only evaluates a much lower number of positions per second:

> *AlphaZero searches just 60,000 positions per second in chess and shogi, compared with 60 million for Stockfish …*

One is inclined to attribute this to some form of acquired tactical and strategic intelligence on the side of AlphaZero as compared to predominantly brute force computation on the side of Stockfish.

REINFORCEMENT AND DEEP LEARNING

The breakthroughs in AI outlined in this sub-section rely on a combination of RL and DL. While DL can be applied in many scenarios, such as standard supervised and unsupervised learning situations, without RL, RL is in general today exclusively applied with the help of DL and DNNs.

Major Building Blocks

It is not that simple to exactly pin down why DQL algorithms are so successful in many domains that were obviously so hard to crack by computer scientists and AI researchers for decades. However, it is relatively straightforward to describe the major building blocks of an RL and DQL algorithm.

It generally starts with an *environment*. This can be an API to play Atari games, an environment to play chess, or an environment for navigating a map indoors or outdoors. Nowadays, there are many such environments available to get started with RL efficiently. One of the most popular ones is the Gymnasium environment.[8] On the Github page you read:

> *Gymnasium is an open source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API.*

At any given point, an environment is characterized by a *state*. The state summarizes all the relevant, and sometimes also irrelevant, information for an agent to receive as input when interacting with an environment. Concerning chess, a board position with all relevant pieces represents such a state. Sometimes additional input is required like, for example, whether castling has happened or not. For an Atari game, the pixels of the screen and the current score could represent the state of the environment.

The *agent* in this context subsumes all elements of the RL algorithm that interact with the environment and that learn

from these interactions. In an Atari games context, the agent might represent a player playing the game. In the context of chess, it can be either the player playing the white or black pieces.

An agent can choose one *action* from an often finite set of allowed actions. In an Atari game, movements to the left or right might be allowed actions. In chess, the rule set specifies both the number of allowed actions and their types.

Given the action of an agent, the state of the environment is updated. One such update is generally called a *step*. The concept of a step is general enough to encompass both heterogeneous and homogeneous time intervals between two steps. Whereas in Atari games, for example, real-time interaction with the game environment is simulated by rather short, homogeneous time intervals (“game clock”), chess players have quite some flexibility with regard to how long it takes them to make the next move (take the next action).

Depending on the action an agent chooses, a *reward* or *penalty* is awarded. For an Atari game, points are a typical reward. In chess, it is often a bit more subtle in that an evaluation of the current board position must take place. Improvements in the

evaluation then represent a reward while a worsening of the evaluation represents a penalty.

In RL, an agent is assumed to maximize an *objective function*. For Atari games, this can simply be the score achieved, that is, the sum of points collected during gameplay. In other words, it is a hunt for new "high scores". In playing chess, it is to set the opponent checkmate as represented by, say, an infinite evaluation score of the board position.

The *policy* defines which action an agent takes given a certain state of the environment. This is done by assigning values — technically, floating point numbers — to all possible combinations of states and actions. An optimal action is then chosen by looking up the highest value possible for the current state and the set of possible actions. Given a certain state in an Atari game, represented by all the pixels that make up the current scene, the policy might specify that the agent chooses "move right" as the optimal action. In chess, given a specific board position, the policy might specify to move the white king from c1 to b1.

An *episode* is a collection of steps from the initial state of the environment until success is achieved or failure is observed. In an Atari game, this means from the start of the game until the

agent has lost all “lives" — or has maybe achieved a final goal of the game. In chess, an episode represents a full game until a win, loss, or draw.

In summary, RL algorithms are characterized by the following building blocks:

- Environment
- State
- Agent
- Action
- Step
- Reward
- Objective
- Policy
- Episode

MODELING ENVIRONMENTS

The famous quote “Things should be as simple as possible, but no simpler”, usually attributed to Albert Einstein, can serve as a guideline for the design of environments and their APIs for reinforcement learning. Like in the context of a scientific model, an environment should capture all relevant aspects of the phenomena to be covered by it and dismiss those that are irrelevant. Sometimes, tremendous simplifications can be made based on this approach. At other times, an environment must represent the complete problem at hand. For example, when playing chess the complete board position with all the pieces is relevant.

Deep Q-Learning

What characterizes so-called deep Q-learning (DQL) algorithms? To begin with, QL is a special form of RL. In that sense, all the major building blocks of RL algorithms apply to QL algorithms as well. There are two specific characteristics of DQL algorithms.

First, DQL algorithms evaluate both the *immediate* reward of an agent's action and the *delayed* reward of the action. The delayed reward is estimated through an evaluation of the state that unfolds when the action is taken. The evaluation of the unfolding state is done under the assumption that all actions going forward are chosen optimally.

In chess, it is obvious that it is by far not sufficient to evaluate the very next move. It is rather necessary to look a few moves ahead and to evaluate different alternatives that can ensue. A chess novice has a hard time, in general, looking just two or three moves ahead. A chess grandmaster on the other hand can look as far as 20 to 30 moves ahead, as some argue.[9]

Second, DQL algorithms use DNNs to approximate, learn, and update the optimal policy. For most interesting environments in RL, the mapping of states and possible actions to values is too

complex to be modeled explicitly, say, through a table or a mathematical function. DNNs are known to have excellent approximation capabilities and provide all the flexibility needed to accommodate almost any type of state that an environment might communicate to the DQL agent.

Considering again chess as an example, it is estimated that there are more than 10^{100} possible moves, with illegal moves included. This compares to 10^{80} as an estimate for the number of atoms in the universe. With legal moves only, there are about 10^{40} possible moves, which is still a pretty large number:

```
In [32]: cm = 10 ** 40
         print(f'{cm:,}')
         10,000,000,000,000,000,000,000,000,000,000,000,0
```

This shows that only an *approximation* of the optimal policy is feasible in almost all interesting RL cases.

Conclusions

This chapter focuses on *learning through interaction* with an environment. It is a natural phenomenon, observed in human beings and animals alike. Simple examples show how an agent can learn probabilities through repeatedly betting on the outcome of a gamble and thereby implementing Bayesian updating. For this book, RL algorithms are the most important ones. Breakthroughs related to RL and the building blocks of RL are discussed. DQL, as a special RL algorithm, is characterized by taking into account not only immediate rewards but also delayed rewards from taking an action. In addition, the optimal policy is generally approximated by DNNs. Later chapters cover the DQL algorithm in much more detail and use it extensively.

References

Articles and books cited in this chapter:

- Bayes, Thomas and Richard Price (1763): "An Essay towards Solving a Problem in the Doctrine of Chances. By the Late Rev. Mr. Bayes, F.R.S. Communicated by Mr. Price, in a Letter to John Canton, A.M. F.R.S" *Philosophical Transactions of the Royal Society of London*, Vol. 53, 370–418.
- Bellemare, Marc et al. (2013): "The Arcade Learning Environment: An Evaluation Platform for General Agents."

Journal of Artificial Intelligence Research, Vol. 47, 253–279.

- Bostrom, Nick (2014): *Superintelligence—Paths, Dangers, Strategies*. Oxford University Press, Oxford.
- Downey, Allen (2021): *Think Bayes*. 2nd. ed., O'Reilly Media, Sebastopol.
- Gerrish, Sean (2018): *How Smart Machines Think*. MIT Press, Cambridge.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016): *Deep Learning*. MIT Press, Cambridge, http://deeplearningbook.org.
- Hanel, Paul, Katia Vione (2016): "Do Student Samples Provide an Accurate Estimate of the General Public?" PLoS ONE, Vol. 11, No. 12.
- Mitchell, Tom (1997): *Machine Learning*. McGraw-Hill, New York.
- Mnih, Volodymyr et al. (2013): "Playing Atari with Deep Reinforcement Learning" https://doi.org/10.48550/arXiv.1312.5602.
- Mnih, Volodymyr et al. (2015): "Human-Level Control through Deep Reinforcement Learning" *Nature*, Vol. 518, 529–533.
- Rachev, Svetlozar et al. (2008): *Bayesian Methods in Finance*. John Wiley & Sons, Hoboken.

- Silver, David et al. (2017): “Mastering the Game of Go without Human Knowledge.” *Nature*, Vol. 550, 354–359.
- Silver, David et al. (2018): “A General Reinforcement Learning Algorithm that Masters Chess, Shogi and Go through Self-Play”. *Science*, Vol. 362, Issue 6419, 1140-1144.
- Sutton, Richard and Andrew Barto (2018): *Reinforcement Learning: An Introduction*. 2nd ed., The MIT Press, Cambridge and London.
- Watkins, Christopher (1989): *Learning from Delayed Rewards*. Ph.D. thesis, University of Cambridge.
- Watkins, Christopher and Peter Dayan (1992): “Q-Learning.” *Machine Learning*, Vol. 8, 279-282.
- West, Richard and Keith Stanovich (2003): “Is Probability Matching Smart? Associations Between Probabilistic Choices and Cognitive Ability.” *Memory & Cognition*, Vol. 31, No. 2, 243-251.

Utility maximization is an economic principle that describes the process by which agents choose the best available option to achieve the highest level of satisfaction or utility given their preferences, constraints (such as income or budget), and the available alternatives.

Modern psychology is rather a discipline focused on university students in particular than on human beings in general. For example, Hanel and Vione (2016) conclude: “In summary, our results indicate that generalizing from students to the

general public can be problematic …, as students vary mostly randomly from the general public.”

For a comprehensive overview of Bayesian methods in finance see Rachev et al. (2008).

The book by Goodfellow et al. (2016) provides a comprehensive treatment of deep neural networks.

See, for example, the seminal works by Watkins (1989) and Watkins and Dayan (1992).

See the Wikipedia article for a detailed history of this console.

Public interest in this achievement is, for example, reflected in the more than 34 million views (as of November 2023) of the YouTube documentary about AlphaGo.

The Gymnasium project is a fork of the original Gym project by OpenAI whose support and maintenance have stopped.

This, of course, depends on the board position at hand. There are differences between opening, middle, and end games.

Chapter 2. Deep Q-Learning

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the second chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *ccollins@oreilly.com*.

> *Like a human, our agents learn for themselves to achieve successful strategies that lead to the greatest long-term rewards. This paradigm of learning by trial-and-error, solely from rewards or punishments, is known as reinforcement learning (RL).*[1]
>
> —DeepMind (2016)

The previous chapter introduces deep Q-learning (DQL) as a major algorithm in artificial intelligence (AI) to learn through interaction with an environment. This chapter provides some more details about the DQL algorithm. It uses the `CartPole` environment from the `gymnasium` Python package to illustrate the API-based interaction with gaming environments. It also implements a DQL agent as a self-contained Python class that serves as a blueprint for later DQL agents applied to financial environments.

However, before the focus is turned on DQL the chapter discusses general decision problems in economics and finance. Dynamic programming is introduced as a solution mechanism for dynamic decision problems. This provides the background for the application of DQL algorithms because they can be considered to lead to approximate solutions to dynamic programming problems.

"Decision Problems" classifies decision problems in economics and finance according to different characteristics. "Dynamic Programming" focuses on a special type of decision problem, so-called finite horizon Markovian dynamic programming problems. "Q-Learning" outlines the major elements of Q-learning and explains the role of deep neural networks in this context. Finally, "CartPole as an Example" illustrates a DQL

setup by the use of the `CartPole` game API and a DQL agent implemented as a Python class.

Decision Problems

In economics and finance, *optimization* and associated techniques play a central role. One could almost say that finance is nothing else than the systematic application of optimization techniques to problems arising in a financial context. Different types of optimization problems can be distinguished in finance. The major differentiating criteria as follows:

Discrete vs. continuous action space

: The quantities or actions to be chosen through optimization can be from a set of finite, discrete options (*optimal choice*) or from a set of infinite, continuous options (*optimal control*).

Static vs. dynamic problems

: Some problems are one-off optimization problems — these are generally called *static* problems. Other problems are characterized by a typically large number of sequential

and connected optimization problems over time — these are called *dynamic* problems.

Finite vs. infinite horizon

Dynamic optimization problems can have a *finite* or *infinite* horizon. Playing a game of chess generally has a finite horizon.[2] Estate planning for multiple generations of a family can be seen as a decision problem with an infinite horizon. Climate policy might be another one.

Discrete vs. continuous time

Some dynamic problems only require *discrete decisions* and optimizations at different points in time. Chess playing is again a good example. Other dynamic problems require *continuous decisions* and optimizations. Driving a car or flying an airplane are examples for when a driver or pilot needs to continuously make sure that appropriate decisions are made.

Given the examples discussed in Chapter 1, betting on the outcome of a biased coin is a static problem with discrete action space. Although such a bet can be repeated multiple times, the optimal betting strategy is independent of the previous bet as well as of the next bet. On the other hand, playing a game of chess is a dynamic problem — with a finite horizon — because a

player needs to make a sequence of optimal decisions that are all dependent on each other. The current board position depends on the player's (and the opponent's) previous moves. The future move options (action space) depend on the current move the player chooses.

