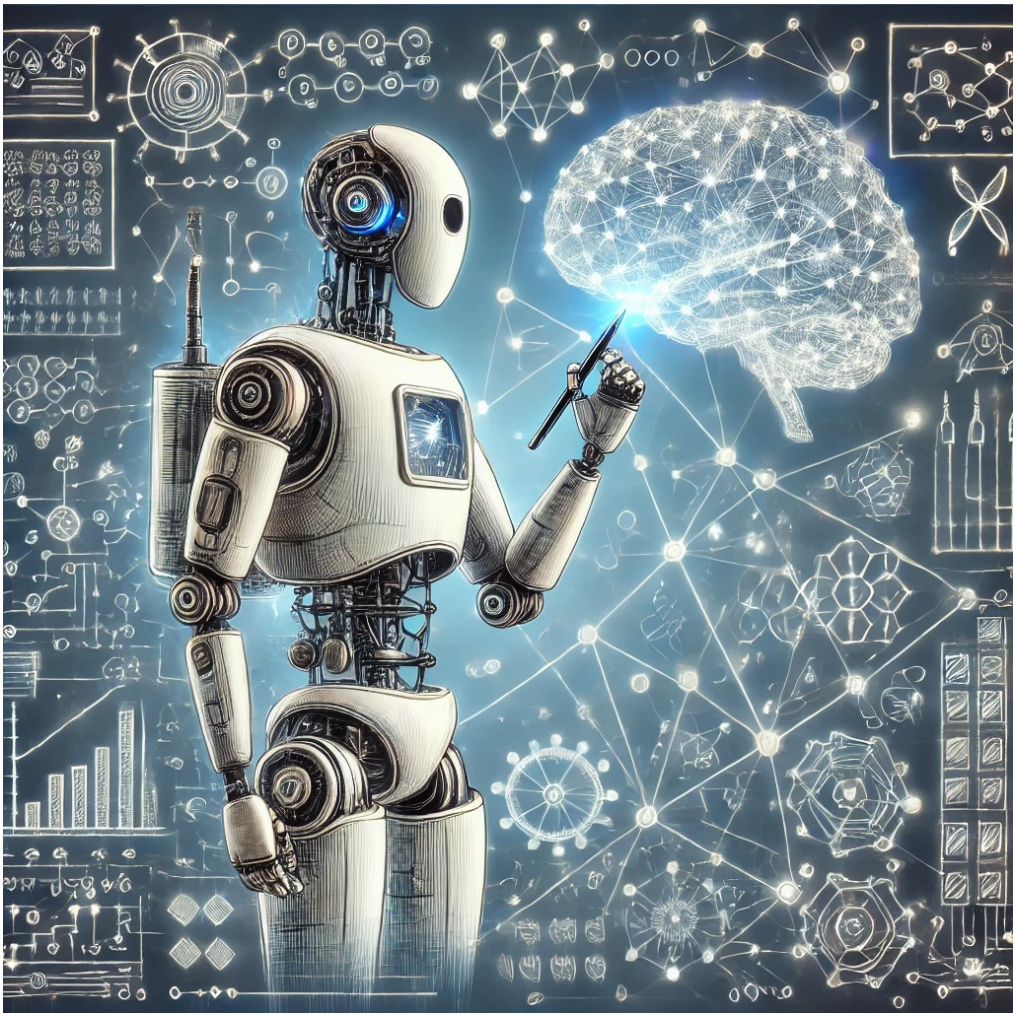
REINFORCEMENT LEARNING: ALGORITHMS & CONVERGENCE

CLEMENS HEITZINGER

mailto:Clemens.Heitzinger@TUWien.ac.at
http://Clemens.Heitzinger.name

Edition 0.1 — November 20, 2025



Contents

Preface

Reinforcement learning (RL) is an incredibly appealing subject. Firstly, it is a very general concept: An agent interacts with an environment with the goal to maximize the rewards it receives. The environment is random and communicates states and rewards to the agent, while the agent chooses actions according to a possibly random policy. The challenge is to find policies that maximize the expected value of all future rewards. Because reinforcement learning is such a general concept, it encompasses many real-world applications and is at the core of artificial intelligence and machine learning.

Secondly, the concepts and algorithms developed in reinforcement learning are leading to more and more general capabilities of artificial-intelligence (AI) systems and will very likely be part of artificial general intelligence. There are many similarities between reinforcement learning and how (human) brains work. These similarities must be explored in order to further our understanding of brains and human intelligence.

Thirdly, superhuman capabilities of RL learning algorithms have been demonstrated in various areas, and the list of extraordinary capabilities of AI systems built on reinforcement learning is continually expanding. Prominent examples are playing backgammon, Atari 2600 games, many more computer games, card games, chess, Go, and shogi at superhuman levels. Probably most famously, however, reinforcement learning is the last and crucial step in training large language models such as ChatGPT.

This book derives and describes the theory of reinforcement learning. The most fundamental as well as the most powerful RL algorithms are discussed. In addition to calculating optimal policies using powerful learning algorithms, we must also strive to provide guarantees regarding the quality of the learned results, i.e. the performance of the policies calculated by the algorithms. Hence the two main kinds of theoretical results concern the convergence of an algorithm to an optimal policy and performance guarantees for these policies. This book collects the theoretic foundations of reinforcement learning in view of these main questions.

