CHAPTER 1

Learning Through Interaction

> 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.

The next section discusses Bayesian learning as an example of learning through interaction. "Reinforcement Learning" on page 11 presents breakthroughs in AI that were made possible through RL. It also describes the major building blocks of RL. "Deep Q-Learning" on page 16 explains the two major characteristics of DQL, which is the most important algorithm in the remainder of the book.

Bayesian Learning

Two examples 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 thereby 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 (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. The reward is 1 USD if the agent wins and nothing if the agent loses. 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)

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)])  ❻
        rl[:10]
Out[5]: array([56, 47, 48, 55, 55, 51, 54, 43, 55, 40])

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 so 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(250)])
         rl[:10]
Out[10]: array([53, 56, 40, 55, 53, 49, 43, 45, 50, 51])

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(250)])
         rl[:10]
Out[14]: array([71, 65, 67, 69, 68, 72, 68, 68, 77, 73])

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

❶ The initial action space

❷ The update of the action space with the observed outcome

With remembering and learning, the agent achieves an average total reward of about \$66.80—a significant improvement over the random strategy without learning. This is close to the expected value of $(0.8^2 + 0.2^2) \cdot 100 = 68$.

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*.

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(250)])
         rl[:10]
Out[19]: array([81, 70, 74, 77, 82, 74, 81, 80, 77, 78])

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 $78.50, which is close to the theoretical optimum of $80. In this context, this strategy seems to be the optimal one.

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.

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 winning bet is rewarded with 1 USD and a losing bet is not rewarded:

1 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 available alternatives.

2 Modern psychology is a discipline focused on university students in particular, rather 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."

```
In [21]: ssp = [1, 2, 3, 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(250)])
         rl[:10]
Out[24]: array([ 92,  96, 106,  99,  96, 107, 101, 106,  92, 117])

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(250)])
         rl[:10]
Out[27]: array([182, 174, 162, 157, 184, 167, 190, 208, 171, 153])

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 $176, which is not that far from the expected value of that strategy of $(0.5^2 + 0.1^2 \cdot 5) \cdot 600 = 180$.

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 $297 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(250)])
         rl[:10]
Out[30]: array([305, 288, 312, 306, 318, 302, 304, 311, 313, 281])

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 subsections 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 (IID) according to the binomial distribution. Assume that an experiment yields f_h times heads and f_t times tails. Furthermore, assume that the binomial coefficient is given by the following:

$$B = \binom{f_h + f_t}{f_h}$$

In that case, we get $P(E \mid \alpha) = B \cdot \alpha^{f_h} \cdot (1 - \alpha)^{f_t}$ 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 $P(E \mid \alpha)$. The problem to solve is as follows:

$$\begin{aligned}
\alpha^{MLE} &= \arg\max_{\alpha} P(E \mid \alpha) \\
&= \arg\max_{\alpha} \ln P(E \mid \alpha) \\
&= \arg\max_{\alpha} \ln \left(B \cdot \alpha^{f_h} \cdot (1 - \alpha)^{f_t}\right) \\
&= \arg\max_{\alpha} \ln B + f_h \ln \alpha + f_t \ln (1 - \alpha)
\end{aligned}$$

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

$$\begin{aligned}
\frac{d}{d\alpha} P(E \mid \alpha) &= 0 \\
f_h \frac{d}{d\alpha} \ln \alpha + f_t \frac{d}{d\alpha} \ln (1 - \alpha) &= 0 \\
\frac{f_h}{\alpha} - \frac{f_t}{1 - \alpha} &= 0
\end{aligned}$$

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

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

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.

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 simultaneous learning from a generally very large number of labeled or unlabeled examples.

This section provides some general background on 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.

Major Breakthroughs

In AI research and practice, two types of algorithms have seen a meteoric rise over the last 10 years: *deep neural networks* (DNNs) and *reinforcement learning*.[4] While DNNs have had their own success stories in many different application areas, they also play an integral role in modern RL algorithms, such as *Q-learning* (QL).[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 AIs playing Atari 2600 games, chess, and Go at 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 at 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 breakthroughs 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 at 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 even beat the

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

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

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

world champion in early 2016.[7] The details are documented in Silver et al. (2017). They summarize:

> A long-standing goal of AI 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 chess and shogi. DeepMind calls this generalized agent AlphaZero. What is most impressive about AlphaZero is that it needs to spend only nine hours on training by self-playing chess 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, but also because AlphaZero evaluates a much lower number of positions per second than Stockfish:

> 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 part of AlphaZero as compared to predominantly brute force computation on the part of Stockfish.

7 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.

Reinforcement and Deep Learning

The breakthroughs in AI outlined in this subsection rely on a combination of RL and DL. While DL can be applied without RL in many scenarios, such as standard supervised and unsupervised learning situations, RL is applied today almost exclusively 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 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 for playing chess, or an environment for navigating a map indoors or outdoors. Nowadays, there are many such environments available for getting started with RL efficiently. One of the most popular ones is the Gymnasium environment.[8] On the Github page you read the following:

> 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, the board positions of all relevant pieces represent such a state. Sometimes, additional input is required; for example, whether castling has happened or not. For an Atari game, the pixels on 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 the player playing either the white or the 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 the allowed action types.

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

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 (on a "game clock"), chess players have quite a bit of 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 positions of the pieces must take place. Improvements in the results of the evaluation then represent a reward while a worsening of the results of the evaluation represents a penalty.

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

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 either lost all their "lives" or achieved the 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 RL. 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 board positions of all the pieces are relevant.

Deep Q-Learning

What characterizes 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. However, 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 with 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:

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

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

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

- Bayes, Thomas, and Richard Price. "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* 53 (1763): 370–418.
- Bellemare, Marc et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents." *Journal of Artificial Intelligence Research* 47, no. 1 (July 2012): 253–279.
- Bostrom, Nick. *Superintelligence: Paths, Dangers, Strategies*. Oxford, UK: Oxford University Press, 2014.
- Downey, Allen B. *Think Bayes: Bayesian Statistics in Python*. 2nd. ed. Sebastopol, CA: O'Reilly, 2021.
- Gerrish, Sean. *How Smart Machines Think*. Cambridge, MA: MIT Press, 2018.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2016.
- Hanel, Paul H. P., and Katia C. Vione. "Do Student Samples Provide an Accurate Estimate of the General Public?" *PLoS One* 11, no. 12 (2016).
- Mitchell, Tom. *Machine Learning*. New York, McGraw-Hill, 1997.
- Mnih, Volodymyr et al. "Playing Atari with Deep Reinforcement Learning". December 19, 2013.

- Mnih, Volodymyr et al. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (2015): 529–533.
- Rachev, Svetlozar et al. *Bayesian Methods in Finance*. Hoboken, NJ: John Wiley & Sons, 2008.
- Silver, David et al. "A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-Play." *Science* 362, no. 6419 (2018): 1140–1144.
- Silver, David et al. "Mastering the Game of Go Without Human Knowledge." *Nature* 550 (2017): 354–359.
- Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge and London: MIT Press, 2018.
- Watkins, Christopher. "Learning from Delayed Rewards." PhD diss., University of Cambridge, 1989.
- Watkins, Christopher, and Peter Dayan. "Q-Learning." *Machine Learning* 8 (1992): 279-282.
- West, Richard F., and Keith E. Stanovich. "Is Probability Matching Smart? Associations Between Probabilistic Choices and Cognitive Ability." *Memory & Cognition* 31, no. 2 (March 2003): 243–251.