In summary, because the action space is finite in both cases, coin betting is a discrete, static optimization problem, whereas playing chess is a discrete, dynamic optimization problem with finite horizon.

Dynamic Programming

An important type of dynamic optimization problem is the *finite horizon Markovian dynamic programming problem* (FHMDP). An FHMDP can formally be described by the following tuple[3]:

$$\left\{S, A, T, (r_t, f_t, \Phi_t)_{t=0}^{T}\right\}$$

S is the *state space* of the problem with generic element s. A is the *action space* of the problem with generic element a. T is a positive integer and represents the *finite horizon* of the problem.

For each point in time, at which an action is to be chosen, $t \in \{0, 1, \cdots, T\}$, there are two relevant functions and one relevant correspondence. The *reward function* maps a state and an action to a real-valued reward. If an agent at time t chooses action a_t in state s_t they receive a reward of r_t:

$$r_t : S \times A \to \mathbb{R}$$

The *transition function* maps a state and an action to another state. This function models the step from state s_t to state s_{t+1} when action a_t is taken:

$$f_t : S \times A \to S$$

Finally, the *feasible action correspondence* maps states to feasible actions. Given a state s_t, the correspondence defines all feasible actions $\left\{a_t^1, a_t^2, \cdots\right\}$ for that state:

$$\Phi_t : S \to P\left(A\right)$$

The objective of an agent is to choose a plan for taking actions at each point in time to maximize the sum of the per-period rewards over the horizon of the model. In other words, an agent needs to solve the following optimization problem:

$$\max_{a_t, t \in \{0,1,\cdots,T\}} \sum_{t=1}^{T} r_t\left(s_t, a_t\right)$$

subject to

$$\begin{array}{lcl} s_0 & = & s \in S \\ s_t & = & f_{t-1}\left(s_{t-1}, a_{t-1}\right), t = 1, \cdots, T \\ a_t & \in & \Phi_t\left(s_t\right), t = 1, \cdots, T \end{array}$$

What does *Markovian* mean in this context? It means that the transition function only depends on the current state and the current action taken and *not* on the full history of all states and actions. Formally, the following equality holds.

$$s_t = f_{t-1}\left(s_{t-1}, a_{t-1}\right) = f_{t-1}\left(s_{t-1}, s_{t-2}, \cdots; a_{t-1}, a_{t-2}, \cdots\right)$$

In this context, one also needs to distinguish between FHMDP problems for which the transition function is *deterministic* or *stochastic*. For chess, it is clear that the transition function is deterministic. On the other hand, typical computer games and all games offered in casinos generally have stochastic elements and as a consequence stochastic transition functions. If the transition function is stochastic, one usually speaks of *stochastic dynamic programming*.

A Markovian *policy* σ is a contingency plan that specifies which action a is to be taken if state s is observed. For an FHMDP, this implies $\sigma : S \to A$ with $\sigma_t\left(s_t\right) \in \Phi_t\left(s_t\right)$. This gives the set of all *feasible policies*, $\sigma \in \Sigma$.

The *total reward* of a feasible policy σ is denoted by

$$W\left(s_0, \sigma\right) = \sum_{t=1}^{T} r_t\left(s_t, \sigma_t\right)$$

The *value function* $V : S \rightarrow \mathbb{R}$ is then defined by the supremum of the total reward over all feasible policies:

$$V\left(s_0\right) = \sup_{\sigma \in \Sigma} W\left(s_0, \sigma\right)$$

For an optimal policy σ^*, the following must hold:

$$W\left(s_0, \sigma^*\right) = V\left(s_0\right), s_0 \in S$$

The problem of an agent faced with an FHMDP can therefore also be interpreted as finding an optimal policy with the above characteristics. If an optimal strategy σ^* exists, it can be shown that the value function, in general, satisfies the so-called *Bellmann equation*:

$$V_t\left(s_t\right) = \max_{a \in \Phi_t(s_t)} \left(r_t\left(s_t, a\right) + V_{t+1}(f_t\left(s_t, a\right))\right)$$

In other words, a dynamic decision problem involving simultaneous optimization over a combination of a potentially infinitely large number of feasible actions can be decomposed into a sequence of static, single-step optimization problems. Duffie (1988, 182), for example, summarizes:

> *In multi-period optimization problems, the problem of selecting actions over all periods can be decomposed into a family of single-period problems. In each period, one merely chooses an action maximizing the sum of the reward for that period and the value of beginning the problem again in the following period.*

In classical and modern economic and financial theory, a large number of FHMDP problems can be found, such as:

- Optimal growth over time,
- Optimal consumption-saving over time,
- Optimal portfolio allocation over time,
- Dynamic hedging of options and derivatives, or
- Optimal execution strategies in algorithmic trading.

Generally, these problems need to be modeled as FHMDP problems with *stochastic* transition functions. This is because most financial quantities, such as commodity prices, interest rates, stock prices, are uncertain and stochastic.

In particular, when dynamic programming involves continuous time modeling and stochastic transition functions — as is often the case in economics and finance — the mathematical requirements are pretty high. They involve, among other

things, analysis on metric spaces, measure-theoretic probability, and stochastic calculus. For an introduction to stochastic dynamic programming in Markovian financial models, refer to the book by Duffie (1988) for the discrete time case and Duffie (2001) for the continuous time case. For a comprehensive review of the required mathematical techniques in deterministic and stochastic dynamic programming and many economic examples, see the book by Stucharski (2009). The book by Sargent and Stucharski (2023) also covers dynamic programming and is accompanied by both Julia and Python code examples.

Q-Learning

Even with the most sophisticated mathematical techniques, many interesting FHMDPs in economics, finance, and other fields defy analytical solutions. In such cases, numerical methods that can approximate optimal solutions are usually the only feasible choice. Among these numerical methods is *Q-learning* (QL) as a major reinforcement learning (RL) technique (see also "Deep Q-Learning").

Watkins (1989) and Watkins and Dayan (1992) are pioneering works about modern QL. At the beginning of his Ph.D. thesis,

Watkins (1989) writes:

> *This thesis will present a general computational approach to learning from rewards and punishments, which may be applied to a wide range of situations in which animal learning has been studied, as well as to many other types of learning problems.*

In Watkins and Dayan (1992), the authors describe the algorithm as follows:

> *Q-learning (Watkins, 1989) is a form of model-free reinforcement learning. It can also be viewed as a method of asynchronous dynamic programming (DP). It provides agents with the capability of learning to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains. ...*
>
> *[A]n agent tries an action at a particular state, and evaluates its consequences in terms of the immediate reward or penalty it receives and its estimate of the value of the state to which it is taken. By trying all actions in all states repeatedly, it learns which are best overall, judged by long-term discounted reward. Q-learning is a primitive (Watkins, 1989) form of learning, but, as such, it can operate as the basis of far more sophisticated devices.*

Consider an FHMDP as in the previous section.

$$\left\{S, A, T, (r_t, f_t, \Phi_t)_{t=0}^{T}\right\}\Big]$$

In this context, the Q in QL stands for an action policy that assigns to each state $s_t \in S$ and feasible action $a_t \in A$ a numerical value. The numerical value is composed of the immediate reward of taking action a_t and the discounted

delayed reward —- given an optimal action $\overset{*}{a}_{t+1}$ taken in the subsequent state. Formally, this can be written as (note the resemblance with the reward function):

$$Q : S \times A \to \mathbb{R}$$

Then, with $\gamma \in (0, 1]$ being a discount factor, Q takes on the following functional form:

$$Q(s_t, a_t) = r_t(s_t, a_t) + \gamma \cdot \max_a Q(s_{t+1}, a)$$

In general, the optimal action policy Q cannot be specified in analytical form, that is, in the form of a table or mathematical function. Therefore, QL relies in general on approximate representations for the optimal policy Q.

If a deep neural network (DNN) is used for the representation, one usually speaks of *deep Q-learning* (DQL). To some extent, the use of DNNs in DQL might seem somewhat arbitrary. However, there are strong mathematical results — for example, the *universal approximation theorem* — that show the powerful approximation capabilities of DNNs. Wikipedia summarizes in this context:

> *In the mathematical theory of artificial neural networks, the universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions … The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters; however, it does not touch upon the algorithmic learnability of those parameters.*

As in RL in general, QL is based on an agent interacting with an environment and learning from the ensuing experiences through rewards and penalties. A QL agent takes actions based on two different principles:

Exploitation

: This refers to actions taken by the QL agent under the current optimal policy Q.

Exploration

: This refers to actions taken by a QL agent that are random. The purpose is to explore random actions and their associated values beyond what the current optimal policy would dictate.

Usually, the QL agent is supposed to follow an $\epsilon-$*greedy* strategy. In this regard, the parameter ϵ defines the ratio with which the agent relies on exploration as compared to exploitation. During the training of the QL agent, ϵ is generally assumed to decrease with an increasing number of training units.

In DQL, the policy Q — that is, the DNN — is regularly updated through what is called *replay*. For replay, the agent must store passed experiences (states, actions, rewards, next states, etc.) and use, in general relatively small, batches from the memorized experiences to re-train the DNN. In the limit — that is the idea and “hope" — the DNN approximates the optimal policy for the problem well enough. In most cases, an optimal policy is not achievable at all since the problem at hand is simply too complex — such as chess is with its 10^{40} possible moves.

DNNS FOR APPROXIMATION

The usage of DNNs in Q-learning agents is not arbitrary. The representation (approximation) of the optimal action policy Q generally is a demanding task. DNNs have powerful approximation capabilities which explains their regular usage as the “brain” for a Q-learning agent.

CartPole as an Example

The `gymnasium` package for Python provides several environments (APIs) that are suited for RL agents to be trained. `CartPole` is a relatively simple game that requires an agent to balance a pole on a cart by pushing the cart to the left or right. This section illustrates the API for the game, that is the environment, and shows how to implement a DQL agent in Python that can learn to play the game perfectly.

The Game Environment

The `gymnasium` package is installed as follows:

```
pip install gymnasium
```

Details about the `CartPole` game are found at *https://gymnasium.farama.org*. The first step is the creation of an environment object:

```
In [1]: import gymnasium as gym

In [2]: env = gym.make('CartPole-v1')
```

This object allows the interaction via simple method calls. For example, it allows us to see how many actions are feasible (*action space*), to sample random actions, or to get more information about the state description (*observation space*):

```
In [3]: env.action_space
Out[3]: Discrete(2)

In [4]: env.action_space.n  ❶
Out[4]: 2

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

In [6]: env.observation_space
Out[6]: Box([-4.8000002e+00 -3.4028235e+38 -4.188
         [4.8000002e+00 3.4028235e+38 4.1887903e-
         float32)

In [7]: env.observation_space.shape  ❷
Out[7]: (4,)
```

❶ Two actions, `0` and `1`, are possible.

❷ The state is described by four parameters.

The environment allows an agent to take one of two actions:

- `0`: Push the cart to the left.
- `1`: Push the cart to the right.

The environment models the state of the game through four physical parameters:

1. Cart position
2. Cart velocity
3. Pole angle
4. Pole angular velocity

Figure 2-1 shows a visual representation of a state of the `CartPole` game.

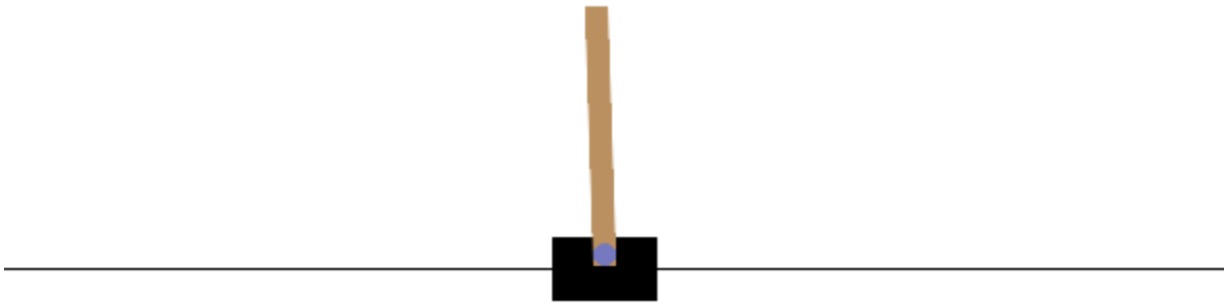

Figure 2-1. CartPole game

To play the game, the environment is first reset, leading by default to a randomized initial state. Every action steps the environment forward to the next state:

```
In [8]: env.reset(seed=100)  ❶
        # cart position, cart velocity, pole angl
Out[8]: (array([ 0.03349816,  0.0096554 , -0.021
         dtype=float32),
         {})

In [9]: env.step(0)  ❷
Out[9]: (array([ 0.03369127, -0.18515752, -0.022
```

```
            dtype=float32),
            1.0,
            False,
            False,
            {})

In [10]: env.step(1)  ❷
Out[10]: (array([ 0.02998812,  0.01027205, -0.01
            dtype=float32),
            1.0,
            False,
            False,
            {})
```

❶ Resets the environment, using a `seed` value for the random number generator.