To help translate theory into practice, this book also includes pseudocode and programming exercises that are concerned with the implementation of algorithms. The purpose of the programming exercises is to gain working knowledge, to try to exhibit both the advantages and disadvantages of certain methods, to show the challenges faced when developing new algorithms, and to inspire the reader to experiment with the algorithms. Both theoretic and programming exercises are an invitation to the reader to further explore this captivating subject.

Clearly, it was necessary to make choices – in many cases hard ones – about which algorithms, theorems, and proofs could be included in this book. Preference has been given to the more fundamental and general ones. In any case, care was taken to provide a complete and self-contained treatment. The appendix collects definitions, concepts, and results coming from outside reinforcement learning in a form that is expedient for our use here. Depending on the background of the reader, the appendix can be skipped in whole or in part.

This book can be used in various ways: It can be used for self-study, but also as the basis for courses on reinforcement learning. I hope that it is useful to various audiences such as anybody interested in AI and/or machine learning, to theoreticians interested in algorithms, theoretic results, and proofs, and to practitioners interested in the inner workings of the algorithms and seeking assurances in the quality of the computations.

I hope that you have as much fun reading the book as I had writing it.

Acknowledgments. I am happy to acknowledge the productive discussion with my students Markus Böck, Florian Chen, Patrizia Daxbacher, Sebastian Eresheim, Helmut Horvath, Lorenz Kapral, Tobias Kietreiber, Julien Malle, Daryna Oliynyk, Daniel Pasterk, Markus Peschl, Tobias Salzer, Jakob aus der Schmitten, Felix Wagner, and Richard Weiss as well as with my colleagues Felix Birkelbach, Rene Hofmann, and Carlotta Sophie von Tubeuf.

Furthermore, I am happy to acknowledge the support of the Austrian Research Promotion Agency (FFG) via the research project RELY (Reliable Reinforcement Learning for Sustainable Energy Systems).

Vienna, November 2025 *Clemens Heitzinger*

Chapter 1

Introduction

Here the basic concept of reinforcement learning and the fundamental notions used in formulating problems in reinforcement learning are introduced. Reinforcement learning is a very general concept and applies to time-dependent learning problems whenever an agent interacts with its environment. The main goal in reinforcement learning is to find optimal policies for the agent to follow. Examples of applications of reinforcement learning are given.

1.1 The Concept of Reinforcement Learning

One of the major appeals of reinforcement learning (RL) is that it applies to all situations where an agent interacts with an environment in a time-dependent manner.

The basic concept is illustrated in Figure 1.1. Firstly, the environment and the agent are possibly randomly initialized at the beginning of each episode. In each new time step t (which is increased in the top right in the figure), the environment and the agent enter the next state S_t and the agent receives the reward R_t. Then the agent chooses the next action A_t according to its possibly random policy. In the next iteration, this action affects the environment and the agent in a possibly random manner; the environment and the agent enter the next state; the agent receives a reward; and so forth.

In this manner, sequences of states S_t, actions A_t, and rewards R_t are generated. These sequences may be infinite (at least theoretically) or they end in a terminal state (which always happens in practice) after a finite number of time steps. A collection of a sequence of states S_t, a sequence of actions A_t, and a sequence of rewards R_t is called an episode. It is convenient to treat both cases, i.e., finite and infinite numbers of time steps, within the same theoretical

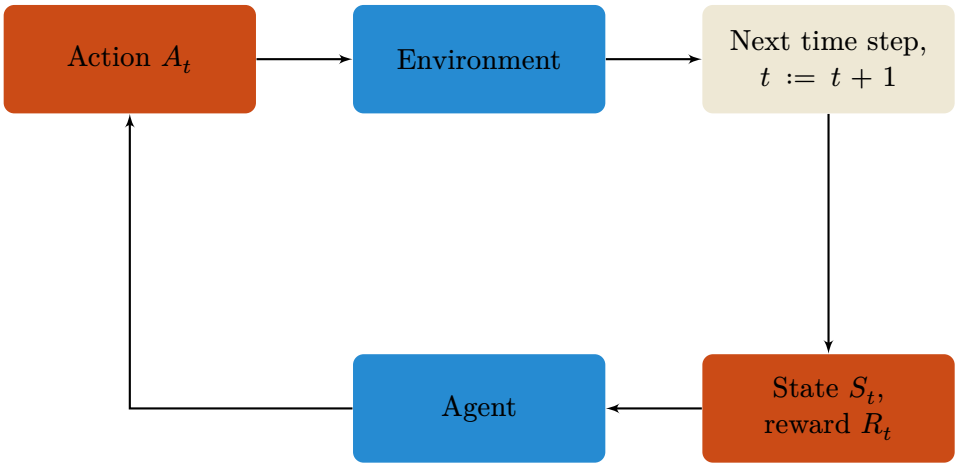


Figure 1.1: Environments, agents, and their interactions.

framework, which can be done under reasonable assumptions.

The return is the expected value of the sum of all (discounted) rewards the agent receives. We call a policy optimal if it maximizes the return.

The main goal in RL is to find optimal policies for the agent to follow. The input to a policy is the current state the agent finds itself in. The output of a deterministic policy is the action to take, and – more generally – the output of a random policy is a probability distribution that assigns probabilities to all actions that are possible in the state. Hence policies may be random, just as environments may be random.

It is necessary to allow random policies. An well-known example where the optimal policy must be random is the game of Rock Paper Scissors (Lizard Spock); if it were not random, it could easily be exploited by the opponent.

Because the main goal in RL is to find optimal policies, the main purpose of this book is to present and to discuss methods and algorithms that calculate such optimal policies.

Because of its generality, the concept of RL encompasses almost all real-world time-dependent problems whose solution requires foresight or intelligence. Therefore, it is at the core of artificial intelligence (AI). Any situation where an agent interacts with an environment and tries to optimize the sum of its rewards is a problem in the realm of RL, irrespective of the cardinality of the sets of spaces and actions and irrespective of whether the environment is deterministic or stochastic.

Depending on the problem, some information may be hidden from the agent. For example, in chess, there is no hidden information; the whole state of the

game is known to both players. On the other hand, in poker, the agents do not have access to all the information and it is hence called a hidden-information game. Therefore, there is observable and unobservable information. In this book, the information that is observable by the agent determines the state, i.e., unobservable information is not included in the definition of the state.

If the discount factor is zero, RL simplifies to supervised learning. In other words, RL is a generalization of supervised learning, which deals with problems that are not time-dependent. There is also a relation between RL and unsupervised learning. In RL, the initially unknown internal structure of the environment is learned and exploited in order to maximize the return. This is done implicitly, in contrast to unsupervised learning, whose goal is to bring hidden structures to light.

1.2 Examples of Applications

Clearly, the more complicated or random the environment is, the more challenging the RL problem is. Thanks to advanced algorithms and – in some cases – to gigantic amounts of data or computational resources or both, it has become possible to solve real-world problems at the level of human intelligence or above. Some applications, mentioned in alphabetical order, are discussed the following.

1.2.1 Autonomous Driving

In autonomous driving, the agent has to traverse a stochastic environment quickly and safely. The agent should also drive smoothly, except in dangerous situations, when it should act decisively. Sophisticated simulation environments including simulated sensor output have been developed, e.g. [1]. For the popular computer game Gran Turismo, agents that drive at a superhuman level have been developed [2]. RL agents can learn to drive on simulated highways and in simulated cities [3, 4], even in changing environments and with failing equipment [5].

1.2.2 Board Games: Backgammon, Chess, Shogi, and Go

Backgammon is a popular two-player board game that is played with counters and dice on tables boards. It is a member of the family of tables games, which date back nearly five thousand years. The use of dice implies that backgammon games include randomness.

In the 1990s, Gerald Tesauro developed the backgammon program TD-Gammon, which used temporal-difference learning and artificial neural networks

[6]. It achieved a level of play only slightly below that of the best human backgammon players at the time; in 1998, it lost a 100-game competition against the world champion by only eight points [7]. TD-Gammon also had strong impact on the backgammon community, since professional players soon adopted its evaluation of certain opening strategies.

In 1996, then world chess champion Garry Kasparov won a six-game match against IBM's Deep Blue chess program by 4:2. In the next year, an updated version of Deep Blue defeated Garry Kasparov by $3\frac{1}{2} : 2\frac{1}{2}$. In 2002, a match between then world chess champion Vladimir Kramnik and the chess program Deep Fritz ended in a 4:4 draw. Ever since that time, chess has been solved in the sense that the best chess programs play better than the best humans.

It is important to note that these chess programs were not self-learning, but they were based on tree search and fixed rules to assess the values of positions. Since self-learning is a defining characteristic of algorithms in machine learning and AI, these programs were not artificial intelligences.

After chess, Go was the only remaining classical board game which humans could play better than computers. Go games have a much larger search spaces than chess games, which is the reason why Go was the last unsolved board game and remained a formidable challenge. It was commonly believed at the time that it would take decades till Go can be solved.

In [8], an RL algorithm called AlphaGo Zero learned to beat the best humans consistently using no prior knowledge apart from the rules of the game, i.e., *tabula rasa,* albeit using vast computational resources. AlphaGo Zero is a truly self-learning algorithm.

In the next step, in [9], a more general RL algorithm called AlphaZero learned to play the three games of chess, shogi (Japanese chess), and Go again *tabula rasa*, but now using only self-play. It defeated world-champion programs in each of the three games.

1.2.3 Card Games: Poker and Schnapsen

Card games differ from board games regarding the amount of information that is available to the players. In board games, the players know the full state of the game at all times, whereas in card games some information is available to all players, some information only to certain players, and some information is known by no player. Such games are called hidden-information games.

In 2019, Brown and Sandholm reported on their poker program program, dubbed Pluribus, that plays six-player no-limit Texas hold'em [10]. Pluribus learned by self-play, playing against five copies of itself. In order to evaluate the performance of Pluribus, it had to compete with five elite professional poker

players or with five copies of itself playing against one professional. Over the course of 10 000 hands of poker, it was found that it performs significantly better than humans [10].

Schnapsen is trick-taking card game of the ace-ten family for two players. It is the national card game of Austria and Hungary, and it is very popular in Bavaria and Upper Silesia as well. Schnapsen is a both a point-trick and trick-and-draw game and involves lots of strategy. Schnapsen is played in many tournaments, and there are also variants for three and four players (Dreierschnapsen and Bauernschnapsen, respectively). In contrast to poker, it is not considered a game of luck, but a game of skill, according to Austrian law. Schnapsen can be traced back to the 1700s, and the earliest description of its predecessor Mariage dates back to 1715.

The aim of the game is to take tricks in order to collect 66 or more card points as quickly as possible in rounds. In each round, a player can win one, two, or three game points. The player who wins seven game points first wins the game.

In his master's thesis, Tobias Salzer developed a deep RL algorithm for Schnapsen, which beat the previously strongest Schnapsen program, playing above human expert level, in a tournament of twenty games by 12:8 [11].