❷ Steps the environment one step forward by taking one of two actions.

The returned tuple contains the following data:

- New state
- Reward
- Terminated
- Truncated

- Additional data

The game can be played until `True` is returned for "terminated". For every step, the agent receives a reward of 1. The more steps, the higher the total reward. The objective of an RL agent is to maximize the total reward or to achieve a minimum reward, for example.

A Random Agent

It is straightforward to implement an agent that only takes random actions. It cannot be expected that the agent will achieve a high total reward on average. However, every once in a while such an agent might be lucky.

The following Python code implements a random agent and collects the results from a larger number of games played:

```
In [11]: class RandomAgent:
             def __init__(self):
                 self.env = gym.make('CartPole-v1
             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_spac
                                state, reward, done, tru
                                if done:
                                    self.trewards.append
                                    break

In [12]: ra = RandomAgent()

In [13]: ra.play(15)

In [14]: ra.trewards
Out[14]: [18, 28, 17, 25, 16, 41, 21, 19, 22, 9,

In [15]: round(sum(ra.trewards) / len(ra.trewards
Out[15]: 18.67
```

❶ Average reward for the random agent.

The results illustrate that the random agent does not survive that long. The total reward might be somewhat above 20 or below. In rare cases, a relatively high total reward — for example, close to 50 — might be observed (called a *lucky punch*).

The DQL Agent

This sub-section implements a DQL agent in multiple steps. This allows for a more detailed discussion of the single elements that make up the agent. Such an approach seems justified because this DQL agent will serve as a blueprint for the DQL agent that will be applied to financial problems.

To get started, the following Python code first does all the required imports and customizes TensorFlow.

```
In [16]: 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 [17]: warnings.simplefilter('ignore')
         os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3
         os.environ['PYTHONHASHSEED'] = '0'

In [18]: from tensorflow.python.framework.ops imp
         disable_eager_execution()  ❶

In [19]: opt = keras.optimizers.legacy.Adam(learn
```

```
In [20]: random.seed(100)
         tf.random.set_seed(100)
```

- ❶ Speeds up the training of the neural network.
- ❷ Defines the optimizer to be used for the training.

The following Python code shows the initial part of the `DQLAgent` class. Among other things, it defines the major parameters and instantiates the DNN that is used for representing the optimal action policy.

```
In [21]: class DQLAgent:
             def __init__(self):
                 self.epsilon = 1.0  ❶
                 self.epsilon_decay = 0.9975  ❷
                 self.epsilon_min = 0.1  ❸
                 self.memory = deque(maxlen=2000)
                 self.batch_size = 32  ❺
                 self.gamma = 0.9  ❻
                 self.trewards = list()  ❼
                 self.max_treward = 0  ❽
                 self._create_model()  ❾
                 self.env = gym.make('CartPole-v1
             def _create_model(self):
```

```
                    self.model = Sequential()
                    self.model.add(Dense(24, activat
                    self.model.add(Dense(24, activat
                    self.model.add(Dense(2, activati
                    self.model.compile(loss='mse', c
```

❶ The initial ratio `epsilon` with which exploration is implemented.

❷ The factor by which `epsilon` is diminished.

❸ The minimum value for `epsilon`.

❹ The `deque` object which collects past experiences.[4]

❺ The number of experiences used for replay.

❻ The factor to discount future rewards.

❼ A `list` object to collect total rewards.

❽ A parameter to store the maximum total reward achieved.

❾ Initiates the instantiation of the DNN.

❿ Instantiates the `CartPole` environment.

The next part of the `DQLAgent` class implements the `.act()` and `.replay()` methods for choosing an action and updating the DNN (optimal action policy) given past experiences.

```
In [22]: class DQLAgent(DQLAgent):
             def act(self, state):
                 if random.random() < self.epsilo
                     return self.env.action_space
                 return np.argmax(self.model.pred
             def replay(self):
                 batch = random.sample(self.memor
                 for state, action, next_state, r
                     if not done:
                         reward += self.gamma * r
                             self.model.predict(n
                     target = self.model.predict(
                     target[0, action] = reward
                     self.model.fit(state, target
                 if self.epsilon > self.epsilon_m
                     self.epsilon *= self.epsilon
```

❽

- ❶ Chooses a random action.
- ❷ Chooses an action according to the (current) optimal policy.

❸ Randomly chooses a batch of past experiences for replay.

❹ Combines the immediate and discounted future reward.

❺ Generates the values for the state-action pairs.

❻ Updates the value for the relevant state-action pair.

❼ Trains/updates the DNN to account for the updated value.

❽ Reduces `epsilon` by the `epsilon_decay` factor.

The major elements are available to implement the core part of the `DQLAgent` class: the `.learn()` method which controls the interaction of the agent with the environment and the updating of the optimal policy. It also generates printed output to monitor the learning of the agent.

```
In [23]: class DQLAgent(DQLAgent):
             def learn(self, episodes):
                 for e in range(1, episodes + 1)
                     state, _ = self.env.reset()
                     state = np.reshape(state, [1
                     for f in range(1, 5000):
                         action = self.act(state)
                         next_state, reward, done
                         next_state = np.reshape(
```

```
                    self.memory.append(
                        [state, action, next
                    state = next_state  ❺
                    if done or trunc:
                        self.trewards.append
                        self.max_treward = m
                        templ = f'episode={e
                        templ += f' | max={s
                        print(templ, end='\r
                        break
                if len(self.memory) > self.b
                    self.replay()  ❽
            print()
```

❶ The environment is reset.

❷ The state object is reshaped.[5]

❸ An action is chosen according to the `.act()` method, given the current state.

❹ The relevant data points are collected for replay.

❺ The `state` variable is updated to the current state.

❻ Once terminated, the total reward is collected.

❼ The maximum total reward is updated if necessary.

❽ Replay is initiated as long as there are enough past experiences.

With the following Python code, the class is complete. It implements the `.test()` method that allows the testing of the agent without exploration.

```
In [24]: class DQLAgent(DQLAgent):
             def test(self, episodes):
                 for e in range(1, episodes + 1)
                     state, _ = self.env.reset()
                     state = np.reshape(state, [1
                     for f in range(1, 5001):
                         action = np.argmax(self
                         state, reward, done, tru
                         state = np.reshape(state
                         if done or trunc:
                             print(f, end=' ')
                             break
```

❶ For testing, only actions according to the optimal policy are chosen.

The DQL agent in the form of the completed `DQLAgent` Python class can interact with the `CartPole` environment to improve its capabilities in playing the game — as measured by the rewards achieved.

```
In [25]: agent = DQLAgent()

In [26]: %time agent.learn(1500)
         episode=1500 | treward= 224 | max= 500
         CPU times: user 1min 52s, sys: 21.7 s, t
         Wall time: 1min 46s

In [27]: agent.epsilon
Out[27]: 0.09997053357470892

In [28]: agent.test(15)
         500 373 326 500 348 303 500 330 392 304
```

At first glance, it is clear that the DQL agent consistently outperforms the random agent by a large margin. Therefore, luck can't be at work. On the other hand, without additional context, it is not clear whether the agent is a mediocre, good, or very good one.

In the documentation for the `CartPole` environment, you find that the threshold for total rewards is 475. This means that

everything above 475 is considered to be good. By default, the environment is truncated at 500 meaning that reaching that level is considered to be a “success” for the game. However, the game can be played beyond 500 steps/rewards which might make the training of the DQL agent more efficient.

Q-Learning vs. Supervised Learning

At the core of DQL is a DNN that resembles those often used and seen in supervised learning. Against this background, what are the major differences between these two approaches in machine learning?

To get started, the *objective* of both approaches is different. Concerning DQL, the objective is to learn an *optimal action policy* that maximizes total reward (or minimizes total penalties, for example). On the other hand, supervised learning aims at learning a *mapping* between features and labels.

Secondly, in DQL the *data* is generated through interaction and in a *sequential fashion*. The sequence of the data in general matters, like the sequence of moves in chess matter. In supervised learning, the data set is generally given upfront in the form of (expert-)labeled data sets and the sequence often

does not matter at all. Supervised learning, in that sense, is based on a *given set of correct examples* while DQL needs to generate appropriate data sets through interaction step-by-step.

Thirdly, in DQL *feedback generally comes delayed* given an action taken now. A DQL agent playing a game might not know until many steps later whether a current action is reward maximizing or not. The algorithm, however, makes sure that delayed feedback backpropagates in time through replay and updating of the DNN. In supervised learning, all relevant examples exist upfront and *immediate feedback is available* as to whether the algorithm gets the mapping between features and labels correct or not.

In summary, while DNNs might be at the core of both DQL and supervised learning they differ in fundamental ways in terms of their objective, the data they use, and the feedback their learning is based on.

Conclusions

Decision problems in economics and finance are manifold. One of the most important types is dynamic programming. This chapter classifies decision problems along the lines of different

binary characteristics (such as discrete or continuous action space) and introduces dynamic programming as an important algorithm to solve dynamic decision problems in discrete time.

Deep Q-learning is formalized and illustrated based on a simple game — `CartPole` from the `gymnasium` Python environment. The major goals in this regard are to illustrate the API-based interaction with an environment suited for RL and the implementation of a DQL agent in the form of a self-contained Python class.

The next chapter develops a simple financial environment that mimics the behavior of the `CartPole` environment so that the DQL agent from this chapter can learn to play a financial prediction game.

References

Articles and books cited in this chapter:

- Duffie, Darrell (1988): *Security Markets: Stochastic Models*. Academic Press, Boston.
- Duffie, Darrell (2001): *Dynamic Asset Pricing Theory*. 3rd ed., Princeton University Press, Princeton.

- Li, Yuxi (2018): “Deep Reinforcement Learning: An Overview.” *https://doi.org/10.48550/arXiv.1701.07274*.
- Sargent, Thomas and John Stucharski (2023): *Dynamic Programming*. *https://dp.quantecon.org/*.
- Stucharski, John (2009): *Economic Dynamics—Theory and Computation*. MIT Press, Cambridge and London.
- Sundaram, Rangarajan (1996): *A First Course in Optimization Theory*. Cambridge University Press, Cambridge.
- Watkins, Christopher (1989): *Learning from Delayed Rewards*. Ph.D. thesis, University of Cambridge.
- Watkins, Christopher and Peter Dayan (1992): “Q-Learning.” *Machine Learning*, Vol. 8, 279-282.

See the blog post by DeepMind.

The repetition rule, for example, prevents the possibility of an infinite chess game. A player can claim a draw if a certain position is repeated at least three times.

The exposition follows roughly Sundaram (1996, ch. 11).

`deque` objects are similar to `list` objects but have a maximum number of elements only. Once it is reached and a new element is added, the first element is dropped. In that sense, it implements a “first in, first out” queue. In the context of modeling the memory of a DQL agent, the `deque` object mimics a human brain that remembers more recent experiences better than older ones. The approach also

prevents the usage of old experiences, having been made based on a probably worse policy, to be used for replay.

This is a technical requirement of TensorFlow when updating DNNs based on a single sample only.

Chapter 3. Financial Q-Learning

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the third chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *ccollins@oreilly.com*.

> *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 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 this finance environment.

This chapter is quite 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" develops step-by-step the `Finance` class that mimics the behavior of the `CartPole` class. "DQL Agent" slightly adjusts the `DQLAgent` class from "CartPole as an Example". 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" 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 idea for the finance environment to be implemented in the following is that of 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 before. 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/rl4
            def __init__(self, symbol, feature, n
                self.symbol = symbol  ❷
                self.feature = feature  ❸
                self.n_features = n_features  ❹
                self.action_space = ActionSpace()
                self.min_accuracy = min_accuracy
                self._get_data()  ❼
                self._prepare_data()  ❽
            def _get_data(self):
                self.raw = pd.read_csv(self.url,
                               index_col=0, parse_dates=
❼
```

❶ The URL for the data set to be used (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 DNNs. From the implementation, it is obvious that the financial time series data is used in a static, non-random 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.data['r'] = np.log(self.data
                self.data['d'] = np.where(self.da
                self.data.dropna(inplace=True)
                self.data_ = (self.data - self.da
            def reset(self):
```

```
                    self.bar = self.n_features  ❻

                    self.treward = 0  ❼
                    state = self.data_[self.feature]
                        self.bar - self.n_features:se
                    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 lines 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 that 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
                    correct = True
                else:
                    correct = False
                reward = 1 if correct else 0  ❷
                self.treward += reward  ❸
                self.bar += 1  ❹
                self.accuracy = self.treward / (s
                if self.bar >= len(self.data):  ❻
                    done = True
                elif reward == 1:  ❼
                    done = False
                elif (self.accuracy < self.min_ac
                    done = True
```

```
                else:
                    done = False
                next_state = self.data_[self.feat
                    self.bar - self.n_features:se
                return next_state, reward, done,
```

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 was 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='EU

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, 2.69560
```

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

In [14]: fin.step(fin.action_space.sample())
Out[14]: (array([2.64643904, 2.69560062, 2.68085
          False, {})

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

In [16]: fin.reset()
         # four lagged, normalized log returns
Out[16]: (array([-1.19130476, -1.21344494,  0.61
```

❶ Specifies the feature type to be *normalized prices*.