1.2.4 Computer Games

The study of computer games in the context of AI dates back at least to Falken's Maze [12].

In the context of RL playing computer games, the game is the environment and the agent has to learn an optimal strategy. One of the great successes in the history of RL and a precursor to the single algorithm that solved chess, Go, and shogi (see Section 1.2.2) was a single deep RL algorithm that can learn to play many (but not all) Atari 2600 games at the human level or above [13].

A few years later, an RL algorithm learned to play Quake III Arena in a mode called "Capture the Flag" at the human level [14]. Also in 2019, a multi-agent RL algorithm learned to play StarCraft II, a contemporary computer game, at the grandmaster level. It did not achieve clearly superhuman capabilities, which may be due to the hidden information in this computer game.

1.2.5 Finance

Oftentimes in finance, the environment is a certain market, the state is given by the positions in the portfolio, and the profits become positive rewards, while a

risk measure may become a negative reward. In this manner, wealth is maximized, while the risk is managed.

1.2.6 Medicine

The main goal in medicine is to treat patients so that they recover as fully and as quickly as possible. In the nomenclature of RL, the patient is the environment, the agent is the medical doctor, the state is given by any measurements of the patient's condition, and the actions are the therapies or medications. In this manner, treating patient becomes an RL problem. Questions such as how to find optimal therapies, how to personalize therapies, and how to evaluate therapies thus become amenable to quantification and can be solved by RL algorithms. This approach to medicine paves the way towards truly personalized and predictive medicine.

Intensive care is particularly well-suited for RL, because many values are recorded over relatively short periods of time in intensive-care units. Typical episodes last a few days, and hundreds of values may be recorded every few hours.

The most prevalent condition and at the same time the most prevalent cause of death in intensive-care units is sepsis. Therefore sepsis is an obvious application of RL in medicine. After clustering of the patients' conditions, RL algorithms for a finite number of states can be utilized. When the actions correspond to two medications at different doses, it was found that RL policies reach or slightly surpass the performance of human doctors [15, 16, 17].

1.2.7 Optimal Control and Robotics

Reinforcement learning can be viewed as optimal control of stochastic environments or systems. It does not matter whether there is a model of the environment or not. If there is model, it is called a model-based control problem, also often called a planning problem; if there is no model, it is called a model-free control problem, also often called a learning problem.

There is a vast number of applications of RL in automation, optimal control, and robotics. Examples for the optimal control of industrial systems are chemical and biological reactors. The output of the product is to be maximized, while the reactor must be kept within safe operating conditions. In robotics, the robot interacts with its environment in order to solve the task it has been assigned by specifying the rewards and/or penalties. Ideally, optimal policies are then learned without any further help or interaction. Generally speaking, RL can used to automate any kind of equipment or machinery, a few examples being

[18, 19].

1.2.8 Recommendation Systems

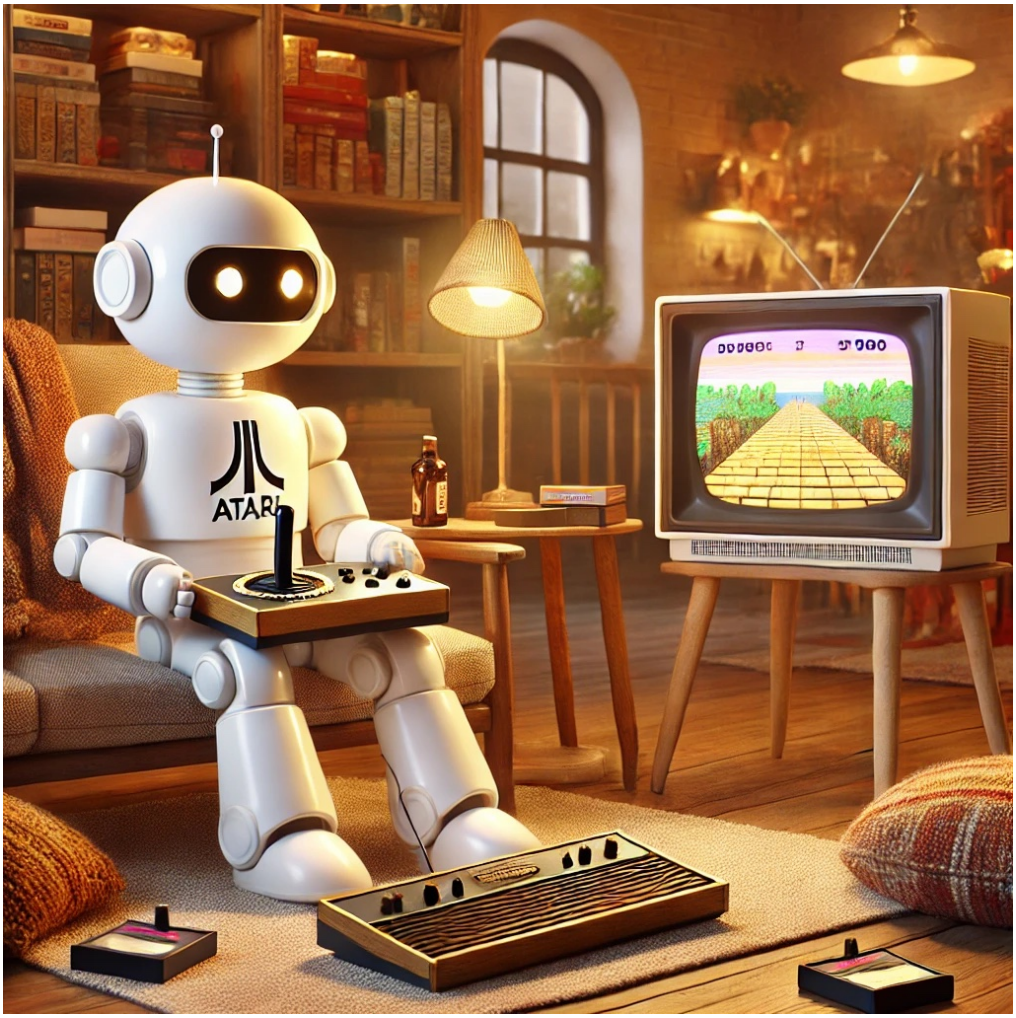
Although recommendation problems have traditionally often been considered to be classification or prediction problems, they are nowadays often formulated as sequential decision problems, which can better reflect the multiple interactions between users and the recommendation system. RL is used to directly optimize for user satisfaction or other metrics in various areas such as music and video playlists [20, 21, 22]. It is known that traffic is routed through RL systems on many large websites.