❷ Shows the available symbols in the data set used.

❸ Specifies the feature type to be *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 slightly above 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_spac
                        state, reward, done, tru
                        if done:
                            self.trewards.append
                            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,

In [21]: round(sum(ra.trewards) / len(ra.trewards
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 (`DQLAgent` class from "The DQL Agent") 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 imp
         disable_eager_execution()

In [26]: opt = keras.optimizers.legacy.Adam(learn
```

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" with some minor adjustments for the context of this chapter.

```
In [27]: class DQLAgent:
             def __init__(self, symbol, feature,
                 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, featu
                                    min_accuracy, n_feat
             def _create_model(self):
                 self.model = Sequential()
```

```
        self.model = Sequential()
        self.model.add(Dense(24, activat
                             input_dim=s
        self.model.add(Dense(24, activat
        self.model.add(Dense(2, activati
        self.model.compile(loss='mse', o
    def act(self, state):
        if random.random() < self.epsilo
            return self.env.action_space
        return np.argmax(self.model.pred
    def replay(self):
        batch = random.sample(self.memor
        for state, action, next_state, r
            if not done:
                reward += self.gamma * r
                    self.model.predict(n
            target = self.model.predict(
            target[0, action] = reward
            self.model.fit(state, target
        if self.epsilon > self.epsilon_r
            self.epsilon *= self.epsilon
    def learn(self, episodes):

        for e in range(1, episodes + 1):
            state, _ = self.env.reset()
            state = np.reshape(state, [1
            for f in range(1, 5000):
                action = self.act(state)
                next_state, reward, done
                next_state = np.reshape(
```

```
                    next_state = np.reshape(
                    self.memory.append(
                        [state, action, next
                    state = next_state
                    if done:
                        self.trewards.append
                        self.max_treward = m
                        templ = f'episode={e
                        templ += f' | max={s
                        print(templ, end='\r
                        break
                if len(self.memory) > self.b
                    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
                for f in range(1, 5001):
                    action = np.argmax(self
                    state, reward, done, tru
                    state = np.reshape(state
                    if done:
                        print(f'total reward
                        break
            self.env.min_accuracy = ma
❷
```

```
```

- ❶ Defines the `Finance` environment object as a class attribute.
- ❷ Captures and resents 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, tot
         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
total reward=2603 | accuracy=0.525
```

1. Test results are all the same given the static data set.

Where the Analogy Fails

The `Finance` environment as introduced in "Finance Environment" 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 afterward 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 limits the training data in a severe fashion. Although the data set has more than 2,500 data points it is just *one* data set. The situation is as if an RL agent should learn to play chess based on a single historical game only, which it can go through over and over again. It is also comparable to a student who wants to prepare for an upcoming mathematics exam but only has one mathematics problem available to study and prepare with. 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 would still play 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, angular velocity — by pushing the cart to the left or right.

In *The Book of Why*, the authors explain that there are three layers based on which causal relationships can be learned and formulated. The first is data that can be observed, processed, and analyzed. For example, this might lead 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 in general. The second layer is about *interventions*. In the real world, you can in general expect that an action has some impact. Whether I exercise regularly or not, it should make a difference in the evolution of my weight and health, for example. This is comparable to the `CartPole` environment for which every action has a direct impact.

For 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 to this end) 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 his parents no matter whether they have an A or D grade in a mathematics exam. For a comprehensive discussion about the role 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 more simple 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.

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 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 to generate a basically unlimited number of financial time series.

References

Books cited in this chapter:

- Hilpisch, Yves (2020): *Artificial Intelligence for Finance: A Python-based Guide.* O'Reilly Media.
- Pearl, Judea and Dana Mackkenzie (2018): *The Book of Why: The New Science of Cause and Effect.* Penguin Science.
- Shanahan, Murray (2015): *The Technological Singularity*. MIT Press, Cambridge & London.
- Schneider, Susan (2012). *The Science of Consequences: How They Affect Genes, Change the Brain, and Impact Our World*. Prometheus Books, Amherst.

- For the definition of different types of AI, see, for example, Hilpisch (2020, ch. 2).

Part II. Data Augmentation

The second part of the book covers concepts and approaches to generate data for financial deep Q-learning:

- Chapter 4 implements data generation approaches based on Monte Carlo simulation. One approach is to add white noise to an existing financial time series. Another one is to simulate financial time series data based on a financial model (stochastic differential equation).
- Chapter 5 shows how to use generative adversarial networks (GANs) from artificial intelligence (AI), or more specifically from deep learning (DL), to generate financial time series data that is consistent with and statistically indistinguishable from the target financial time series. Such a target time series can be the historical price series for the share of a company stock (think Apple shares) or the historical foreign exchange quotes (think EUR/USD exchange rate).

Chapter 4. Simulated Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *ccollins@oreilly.com*.

> *It is often said that data is the new oil, but this analogy is not quite right. Oil is a finite resource that must be extracted and refined, whereas data is an infinite resource that is constantly being generated and refined.*
>
> —Halevy et al. (2009)

A major drawback of the financial environment as introduced in the previous chapter is that it relies by default on a single, historical financial time series. This is a too-limited data set to train a deep Q-learning (DQL) agent. It is like training an AI on a single game of chess and expecting it to perform well overall in chess.

This chapter introduces simulation-based approaches to augment the available data for the training of a DQL agent. The first approach, as introduced in "Noisy Time Series Data", is to add random noise to a static financial time series. Although it is commonly agreed upon that financial times data generally already contains noise — as compared to price movements or returns that are information-induced — the idea is to train the agent on a large number of "similar" time series in the hope that it learns to distinguish information from noise.

The second approach, discussed in "Simulated Time Series Data", is to generate financial time series data through simulation under certain constraints and assumptions. In general, a stochastic differential equation is assumed for the dynamics of the time series. The time series is then simulated given a discretization scheme and appropriate boundary conditions. This is one of the core numerical approaches used

in computational finance to price financial derivatives or to manage financial risks, for example.

Both data augmentation methods discussed in this chapter make it possible to generate an unlimited amount of training, validation, and test data for reinforcement learning.

Noisy Time Series Data

This section adjusts the first `Finance` environment from "Finance Environment" to add white noise, which is normally distributed data, to the original financial time series. First, the helper class for the action space:

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

The new `NoisyData` environment class only requires a few adjustments compared to the original `Finance` class. In the following Python code, two parameters are added to the initialization method:

```
In [2]: import numpy as np
        import pandas as pd
```

```
           from numpy.random import default_rng  ❶

In [3]: rng = default_rng(seed=100)  ❶

In [4]: class NoisyData:
            url = 'https://certificate.tpq.io/fi
            def __init__(self, symbol, feature, n
                         min_accuracy=0.485, nois
                         noise_std=0.001):
                self.symbol = symbol
                self.feature = feature
                self.n_features = n_features
                self.noise = noise  ❷
                self.noise_std = noise_std  ❸
                self.action_space = ActionSpace()
                self.min_accuracy = min_accuracy
                self._get_data()
                self._prepare_data()
            def _get_data(self):
                self.raw = pd.read_csv(self.url,
                       index_col=0, parse_dates=
```

❶ The random number generator is imported and initialized.

❷ The flag that specifies whether noise is added or not.

❸ The noise level to be used when adjusting the data; it is to be given in % of the price level.

The following part of the Python class code is the most important one. It is where the noise is added to the original time series data:

```
In [5]: class NoisyData(NoisyData):
            def _prepare_data(self):
                self.data = pd.DataFrame(self.raw
                if self.noise:
                    std = self.data.mean() * self
                    self.data[self.symbol] = (self
                        rng.normal(0, std, len(se
                self.data['r'] = np.log(self.data
                self.data['d'] = np.where(self.da
                self.data.dropna(inplace=True)
                ma, mi = self.data.max(), self.da
                self.data_ = (self.data - mi) / (
            def reset(self):
                if self.noise:
                    self._prepare_data()  ❹
                self.bar = self.n_features
                self.treward = 0
                state = self.data_[self.feature]
                    self.bar - self.n_features:se
                return state, {}
```

❶ The standard deviation for the noise is calculated in absolute terms.

❷ The white noise is added to the time series data.

❸ The features data is normalized through min-max scaling.

❹ A new noisy time series data set is generated.

INFORMATION VS. NOISE

Generally, it is assumed that financial time series data includes a certain amount of noise already. Investopedia defines noise as follows: “Noise refers to information or activity that confuses or misrepresents genuine underlying trends.” In this section, we take the historical price series as given and actively add noise to it. The idea is that a DQL agent learns about the fundamental price and or return trends embodied by the historical data set.

The final part of the Python class, the `.step()` method, can remain unchanged:

```
In [6]: class NoisyData(NoisyData):
            def step(self, action):
                if action == self.data['d'].iloc
                    correct = True
                else:
                    correct = False
```

```
                    reward = 1 if correct else 0
                    self.treward += reward
                    self.bar += 1
                    self.accuracy = self.treward / (s
                    if self.bar >= len(self.data):
                        done = True
                    elif reward == 1:
                        done = False
                    elif (self.accuracy < self.min_ac
                          self.bar > self.n_features
                        done = True
                    else:
                        done = False

                    next_state = self.data_[self.feat
                        self.bar - self.n_features:se
                    return next_state, reward, done,
```

Every time the financial environment is reset, a new time series is created by adding noise to the original time series. The following Python code illustrates this numerically:

```
In [7]: fin = NoisyData(symbol='EUR=', feature='E
                        noise=True, noise_std=0.0

In [8]: fin.reset()  ❶
Out[8]: (array([0.79295659, 0.81097879, 0.788409
```

```
In [9]: fin.reset()  ❶
Out[9]: (array([0.80642276, 0.77840938, 0.800963

In [10]: fin = NoisyData('EUR=', 'r', n_features=
                         noise=True, noise_std=0

In [11]: fin.reset()  ❷
Out[11]: (array([0.54198375, 0.30674865, 0.45688

In [12]: fin.reset()  ❷
Out[12]: (array([0.37967631, 0.40190291, 0.49196
```

❶ Different initial states for the normalized price data.

❷ Different initial states for the normalized returns data.

Finally, the following code visualizes several noisy time series data sets (see Figure 4-1):

```
In [13]: from pylab import plt, mpl
         plt.style.use('seaborn-v0_8')
         mpl.rcParams['figure.dpi'] = 300
         mpl.rcParams['savefig.dpi'] = 300
         mpl.rcParams['font.family'] = 'serif'

In [14]: import warnings
         warnings.simplefilter('ignore')
```

```
In [15]: for _ in range(5):
             fin.reset()
             fin.data[fin.symbol].loc['2022-7-1'
```

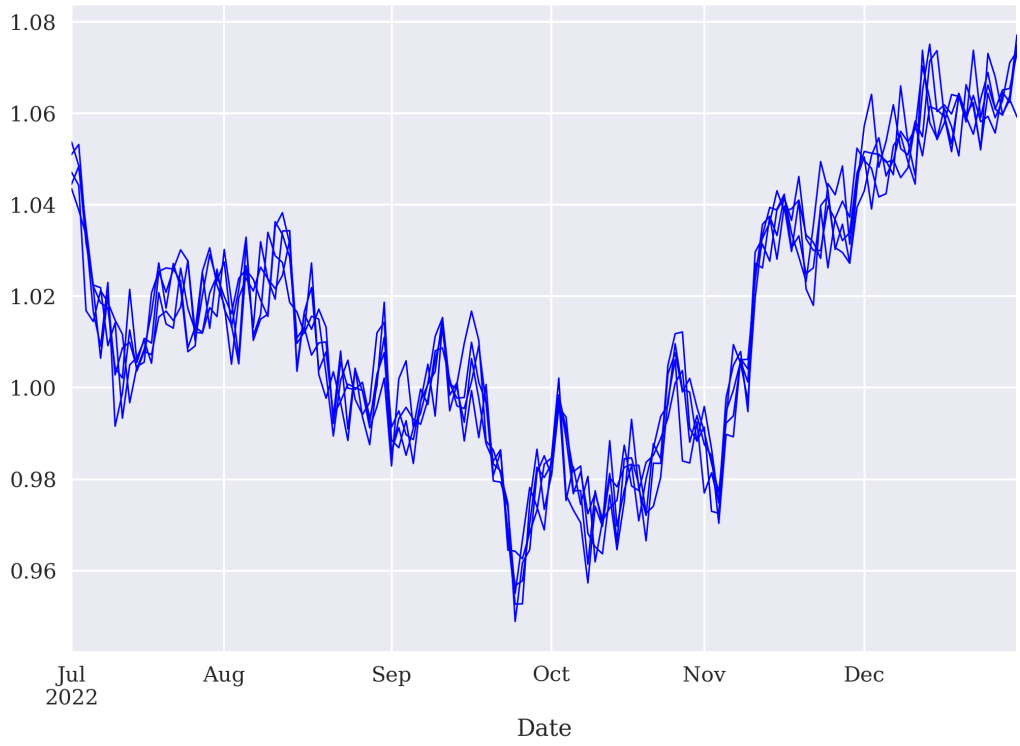

Figure 4-1. Noisy time series data for half a year

Using the new type of environment, the DQL agent—see the Python class in "`DQLAgent` Python Class"—can now be trained with a new, noisy data set for each episode. As the following Python code shows, the agent learns to distinguish between

information (original movements) and the noisy components quite well:

```
In [16]: %run dqlagent.py

In [17]: os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3

In [18]: agent = DQLAgent(fin.symbol, fin.feature

In [19]: %time agent.learn(250)
         episode= 250 | treward=   8.00 | max=14
         CPU times: user 27.3 s, sys: 3.92 s, tot
         Wall time: 26.9 s

In [20]: agent.test(5)
         total reward=2604 | accuracy=0.601
         total reward=2604 | accuracy=0.590
         total reward=2604 | accuracy=0.597
         total reward=2604 | accuracy=0.593
         total reward=2604 | accuracy=0.617
```

Simulated Time Series Data

In "Noisy Time Series Data", a historical financial time series is adjusted by adding white noise to it. In this section, the financial time series itself is simulated under suitable assumptions. Both approaches have in common that they allow the generation of an infinite number of different paths. However, using the *Monte Carlo simulation* (MCS) approach in this section leads to quite different paths in general that only on average show desired properties, such as a certain drift or a certain volatility.