1.2.9 Supply Chains and Inventories

In supply-chain and inventory management, goods must be moved efficiently such that they arrive at their points of destination in sufficient, but not abundant quantity. Rewards are obtained whenever the goods arrive on time in the specified quantity at their points of destination, while penalties are obtained when too few or too many goods arrive or when they are delayed.

1.3 Bibliographical Remarks

Here textbooks on RL are mentioned. The most important and influential introductory textbook is [23]. RL is closely related to optimal control and often builds on dynamic programming; textbooks at the intersections of these subjects are [24, 25, 26]. [27]. A textbook on distributional RL is [28]. A comprehensive collection of approaches to dynamic programming, RL, and stochastic optimization can be found in [29]. Practical approaches to deep RL are discussed in [30, 31, 32, 33].


ATARI

Chapter 2

Markov Decision Processes and Dynamic Programming

> I was intrigued by dynamic programming. It was clear to me that there was a good deal of good analysis there. Furthermore, I could see many applications. It was a clear choice. I could either be a traditional intellectual, or a modern intellectual using the results of my research for the problems of contemporary society. This was a dangerous path. Either I could do too much research and too little application, or too little research and too much application. I had confidence that I could do this delicate activity, pie à la mode.
>
> Richard Bellman [34, p. 173].

This chapter starts with one of the simplest learning problems, the so-called multi-armed bandits. Multi-armed bandits serve to introduce basic notions and challenges in time-dependent learning. Then Markov decision processes are defined, as they serve as the foundation for the description of reinforcement-learning problems. Once the basic language to describe reinforcement-learning problems has been defined, a collection of standard environments is presented. Based on the definitions, the most important notions in and results of dynamic programming are presented, since they serve as a foundation that inspires learning algorithms.

2.1 Multi-Armed Bandits

A relatively simple, but illustrative example of a reinforcement-learning problem are multi-armed bandits. The name of the problem stems from slot machines.

There are $k = |\mathcal{A}|$ slot machines or bandits, and the action is to choose one machine and play it to receive a reward. The reward each slot machine or bandit distributes is taken from a stationary probability distribution, which is of course unknown to the agent and different for each machine.

In more abstract terms, the problem is to learn the best policy when being repeatedly faced with a choice among k different actions. After each action, the reward is chosen from the stationary probability distribution associated with each action. The objective of the agent is to maximize the expected total reward over some time period or for all times.

In time step t, the action is denoted by $A_t \in \mathcal{A}$ and the reward by R_t. In this example, we define the (true) value of an action a to be

$$q_* \colon \mathcal{A} \to \mathbb{R}, \qquad q_*(a) := \mathbb{E}[R_t \mid A_t = a], \qquad a \in \mathcal{A}.$$

(This definition is simpler than the general one, since the time steps are independent from one another. There are no states.) This function is called the action-value function.

Since the true value of the actions is unknown (at least in the beginning), we calculate an approximation called $Q_t \colon \mathcal{A} \to \mathbb{R}$ in each time step; it should be as close to q_* as possible. A reasonable approximation is the sample mean of the observed rewards, i.e.,

$$Q_t(a) := \frac{\text{sum of rewards obtained when action } a \text{ was taken prior to } t}{\text{number of times action } a \text{ was taken prior to } t}.$$

Based on this approximation, the greedy way to select an action is to choose

$$A_t :\in \arg\max_a Q_t(a),$$

which serves as a substitute for the ideal choice

$$\arg\max_a q_*(a).$$

Note that $\arg\max_a Q_t(a)$ is a set, since multiple actions may maximize the argument. Hence the notation "$x :\in X$" means that an element of X is chosen randomly with each element having the same probability and assigned to x, analogously to the notation ":=".

In summary, these simple considerations have led us to a first example of an action-value method. In general, an action-value method is a method which is based on (an approximation Q_t of) the action-value function q_*.

Choosing the actions in a greedy manner is called exploitation. However, there is a problem. In each time step, we only have the approximation Q_t at

our disposal. For some of the k bandits or actions, it may be a bad approximation, in the sense that it leads us to choose an action a whose estimated value $Q_t(a)$ is higher than its true value $q_*(a)$. Such an approximation error would be misleading and reduce or rewards.

In other words, exploitation by greedy actions is not enough. We also have to ensure that our approximations are sufficiently accurate; this process is called exploration. If we explore all actions sufficiently well, bad actions cannot hide behind high rewards obtained by chance.