In the following, a stochastic process according to Vasicek (1977) is simulated. Originally used to model the stochastic evolution of interest rates, it allows the simulation of trending or mean-reverting financial time series. The Vasicek model is described through the following stochastic differential equation[1]:

$$dx_t = \kappa\left(\theta - x_t\right)dt + \sigma x_t dZ_t$$

Such processes are also called *Ornstein-Uhlenbeck* (OU) processes. The variables and parameters have the following meaning: x_t is the process level at date *t*, κ is the mean-reversion factor, θ is the long-term mean of the process, and σ is the constant volatility parameter for Z_t which is a standard Brownian motion.

For the simulations, a simple Euler discretization scheme is used (with $s = t - \Delta t$ and z_t being standard normal):

$$x_t = x_s + \kappa (\theta - x_s) \Delta t + x_s \sigma \sqrt{\Delta t} z_t$$

The `Simulation` class implements a financial environment that relies on the simulation of the OU process above. The following Python code shows the initialization part of the class:

```
In [21]: class Simulation:
             def __init__(self, symbol, feature,
                          start, end, periods,
                          min_accuracy=0.525, x0=
                          kappa=1, theta=100, sig
                          normalize=True, new=Fal
                 self.symbol = symbol
                 self.feature = feature
                 self.n_features = n_features
                 self.start = start  ❶
                 self.end = end  ❷
                 self.periods = periods  ❸
                 self.x0 = x0  ❹
                 self.kappa = kappa  ❹
                 self.theta = theta  ❹
                 self.sigma = sigma  ❹
                 self.min_accuracy = min_accuracy
                 self.normalize = normalize  ❻
```

```
                    self.new = new  ❼
                    self.action_space = ActionSpace(
                    self._simulate_data()
                    self._prepare_data()
```

❶ The start date for the simulation.

❷ The end date for the simulation

❸ The number of periods to be simulated.

❹ The OU model parameters for the simulation.

❺ The minimum accuracy required to continue.

❻ The parameter indicating whether normalization is applied to the data or not.

❼ The parameter indicating whether a new simulation is initiated for every episode or not.

The following Python code shows the core method of the class. It implements the MCS for the OU process:

```
In [22]: import math
         class Simulation(Simulation):

             def _simulate_data(self):
```

```
                    index = pd.date_range(start=sel
                                end=self.end, perio
                    x = [self.x0]  ❶
                    dt = (index[-1] - index[0]).days
                    for t in range(1, len(index)):
                        x_ = (x[t - 1] + self.kappa
                              x[t - 1] * self.sigma
                        x.append(x_)  ❹

                    self.data = pd.DataFrame(x, colu
 ❺
```

❶ The initial value of the process (boundary condition).

❷ The length of the time interval, given the one-year horizon and the number of steps.

❸ The Euler discretization scheme for the simulation itself.

❹ The simulated value is appended to the `list` object.

❺ The simulated process is transformed into a `DataFrame` object.

Data preparation is taken care of by the following code:

```
In [23]: class Simulation(Simulation):
```

```
                def _prepare_data(self):
                    self.data['r'] = np.log(self.dat
                    self.data.dropna(inplace=True)
                    if self.normalize:
                        self.mu = self.data.mean()
                        self.std = self.data.std()
                        self.data_ = (self.data - se
                    else:
                        self.data_ = self.data.copy(
                    self.data['d'] = np.where(self.d
                    self.data['d'] = self.data['d']
❸
```

❶ Derives the log returns for the simulated process.

❷ Applies Gaussian normalization to the data.

❸ Derives the directional values from the log returns.

The following methods are helper methods and allow you, for example, to reset the environment:

```
In [24]: class Simulation(Simulation):
                def _get_state(self):
                    return self.data_[self.feature]
                                                self.n_
                def seed(self, seed):
```

```
                random.seed(seed)  ❷
                tf.random.set_seed(seed)  ❷
            def reset(self):
                self.treward = 0
                self.accuracy = 0
                self.bar = self.n_features
                if self.new:
                    self._simulate_data()
                    self._prepare_data()
                state = self._get_state()
                return state.values, {}
```

❶ Returns the current set of feature values.

❷ Fixes the seed for different random number generators.

The final method `.step()` is the same as for the `NoisyData` class:

```
In [25]: class Simulation(Simulation):
            def step(self, action):
                if action == self.data['d'].iloc
                    correct = True
                else:
                    correct = False
                reward = 1 if correct else 0
                self.treward += reward
```

```
                    self.bar += 1
                    self.accuracy = self.treward / (
                    if self.bar >= len(self.data):
                        done = True
                    elif reward == 1:
                        done = False
                    elif (self.accuracy < self.min_a
                        done = True
                    else:
                        done = False
                    next_state = self.data_[self.fea
                        self.bar - self.n_features:s
                    return next_state, reward, done,
```

With the complete `Simulation` class, different processes can be simulated. The next code snippet uses three different sets of parameters:

- *Baseline*: No volatility or trending (long-term mean > initial value)
- *Trend*: Volatility and trending (long-term mean > initial value)
- *Mean-reversion*: Volatility and mean-reverting (long-term mean = initial value)

Figure 4-2 shows the simulated processes graphically.

```
In [26]: sym = 'EUR='

In [27]: env_base = Simulation(sym, sym, 5, start
                               periods=252, x0=1, kapp
                               normalize=True)  ❶
         env_base.seed(100)

In [28]: env_trend = Simulation(sym, sym, 5, star
                               periods=252, x0=1, kapp
                               normalize=True)  ❷
         env_trend.seed(100)

In [29]: env_mrev = Simulation(sym, sym, 5, start
                               periods=252, x0=1, kapp
                               normalize=True)  ❸
         env_mrev.seed(100)

In [30]: env_mrev.data[sym].iloc[:3]
Out[30]: 2024-01-02 10:59:45.657370517    1.00423
         2024-01-03 21:59:31.314741035    1.00975
         2024-01-05 08:59:16.972111553    1.01101
         Name: EUR=, dtype: float64

In [31]: env_base.data[sym].plot(figsize=(10, 6),
         env_trend.data[sym].plot(label='trend',
         env_mrev.data[sym].plot(label='mean-reve
         plt.legend();
```

❶ The *baseline* case.

❷ The *trend* case.

❸ The *mean-reversion* case.

Figure 4-2. The simulated OU processes[2]

MODEL PARAMETER CHOICE

The Vasicek (1977) model provides a certain degree of flexibility to simulate stochastic processes with different characteristics. However, in practical applications the parameters would not be chosen arbitrarily but rather derived — through optimization methods — from market observed data. This procedure is generally called *model calibration* and has a long tradition in computational finance. See, for example, Hilpisch (2015) for more details.

By default, resetting the `Simulation` environment generates a new simulated OU process, as Figure 4-3 illustrates.

```
In [32]: sim = Simulation(sym, 'r', 4, start='202
                          periods=2 * 252, min_ac
                          kappa=2, theta=2, sigma
                          normalize=True, new=Tru
         sim.seed(100)

In [33]: for _ in range(10):
             sim.reset()
             sim.data[sym].plot(figsize=(10, 6),
```

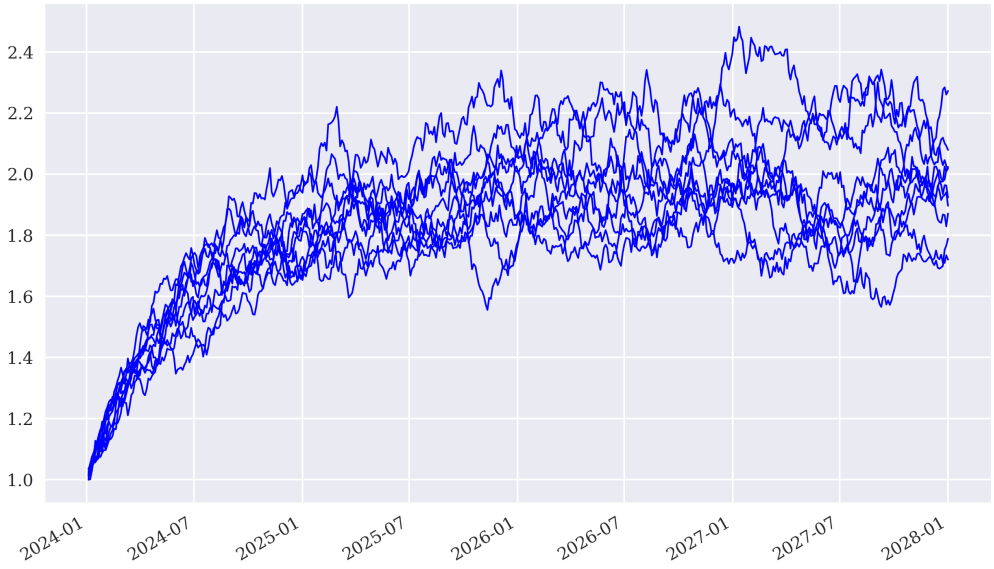

Figure 4-3. Multiple simulated, trending OU processes

The `DQLAgent` from "`DQLAgent` Python Class" works with this environment in the same way it worked with the `NoisyData` environment in the previous section. The following example uses the parametrization from before for the `Simulation` environment, which is a trending case. The agent learns quite well to predict the future directional movement:

```
In [34]: agent = DQLAgent(sim.symbol, sim.feature

In [35]: %time agent.learn(500)
         episode= 500 | treward= 265.00 | max= 28
         CPU times: user 42.1 s, sys: 5.87 s, tot
         Wall time: 40.1 s
```

```
In [36]: agent.test(5)
         total reward= 499 | accuracy=0.547
         total reward= 499 | accuracy=0.515
         total reward= 499 | accuracy=0.561
         total reward= 499 | accuracy=0.533
         total reward= 499 | accuracy=0.549
```

The next example assumes a mean-reverting case, in which the `DQLAgent` is not able to predict the future directional movements as well as before. It seems that learning a trend might be easier than learning from simulated mean-reverting processes:

```
In [37]: sim = Simulation(sym, 'r', 4, start='202
                          periods=2 * 252, min_ac
                          kappa=1.25, theta=1, si
                          normalize=True, new=Tru
         sim.seed(100)

In [38]: agent = DQLAgent(sim.symbol, sim.feature

In [39]: %time agent.learn(500)
         episode= 500 | treward=  12.00 | max=  1
         CPU times: user 17.8 s, sys: 2.66 s, tot
         Wall time: 16.3 s
```

```
In [40]: agent.test(5)
         total reward= 499 | accuracy=0.487
         total reward= 499 | accuracy=0.495
         total reward= 499 | accuracy=0.511
         total reward= 499 | accuracy=0.487
         total reward= 499 | accuracy=0.449
```

Conclusions

The addition of white noise to a historical financial time series allows, in principle, the generation of an unlimited number of data sets to train a DQL agent. By varying the degree of noise, i.e. the standard deviation, the adjusted time series data might be close to or very different from the original time series. For the DQL agent, it can therefore be made easier or more difficult to learn to distinguish information from the added noise.

Simulation approaches were introduced to finance long before the wide-spread adoption of computers in the industry. Boyle (1977) is considered the seminal article in this regard. Glasserman (2004) provides a comprehensive overview of MCS techniques for finance.

Using MCS for OU processes allows the simulation of trending and mean-reverting processes. Typical trending financial time series are stock index levels or individual stock prices. Typical mean-reverting financial time series are FX rates or commodity prices.

In this chapter, the parameters for the simulation are assumed "out-of-the-blue". In a more realistic setting, appropriate parameter values could be found, for example, through the calibration of the Vasicek (1977) model to the prices of liquidly traded options — an approach with a long tradition in computational finance.[3]

The examples in this chapter show that the `DQLAgent` can more easily learn about trending time series than about mean-reverting ones. The next chapter turns the attention on generative approaches for the creation of synthetic time series data based on neural networks.

References

Books and articles cited in this chapter:

- Boyle, Phelim (1977): "Options: A Monte Carlo Approach." *Journal of Financial Economics*, Vol. 4, No. 4, pp. 322–338.

- Glasserman, Paul (2004): *Monte Carlo Methods in Financial Engineering*. Springer, New York.
- Halevy, Alon, Peter Norvig, and Fernando Preira (2009): "The Unreasonable Effectiveness of Data." *IEEE Intelligent Systems*, March/April, 9-12.
- Hilpisch, Yves (2018) *Python for Finance—Mastering Data-Driven Finance.* 2nd ed., O'Reilly, Sebastopol et al.
- Hilpisch, Yves (2015): *Derivatives Analytics with Python*. Wiley Finance, Chichester.
- Vasicek, Oldrich (1977). "An equilibrium characterization of the term structure." *Journal of Financial Economics*, Vol. 5, No. 2, 177–188.

`DQLAgent` Python Class

The following Python code is from the `dqlagent.py` module and contains the `DQLAgent` class used in this chapter:

```
#
# Deep Q-Learning Agent
#
# (c) Dr. Yves J. Hilpisch
# Reinforcement Learning for Finance
```

```
#

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, Flatten
from keras.models import Sequential

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

from tensorflow.python.framework.ops import disab
disable_eager_execution()

opt = keras.optimizers.legacy.Adam


class DQLAgent:
    def __init__(self, symbol, feature, n_feature
        self.epsilon = 1.0
        self.epsilon_decay = 0.9975
        self.epsilon_min = 0.1
        self.memory = list()
        self.batch_size = 32
```

```
        self.gamma = 0.5
        self.trewards = deque(maxlen=2000)
        self.max_treward = -np.inf
        self.n_features = n_features
        self.env = env
        self.episodes = 0
        self._create_model(hu, lr)

    def _create_model(self, hu, lr):
        self.model = Sequential()
        self.model.add(Dense(hu, activation='relu
                             input_dim=self.n_fea
        self.model.add(Dense(hu, activation='relu
        self.model.add(Dense(2, activation='linea
        self.model.compile(loss='mse', optimizer=

    def _reshape(self, state):
        #if state.ndim == 1:
        #    return np.reshape(state, [1, self.n_
        #else:
        #    return np.array(state.flatten())
        state = state.flatten()
        return np.reshape(state, [1, len(state)])

    def act(self, state):
        if random.random() < self.epsilon:
            return self.env.action_space.sample()
        return np.argmax(self.model.predict(state
```

```
    def replay(self):
        batch = random.sample(self.memory, self.b
        for state, action, next_state, reward, do
            if not done:
                reward += self.gamma * np.amax(
                    self.model.predict(next_state
                target = self.model.predict(state
                target[0, action] = reward
                self.model.fit(state, target, epo
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def learn(self, episodes):
        for e in range(1, episodes + 1):
            self.episodes += 1
            state, _ = self.env.reset()
            state = self._reshape(state)
            treward = 0
            for f in range(1, 5000):
                action = self.act(state)
                next_state, reward, done, trunc,
                treward += reward
                next_state = self._reshape(next_s
                self.memory.append(
                    [state, action, next_state, r
                state = next_state
                if done:
```

```
                    self.trewards.append(treward)
                    self.max_treward = max(self.m
                    templ = f'episode={e:4d} | tr
                    templ += f' | max={self.max_t
                    print(templ, end='\r')
                    break
            if len(self.memory) > self.batch_size
                self.replay()
        print()

    def test(self, episodes, min_accuracy=0.0,
             min_performance=0.0, verbose=True,
             full=True):
        ma = self.env.min_accuracy
        self.env.min_accuracy = min_accuracy
        if hasattr(self.env, 'min_performance'):
            mp = self.env.min_performance
            self.env.min_performance = min_perfor
            self.performances = list()
        for e in range(1, episodes + 1):
            state, _ = self.env.reset()
            state = self._reshape(state)
            for f in range(1, 5001):
                action = np.argmax(self.model.pre
                state, reward, done, trunc, _ = s
                state = self._reshape(state)
                if done:
                    templ = f'total reward={f:4d}
```

```
                    templ += f'accuracy={self.env
                    if hasattr(self.env, 'min_per
                        self.performances.append(
                        templ += f' | performance
                    if verbose:
                        if full:
                            print(templ)
                        else:
                            print(templ, end='\r
                    break
            self.env.min_accuracy = ma
            if hasattr(self.env, 'min_performance'):
                self.env.min_performance = mp
            print()
```

For more details on Monte Carlo simulation with Python, see Chapter 12 of Hilpisch (2018).

The careful observer will notice that the three processes do not start exactly at one. This is due to the fact that the initial value gets "lost" after the calculation of the log returns and the cleaning up of the `DataFrame` object.

For details, numerical techniques, and Python code examples in the context of financial model calibration see Hilpisch (2015).

Chapter 5. Generated Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fifth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *ccollins@oreilly.com*.

> *In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution. The generative model can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model is analogous to the police, trying to detect the counterfeit currency. Competition in this game drives both teams to improve their methods until the counterfeits are indistinguishable from the genuine articles.*
>
> —Goodfellow et al. (2014)

In their seminal paper Goodfellow et al. (2014), the authors introduce *generative adversarial nets* (GANs) that rely on a so-called *generator* and *discriminator*. The generator is trained on a given data set. Its purpose is to generate data that is similar "in nature", that is, in a statistical sense, to the original data set. The discriminator is trained to distinguish between samples from the original data set and samples generated by the generator. The goal is to train the generator in a way that the discriminator cannot distinguish anymore between original samples and generated ones.

Although this approach might sound relatively simple at first, it has seen a large number of breakthrough applications after its publication. There are GANs available nowadays that create images, paintings, cartoons, text, poems, songs, computer code, and even videos that are hardly or even impossible to distinguish from human work. Between 2022 and 2024 alone, so many GANs have been published — open ones and commercial ones — that it is impossible to provide an exhaustive list.

GANs can also be used to create synthetic time series data that in turn can be used to train reinforcement learning agents. Similar to the noisy data and Monte Carlo simulation approaches of Chapter 4, GANs can generate a theoretically infinite set of synthetic time series.

The chapter proceeds as follows. "Simple Example" illustrates the training of a GAN based on data generated by a deterministic function. "Financial Example" then trains a GAN based on historical returns data of a financial instrument. The goal for the generator is to generate synthetic returns data that is, in the best case, indistinguishable for the discriminator from the real returns data. In addition, the Kolmogorow-Smirnow statistical test is applied to illustrate that synthetic returns data can also be indistinguishable from real data for traditional statistical tests.

Simple Example

This section deals with data generated by a deterministic mathematical function. First, some typical Python imports and configurations:

```
In [1]: import os
        import numpy as np
        import pandas as pd
        from pylab import plt, mpl

In [2]: import tensorflow as tf
        from tensorflow.keras.models import Seque
        from tensorflow.keras.layers import Dense
        from tensorflow.keras.optimizers import A
        from sklearn.preprocessing import Standar

In [3]: plt.style.use('seaborn-v0_8')
        mpl.rcParams['figure.dpi'] = 300

        mpl.rcParams['savefig.dpi'] = 300
        mpl.rcParams['font.family'] = 'serif'
        os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
```

Second, the original data is generated from a simple mathematical function and is normalized. Figure 5-1 shows the

two data sets as lines.

```
In [4]: x = np.linspace(-2, 2, 500)  ❶

In [5]: def f(x):
            return x ** 3  ❷

In [6]: y = f(x)  ❸

In [7]: scaler = StandardScaler()  ❹

In [8]: y_ = scaler.fit_transform(y.reshape(-1, 1

In [9]: plt.plot(x, y, 'r', lw=1.0, label='real d
        plt.plot(x, y_, 'b--', lw=1.0, label='nor
        plt.legend();
```

❶ Generates the input values of a given interval.

❷ Defines the mathematical function (cubic monomial).

❸ Generates the output values.

❹ Normalizes the data using Gaussian normalization.

Figure 5-1. Real data (solid line), normalized data (dashed line)

The following Python code creates the first component of the GAN: the *generator*. It is a simple, standard dense neural network (DNN) for estimation:

```
In [10]: def create_generator(hu=32):
             model = Sequential()
             model.add(Dense(hu, activation='relu
             model.add(Dense(hu, activation='relu
             model.add(Dense(1, activation='linea
             return model
```

The second component of the GAN is the *discriminator* which is created through the following Python function. The model is again a simple, standard DNN — but this time for binary classification:

```
In [11]: def create_discriminator(hu=32):
             model = Sequential()
             model.add(Dense(hu, activation='relu
             model.add(Dense(hu, activation='relu
             model.add(Dense(1, activation='sigmo
             model.compile(loss='binary_crossentr
                           metrics=['accuracy'])
             return model
```

The GAN is created by taking the generator and discriminator models as input arguments. For the GAN, the discriminator is set to “not trainable" — only the generator is trained with the GAN:

```
In [12]: def create_gan(generator, discriminator,
             discriminator.trainable = False  ❶
             model = Sequential()
             model.add(generator)  ❷
             model.add(discriminator)  ❸
             model.compile(loss='binary_crossentr
                           optimizer=Adam(learnin
```

```
                return model

In [13]: generator = create_generator()  ❹
         discriminator = create_discriminator()
         gan = create_gan(generator, discriminato
❹
```

❶ The discriminator model is not trained.

❷ The generator model is added first to the GAN.

❸ The discriminator model is added second to the GAN.

❹ The three models are created in sequence.

With the three models created, the training of the models can take place. The following Python code trains the models over many epochs and a randomly sampled batch of a given size per epoch:

```
In [14]: from numpy.random import default_rng

In [15]: rng = default_rng(seed=100)

In [16]: def train_models(y_, epochs, batch_size)
             for epoch in range(epochs):
                 # Generate synthetic data
```

```
                noise = rng.normal(0, 1, (batch_
                synthetic_data = generator.predi

                # Train discriminator
                real_data = y_[rng.integers(0,
                discriminator.train_on_batch(rea

                discriminator.train_on_batch(syn
                                             np

                # Train generator
                noise = rng.normal(0, 1, (batch_
                gan.train_on_batch(noise, np.one

                # Print progress
                if epoch % 1000 == 0:
                    print(f'Epoch: {epoch}')
            return real_data, synthetic_data

In [17]: %%time
         real_data, synthetic_data = train_models
         Epoch: 0
         Epoch: 1000
         Epoch: 2000
         Epoch: 3000
         Epoch: 4000
         Epoch: 5000
         CPU times: user 1min 47s, sys: 10.9 s, t
```

```
Wall time: 1min 49s
```

❶ Generates standard normally distributed noise ...

❷ ... as input for the generator to create synthetic data.

❸ Randomly samples data from the real data set.

❹ Trains the discriminator on the real data sample (labels are 1).

❺ Trains the discriminator on the synthetic data sample (labels are 0).

❻ Generates standard normally distributed noise ...

❼ ... as input for the training of the generator.