The duality between exploitation and exploration is fundamental to reinforcement learning, and it is worthwhile to always keep these two concepts in mind. Here we saw how these two concepts are linked to the quality of the approximation of the action-value function q_*.

In general, a greedy policy always exploits the current knowledge (in the form of an approximation of the action-value function) in order to maximize the immediate reward, but it spends no time on the long-term picture. A greedy policy does not sample apparently worse actions to see whether their true action values are better or whether they lead to more desirable states. (Note that the multi-bandit problem is stateless.)

A common and simple way to combine exploitation and exploration into one policy in the case of finite action sets is to choose the greedy action most of the time, but any action with a (usually small) probability ϵ. This is the subject of the following definition.

Definition 2.1 (ϵ-greedy policy)**.** Suppose that the action set $\mathcal{A}$ is finite, that Q_t is an approximation of the action-value function, and that $\epsilon_t \in [0, 1]$. Then the policy defined by

$$A_t := \begin{cases} \arg\max_{a \in \mathcal{A}} Q_t(a) & \text{with probability } 1 - \epsilon_t \text{ breaking ties randomly,} \\ \text{a random action } a & \text{with probability } \epsilon_t \end{cases}$$

is called the ϵ-greedy policy.

In the first case, it is important to break ties randomly, because otherwise a bias towards certain actions would be introduced. The random action in the second case is usually chosen uniformly from all actions $\mathcal{A}$.

Learning methods that use ϵ-greedy policies are called ϵ-greedy methods. Intuitively speaking, every action will be sampled an infinite number of times as $t \to \infty$, which ensures convergence of Q_t to q_*.

Regarding the numerical implementation, it is clear that storing all previous actions and rewards becomes inefficient as the number of time steps increases. Can memory and the computational effort per time step be kept constant, which

would be the ideal case? Yes, it is possible to achieve this ideal case using a trick, which will lead to our first learning algorithm.

To simplify notation, we focus on the action-value function of a certain action. We denote the reward received after the k-th selection of this specific action by R_k and use the approximation

$$Q_n := \frac{1}{n-1} \sum_{k=1}^{n-1} R_k$$

of its action value after the action has been chosen $n-1$ times. This is called the sample-average method. The trick is to find a recursive formula for Q_n by calculating

$$\begin{aligned} Q_{n+1} &= \frac{1}{n} \sum_{k=1}^{n} R_k \\ &= \frac{1}{n} \left(R_n + (n-1) \frac{1}{n-1} \sum_{k=1}^{n-1} R_k \right) \\ &= \frac{1}{n} (R_n + (n-1) Q_n) \\ &= Q_n + \frac{1}{n} (R_n - Q_n) \qquad \forall n \geq 1. \end{aligned}$$

(If $n = 1$, this formula holds for arbitrary values of Q_1 so that the starting point Q_1 does not play a role.)

This yields our first learning algorithm, Algorithm 1. The implementation of this recursion requires only constant memory for n and Q_n and a constant amount of computation in each time step.

The recursion above has the form

$$\text{new estimate} := \text{old estimate} + \text{learning rate} \cdot (\text{target} - \text{old estimate}), \quad (2.1)$$

which is a common theme in reinforcement learning. Its intuitive meaning is that the estimate is updated towards a target value. Since the environment is stochastic, the learning rate only moves the estimate towards the observed target value. Updates of this form will occur many times in this book.

In the most general case, the learning rate α depends on the time step and the action taken, i.e., $\alpha = \alpha_t(a)$. In the sample-average method above, the learning rate is $\alpha_n(a) = 1/n$. It can be shown that the sample-average approximation Q_n above converges to the true action-value function q_* by using the law of large numbers.

Algorithm 1 a simple algorithm for the multi-bandit problem.

Initialization:
choose $\epsilon \in (0, 1)$,
initialize two vectors $\mathbf{q}$ and $\mathbf{n}$ of length $|\mathcal{A}|$ with zeros.

loop
  select an action a ϵ-greedily using $\mathbf{q}$ (see Definition 2.1)
  perform the action a and receive the reward r from the environment
  $n_a := n_a + 1$
  $q_a := q_a + \frac{1}{n_a}(r - q_a)$
end loop

return q

Sufficient conditions that yield convergence with probability one are

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty, \tag{2.2a}$$

$$\sum_{k=1}^{\infty} \alpha_k(a)^2 < \infty. \tag{2.2b}$$

They are well-known in stochastic-approximation theory and are a recurring theme in convergence proofs. The first condition ensures that the steps are sufficiently large to eventually overcome any initial conditions or random fluctuations. The second condition means the steps eventually become sufficiently small. Of course, these two conditions are satisfied for the learning rate

$$\alpha_k(a) := \frac{1}{k},$$

but they are not satisfied for a constant learning rate $\alpha_k(a) := \alpha$. However, a constant learning rate may be desirable when the environment is time-dependent, since then the updates continue to adjust the policy to changes in the environment.

Finally, we shortly discuss an important improvement over ϵ-greedy policies, namely action selection by upper confidence bounds. The disadvantage of an ϵ-greedy policy is that it chooses the non-greedy actions without any further consideration. It is however better to select the non-greedy actions according to their potential to actually being an optimal action and according to the

uncertainty in our estimate of the value function. This can be done using the so-called upper-confidence-bound action selection

$$A_t :\in \arg\max_{a\in\mathcal{A}} \left(Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right),$$

where $N_t(a)$ is the number of times that action a has been selected before time t. If $N_t(a) = 0$, the action a is treated as an optimal action. The constant c controls the amount of exploration.

The term $\sqrt{(\ln t)/N_t(a)}$ measures the uncertainty in the estimate $Q_t(a)$. When the action a is selected, $N_t(a)$ increases and the uncertainty decreases. On the other hand, if an action other than a is chosen, t increases and the uncertainty relative to other actions increases. Therefore, since $Q_t(a)$ is the approximation of the value and $c\sqrt{(\ln t)/N_t(a)}$ is the uncertainty, where c is the confidence level, the sum of these two terms acts as an upper bound for the true value $q_*(a)$.

Since the logarithm is unbounded, all actions are ensured to be chosen eventually. Actions with lower value estimates $Q_t(a)$ and actions that have often been chosen (large $N_t(a)$ and low uncertainty) are selected with lower frequency, just as they should in order to balance exploitation and exploration.

2.2 Markov Decision Processes

The mathematical language and notation for describing and solving reinforcement-learning problems is deeply rooted in Markov decision processes. Having discussed multi-armed bandits as a concrete example for a reinforcement-learning problem, we now generalize some notions and fix the notation for the rest of the book using the language of Markov decision processes.

As already discussed in Chapter 1, the whole world is divided into an agent and an environment. The agent interacts with the environment iteratively. The agent takes actions in the environment, which changes the state of the environment and for which it receives a reward (see Figure 1.1). The task of the agent is to learn optimal policies, i.e., to find the best action in order to maximize all future rewards it will receive. We will now formalize the problem of finding optimal policies.

We note the sequence of (usually discrete) time steps by $t \in \{0, 1, 2, \dots\}$. The state (or observation) that the agent receives in time step t from the environment is denoted by $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all states. We use capital letters to denote random variables. In general, $\mathcal{S} \subset \mathbb{R}^{d_s}$, $d_s \in \mathbb{N}$, but if there is a finite number of states, $\mathcal{S} \subset \mathbb{Z} \subset \mathbb{R}$ suffices.

The action performed by the agent in time step t is denoted by $A_t \in \mathcal{A}(s)$ if the environment is in state $s = S_t$. In general, $\mathcal{A} \subset \mathbb{R}^{d_a}$, $d_a \in \mathbb{N}$, but if there is a finite number of actions, $\mathcal{A}(s) \subset \mathbb{Z} \subset \mathbb{R}$ suffices. In general, the set $\mathcal{A}(s)$ of all available actions depends on the very state s, although this dependence is sometimes dropped to simplify notation. In the subsequent time step, the agent receives the reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and finds itself in the next state S_{t+1}. Then the iteration is repeated.

The whole information about these interactions between the agent and the environment can be recorded in sequences $\langle S_t \rangle_{t\in\mathbb{N}}$, $\langle A_t \rangle_{t\in\mathbb{N}}$, and $\langle R_t \rangle_{t\in\mathbb{N}}$ or in the sequence

$$\langle S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, \dots \rangle$$

These (finite or infinite) sequences are called episodes.

The random variables S_t and R_t provided by the environment depend only on the preceding state and action. This is the Markov property, and the whole process is a Markov decision process (MDP). We assume that the Markov property is always satisfied.

In a finite MDP, all three sets $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ are finite, and hence the random variables S_t, A_t, and R_t are discrete.

The purpose of the following definitions is to describe the dynamics of the environment, i.e., the random variables S_{t+1} and R_{t+1} knowing $S_t = s$ and $A_t = a$, starting from the general case of states and rewards that are real vectors and ending with finite MDP.

The cumulative distribution function F_X of a random variable X is defined as

$$F_X(x) := \mathbb{P}[X \leq x]$$

in the univariate case and as

$$F_{X_1,\dots,X_d}(x_1, \dots, x_d) := \mathbb{P}[X_1 \leq x_1 \wedge \dots \wedge X_d \leq x_d]$$

in the multivariate case, where $d \in \mathbb{N}$. For vectors $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{y} \in \mathbb{R}^d$, we define

$$\mathbf{x} \leq \mathbf{y} \quad :\Longleftrightarrow \quad x_k \leq y_k \quad \forall k \in \{1, \dots, d\}.$$

Then we can write

$$F_X(\mathbf{x}) = \mathbb{P}[X \leq \mathbf{x}].$$

In general, the expectation of a function h of a random variable X is defined as

$$\mathbb{E}[h(x)] := \int_{\mathbf{x}\in\mathbb{R}^d} h(\mathbf{x}) \mathrm{d}F_X(\mathbf{x}) = \int_{x_1=-\infty}^{\infty} \cdots \int_{x_d=-\infty}^{\infty} h(\mathbf{x}) \mathrm{d}F_{X_d}(x_d) \cdots \mathrm{d}F_{X_1}(x_1).$$

The probability of being put into state $s' \in \mathcal{S}$ and receiving a reward $r \in \mathcal{R}$ after starting from a state $s \in \mathcal{S}$ and performing action $a \in \mathcal{A}(s)$ is recorded by the transition probability

$$p\colon \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0,1],$$
$$p(s', r \mid s, a) := \mathbb{P}[S_t = s',\ R_t = r \mid S_{t-1} = s,\ A_{t-1} = a].$$

Despite the notation with the vertical bar in the argument list of p, which is reminiscent of a conditional probability, the function p is a deterministic function of four arguments.