Figure 5-2 shows the last real data and synthetic data samples from the training. These are the data sets the discriminator is confronted with. It is difficult to tell, just by visual inspection, whether the data sets are sampled from the real data or not. That feature is actually what the generator is striving for:

```
In [18]: plt.plot(real_data, 'r', lw=1.0, label=
         plt.plot(synthetic_data, 'b:', lw=1.0,
         plt.legend();
```

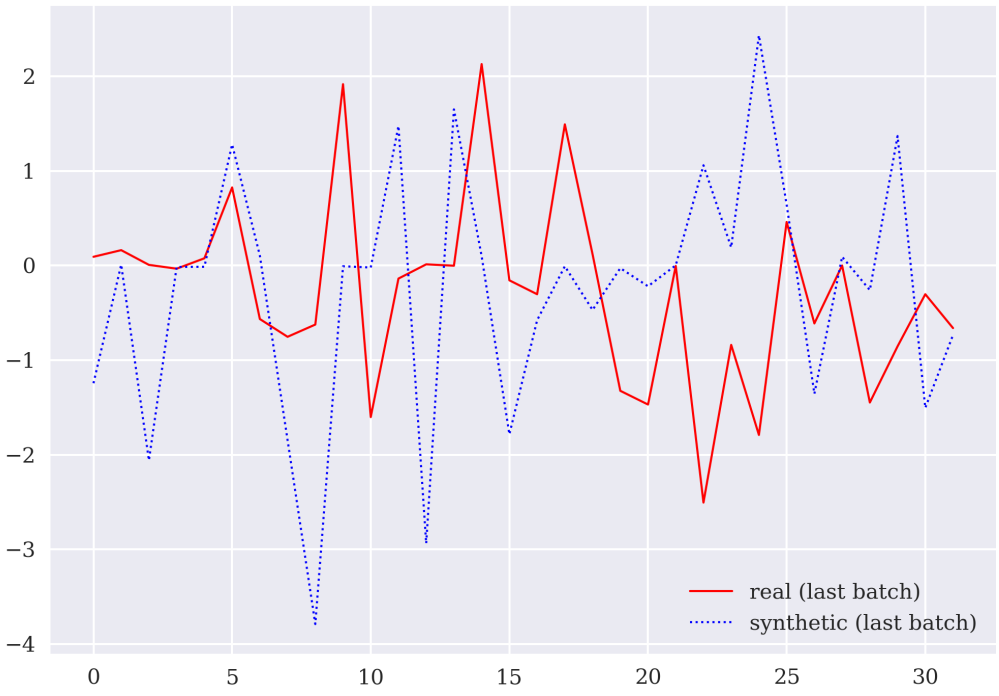

Figure 5-2. Normalized real and synthetic data sample

A more thorough analysis can shed more light on the statistical properties of the synthetic data sets generated by the GAN as compared to the real data from the mathematical function.

To this end, the following Python code generates several synthetic data sets of the length of the real data set. Several descriptive statistics, such as minimum, mean, and maximum values can shed light on the similarity of the synthetic data sets and the real data set. In addition, the normalization of the data

is reversed. As we can see, the descriptive statistics of the real data set and the synthetic data sets are not too dissimilar:

```
In [19]: data = pd.DataFrame({'real': y}, index=x

In [20]: N = 5  ❶
         for i in range(N):
             noise = rng.normal(0, 1, (len(y), 1)
             synthetic_data = generator.predict(n
             data[f'synth_{i:02d}'] = scaler.inve

In [21]: data.describe().round(3)
Out[21]:           real  synth_00  synth_01  synt
         count  500.000   500.000   500.000   500
         mean    -0.000    -0.110    -0.107    -0
         std      3.045     2.768     2.888     2
         min     -8.000   -12.046   -11.748   -10
         25%     -1.000    -0.890    -1.035    -1
         50%     -0.000    -0.031    -0.035    -0
         75%      1.000     0.862     0.884     0
         max      8.000     9.616    11.951     8
```

❶ Five synthetic data sets of full length are generated.

The real data set is generated from a *monotonically increasing* function. Therefore, the following visualization shows the real

data set and the synthetically generated data sets sorted, that is, in ascending order from the smallest to the largest value. As Figure 5-3 shows, the sorted synthetic data captures the basic shape of the real data quite well. It does it particularly well around 0. It does not do so well on the left and right limits of the interval. The similarity of the data sets is illustrated by the relatively low mean-squared error (MSE) for the first synthetic data set.

```
In [22]: ((data.apply(np.sort)['real'] -
            data.apply(np.sort)['synth_00']) ** 2
Out[22]: 0.2262292866493703

In [23]: data.apply(np.sort).plot(style=['r'] + N
```

❶ MSE for the sorted first synthetic data set, given the real data set.

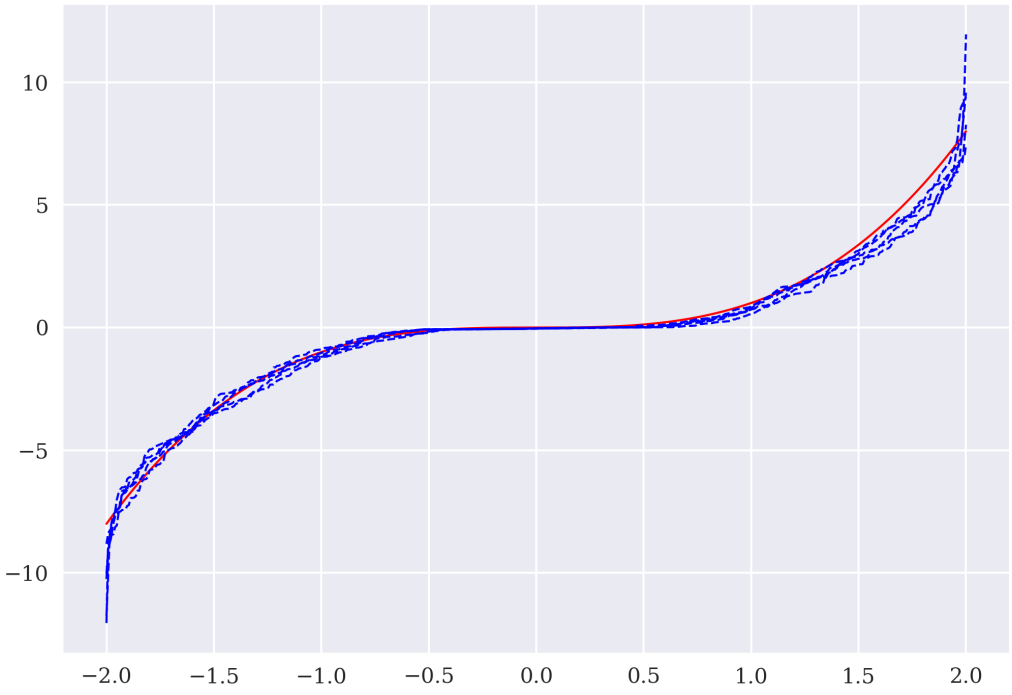

Figure 5-3. Real data (solid line) and sorted synthetic data sets (dashed lines)

Financial Example

This section applies the GAN approach from "Simple Example" to financial returns data. The goal for the generator is to generate synthetic returns data that the discriminator cannot distinguish from the real returns data. The Python code is essentially the same.

First, the financial data is retrieved and the log returns are calculated and normalized:

```
In [24]: raw = pd.read_csv('https://certificate.t
                           index_col=0, parse_date

In [25]: rets = raw['GLD'].iloc[-2 * 252:]  ❷
         rets = np.log((rets / rets.shift(1)).dro
         rets = rets.values  ❹

In [26]: scaler = StandardScaler()  ❺

In [27]: rets_ = scaler.fit_transform(rets.reshap
❺
```

❶ Retrieves the financial data set from the remote source.

❷ Selects, for a given symbol, a subset of the price data.

❸ Calculates the log returns from the price data.

❹ Transforms the `pandas Series` object into a `numpy ndarray` object.

❺ Applies Gaussian normalization to the log returns.

Second, the creation of the three models: the generator, the discriminator, and the GAN itself:

```
In [28]: rng = default_rng(100)
```

```
In [28]: rng = default_rng(100)
         tf.random.set_seed(100)

In [29]: generator = create_generator(hu=24)
         discriminator = create_discriminator(hu=
         gan = create_gan(generator, discriminato
```

Third, the training of the models:

```
In [30]: %time rd, sd = train_models(y_=rets_, ep
         Epoch: 0
         Epoch: 1000
         Epoch: 2000
         Epoch: 3000
         Epoch: 4000
         Epoch: 5000
         CPU times: user 1min 44s, sys: 10.6 s, t
         Wall time: 1min 45s
```

Fourth, the generation of the synthetic data. Figure 5-4 shows the real log returns and one synthetic data set for comparison:

```
In [31]: data = pd.DataFrame({'real': rets})

In [32]: N = 25

In [33]: for i in range(N):
```

```
In [33]: for i in range(N):
             noise = np.random.normal(0, 1, (len(
             synthetic_data = generator.predict(n
             data[f'synth_{i:02d}'] = scaler.inve

In [34]: res = data.describe().round(4)  ❸
         res.iloc[:, :5]  ❸
Out[34]:              real  synth_00  synth_01  syn
         count  503.0000  503.0000  503.0000  503
         mean     0.0002    0.0003    0.0007   -0
         std      0.0090    0.0088    0.0082    0
         min     -0.0302   -0.0269   -0.0385   -0
         25%     -0.0052   -0.0052   -0.0044   -0
         50%      0.0003   -0.0004    0.0007    0
         75%      0.0054    0.0059    0.0062    0
         max      0.0316    0.0263    0.0275    0

In [35]: data.iloc[:, :2].plot(style=['r', 'b--',
```

❶ Generates random synthetic data.

❷ Inverse transforms the data and stores it.

❸ Shows descriptive statistics for real and synthetic data.

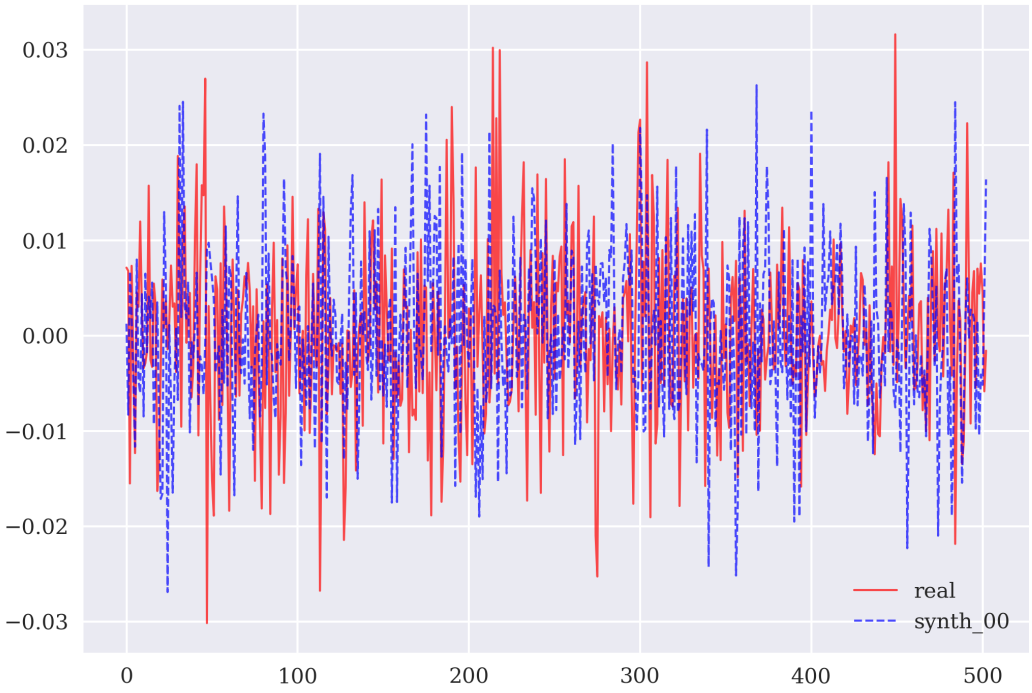

Figure 5-4. Real and synthetic log returns

The following Python code compares the real and synthetic log returns based on their histograms (see Figure 5-5). The histograms show a large degree of similarity:

```
In [36]: data['real'].plot(kind='hist', bins=50,
                           color='r', alpha=0.7)
         data['synth_00'].plot(kind='hist', bins=
                           label='synthetic', co
         plt.legend();
```

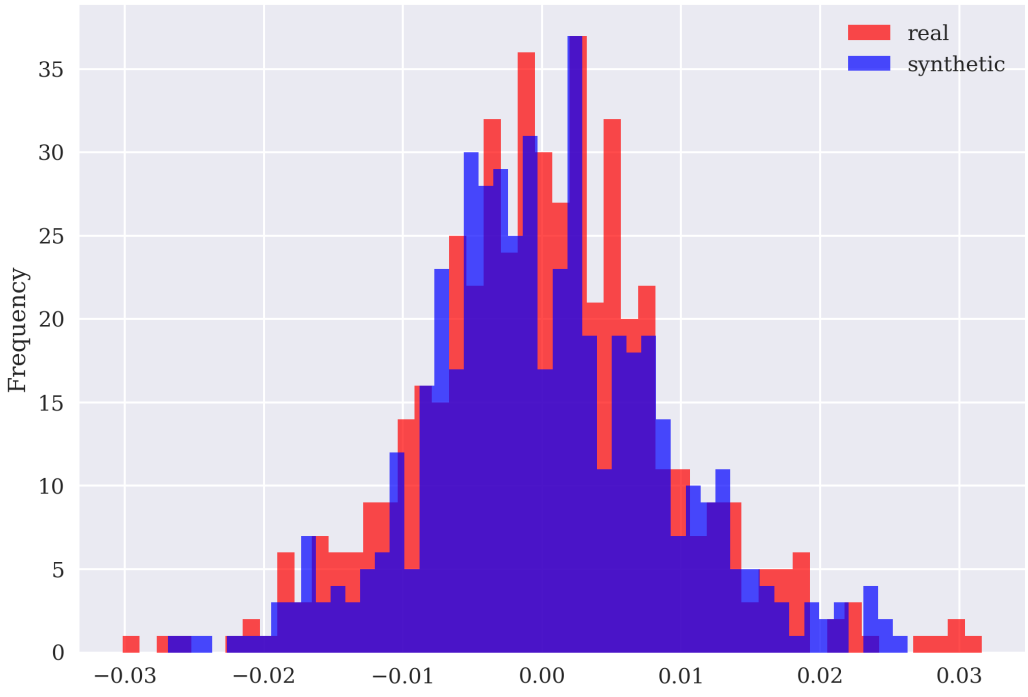

Figure 5-5. Histogram of the real and synthetic log returns

Figure 5-6 provides yet another comparison, this time based on the empirical cumulative distribution function (CDF) of the real and the synthetic log returns:

```
In [37]: plt.plot(np.sort(data['real']), 'r', lw=
         plt.plot(np.sort(data['synth_00']), 'b-
         plt.legend();
```

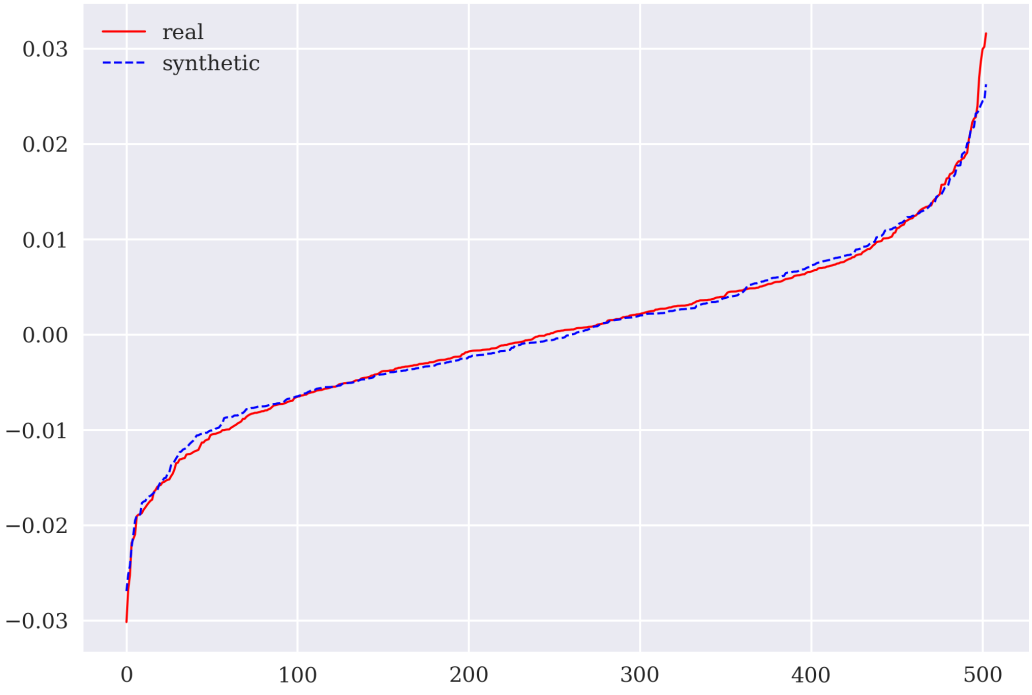

Figure 5-6. CDF of the real and synthetic log returns

Finally, the following Python code visualizes the cumulative real gross returns as well as several synthetic cumulative log return time series. The real financial time series looks like one that is generated with the GAN. Without the visual highlighting, it might indeed be indistinguishable from the other processes (see Figure 5-7).

```
In [38]: sn = N
         data.iloc[:, 1:sn + 1].cumsum().apply(np
             style='b--', lw=0.7, legend=False)
         data.iloc[:, 1:sn + 1].mean(axis=1).cums
```

```
            np.exp).plot(style='g', lw=2)
data['real'].cumsum().apply(np.exp).plot
```

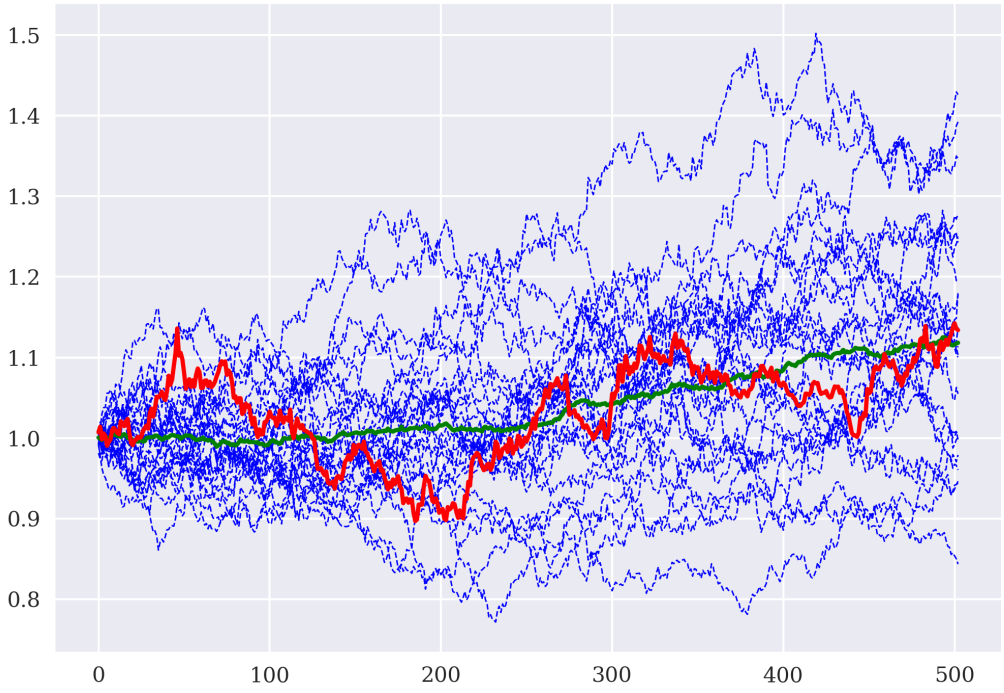

Figure 5-7. Real and synthetic cumulative log returns series

Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (KS) test is a statistical test that answers the following question: "How likely is it that a given data sample has been drawn from a given distribution?".[1] This description applies quite well to the situation in this chapter. A frequency distribution of the historical returns of a financial

instrument is given, and it is the starting point for everything. A GAN is trained based on these historical returns. The GAN then generates multiple return samples synthetically. The question is how likely is it — applying the KS test — whether a given synthetic sample is drawn from the original distribution of historically observed returns? In other words, can the generator not only fool the discriminator but also the KS test?

The following Python code implements the KS test on the synthetically generated data samples. The results show that the KS test indicates in all cases that the sample is likely from the original distribution. Figure 5-8 shows the frequency distribution of the p—values of the KS test. All p—values are above the threshold value of 0.05 (vertical line) — in many instances, the values are significantly larger than the threshold value. The GAN seems to do a great job of fooling the KS test into indicating that the synthetic samples are from the original distribution.

```
In [39]: from scipy import stats

In [40]: pvs = list()
         for i in range(N):
             pvs.append(stats.kstest(data[f'synth
         pvs = np.array(pvs)
```

```
In [41]: np.sort((pvs > 0.05).astype(int))
Out[41]: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
          1,
                1, 1, 1])

In [44]: sum(np.sort(pvs > 0.05)) / N
Out[44]: 1.0

In [43]: plt.hist(pvs, bins=100)
         plt.axvline(0.05, color='r');
```

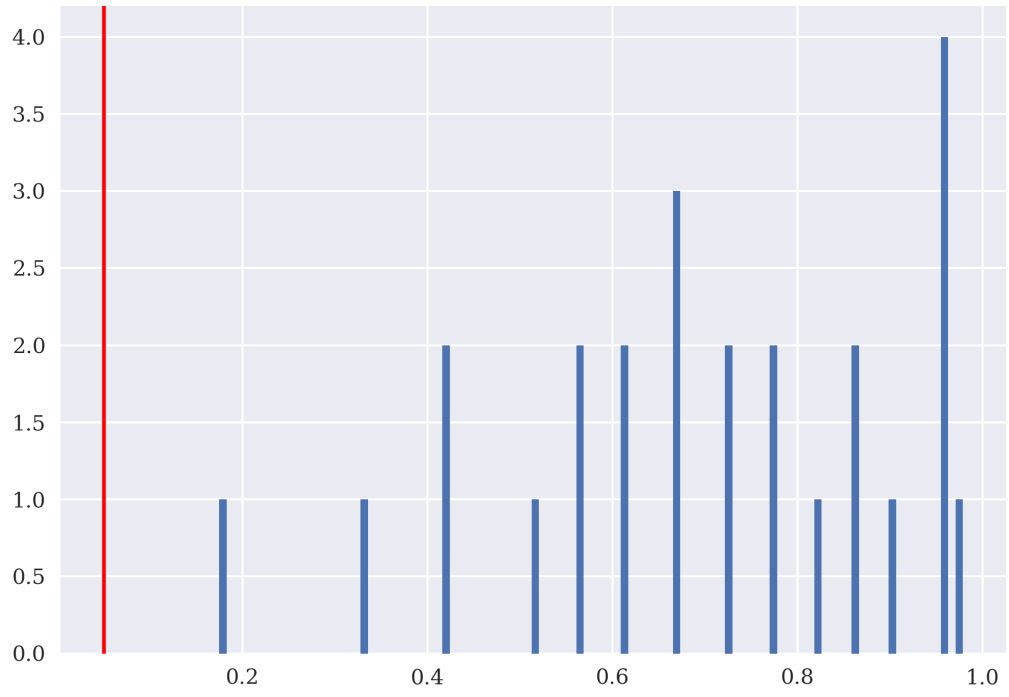

Figure 5-8. Histogram of p—values of KS test

THE POWER OF GANS

The GAN approach to generating synthetic time series data seems to be a great one. Visualizations generally do not allow a human observer to distinguish between real data and synthetic. Nor is a DNN, that is, the discriminator, capable of properly distinguishing between the data sets. In addition, as this section shows, traditional and widely used statistical tests also fail to properly distinguish between real and synthetic data. For RL learning projects, GANs therefore seem to provide one option to generate theoretically infinite synthetic data sets that have all the qualities of the original data set of interest.

Conclusions

Neural networks can be trained to generate data that is similar to, or even indistinguishable from, real financial data. This chapter introduces GANs based on simple sample data generated from a deterministic mathematical function. It then shows how to apply the same GAN architecture to log returns from a real financial time series. The result is the availability of a theoretically infinite number of generated financial time series that can be used in RL or other financial applications. Creswell et al. (2017) provide an early overview of GANs while Eckerli and Osterrieder (2021) do so particularly for GANs in finance.

At first sight, GANs seem to do something very similar to the Monte Carlo simulation approach from Chapter 4. However, there are major differences. Monte Carlo simulation in general relies on a relatively simple, parsimonious mathematical model. A few parameters can be chosen to reflect certain statistical facts of the real financial time series to be simulated. One such approach is the calibration of the model parameters to the prices of liquidly traded options on the financial instrument whose price series is to be simulated.[2]

On the other hand, GANs learn about the full distribution, say, of the log returns to be generated synthetically. The training of the generator DNN happens in competition with the discriminator DNN so the generator is getting better and better in mimicking the historical distribution. At the same time, the discriminator improves in distinguishing between real samples and synthetic samples of the log returns. Both DNNs are expected to improve during training to achieve good results overall.

The next part and the following chapters are about the application of the DQL algorithm to typical dynamic financial problems. They leverage the methods as introduced in this part to provide as many data samples for training and testing of the DQL agents as necessary.

References

Books and articles cited in this chapter:

- Ecerkli, Florian and Joerg Osterrieder (2021): “Generative Adversarial Networks in Finance: An Overview.” *https://ssrn.com/abstract=3864965*.
- Creswell, Antonia et al. (2017): “Generative Adversarial Networks: An Overview.” *https://arxiv.org/abs/1710.07035*.
- Goodfellow, Ian et al. (2014): “Generative Adversarial Nets.” *https://arxiv.org/abs/1406.2661*.
- Hilpisch, Yves (2015): *Derivatives Analytics with Python—Data Analysis, Models, Simulation, Calibration and Hedging.* John Wiley & Sons, Chichester.
- Kolmogorov, Andrey (1933): “Sulla Determinazione Empirica di una Legge di Distribuzione.” *Giornale dell’Istituto Italiano degli Attuari*, Vol. 4, 83–91.

The KS test dates back to the seminal paper by Kolmogorov already published in 1933.

Hilpisch (2015) provides details for the calibration of stochastic models for Monte Carlo simulation in the context of option pricing and hedging.

Your gateway to knowledge and culture. Accessible for everyone.