The function p records the dynamics of the MDP, and it is therefore also called the dynamics function of the MDP. Since it is a probability distribution, the equality

$$\forall s \in \mathcal{S}\colon \quad \forall a \in \mathcal{A}(s)\colon \qquad \int_{\mathcal{S}} \int_{\mathcal{R}} p(s', r \mid s, a) \mathrm{d}s' \mathrm{d}r = 1$$

holds.

If there is a finite number of states and rewards, the integrals become sums and the equalities are

$$\forall s \in \mathcal{S}\colon \quad \forall a \in \mathcal{A}(s)\colon \qquad \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1.$$

The requirement that the Markov property holds is met by ensuring that the information recorded in the states $s \in \mathcal{S}$ is sufficient. This is an important point when translating an informal problem description into the framework of MDPs and reinforcement learning. In practice, this often means that the states become sufficiently long vectors that contain enough information about the past ensuring that the Markov property holds. This in turn has the disadvantage that the dimension of the state space may have to increase to ensure the Markov property.

It is important to note that the transition probability p, i.e., the dynamics of the MDP, is *unknown*. It is sometimes said that the term *learning* refers to problems where information about the dynamics of the system is absent. Learning algorithms face the task of calculating optimal strategies with only very little knowledge about the environment, i.e., just the sets $\mathcal{S}$ and $\mathcal{A}(s)$.

The dynamics function contains all relevant information about the MDP, and therefore other quantities can be derived from it. The first quantity are the state-transition probabilities

$$p\colon \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0,1],$$

$$p(s' \mid s, a) := \mathbb{P}\{S_t = s' \mid S_{t-1} = s,\ A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a),$$

also denoted by p, but taking only three arguments.

Next, the expected rewards for state-action pairs are

$$r \colon \mathcal{S} \times \mathcal{A} \to \mathbb{R},$$
$$r(s, a) := \mathbb{E}[R_t \mid S_{t-1} = s,\ A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a).$$

(Note that $\sum_{r \in \mathcal{R}} \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) = 1$ must hold.) Furthermore, the expected rewards for state–action–next-state triples are given by

$$r \colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}, \tag{2.3a}$$
$$r(s, a, s') := \mathbb{E}[R_t \mid S_{t-1} = s,\ A_{t-1} = a,\ S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r \mid s, a)}{p(s' \mid s, a)}. \tag{2.3b}$$

(Note that $\sum_{r \in \mathcal{R}} p(s', r \mid s, a)/p(s' \mid s, a) = 1$ must hold.)

MDPs can be visualized as directed graphs. The nodes are the states, and the edges starting from a state s correspond to the actions $\mathcal{A}(s)$. The edges starting from state s may split and end in multiple target nodes s'. The edges are labeled with the state-transition probabilities $p(s' \mid s, a)$ and the expected reward $r(s, a, s')$.

Further important data structures are the transition and the experience. A transition is a tuple

$$(s, a, s', r, \text{done?}),$$

which records all information in a time step. Therefore an episode can be viewed as a sequence of transitions. The binary variable done? records whether a terminal state has been reached and the episode has ended; this information is useful in certain learning algorithms. An experience (also often called an experience-replay buffer) is a set of transitions.

2.3 Rewards, Returns, and Episodes

We start with a remark on how rewards should be defined in practice when translating an informal problem description into a precisely defined environment. It is important to realize that the learning algorithms will learn to maximize the expected value of the discounted sum of all future rewards (defined below), nothing more and nothing less.

For example, if the agent should learn to escape a maze quickly, it is expedient to set $R_t := -1$ for all times t. This ensures that the task is completed quickly.

The obvious alternative to define $R_t := 0$ before escaping the maze and a positive reward when escaping fails to convey to the agent that the maze should be escaped quickly; there is no penalty for lingering in the maze.

Furthermore, the temptation to give out rewards for solving subproblems must be resisted. For example, when the goal is to play chess, there should be no rewards to taking opponents' pieces. Because otherwise the agent would become proficient in taking opponents' pieces, but not in checkmating the king.

From now on, we make the following assumption, which is needed for defining the return in the general case, which is the next concept we discuss.

Assumption 2.2 (bounded rewards)**.** The reward sequence $\langle R_t \rangle_{t \in \mathbb{N}}$ is bounded.

There are two cases to be discerned, namely whether the episodes are finite or infinite. We denote the time of termination, i.e., the time when the terminal state of an episode is reached, by T. The case of a finite episode is called episodic, and $T < \infty$ holds; the case of an infinite episode is called continuing, and $T = \infty$ holds.

Definition 2.3 (discounted return)**.** The *discounted return* is

$$G_t := \sum_{k=t+1}^{T} \gamma^{k-(t+1)} R_k = R_{t+1} + \gamma R_{t+2} + \cdots,$$

where $T \in \mathbb{N} \cup \{\infty\}$ is the terminal state of the episode and $\gamma \in [0, 1]$ is the discount rate.

From now on, we also make the following assumption.

Assumption 2.4 (finite returns)**.** $T = \infty$ and $\gamma = 1$ do not hold at the same time.

Assumptions 2.2 and 2.4 ensure that all returns are finite.

There is an important recursive formula for calculating the returns from the rewards of an episode. It is found by calculating

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-(t+1)} R_k \tag{2.4a}$$

$$= R_{t+1} + \gamma \sum_{k=t+2}^{T} \gamma^{k-(t+2)} R_k \tag{2.4b}$$

$$= R_{t+1} + \gamma G_{t+1}. \tag{2.4c}$$

The calculation also works when $T < \infty$, since $G_T = 0$, $G_{T+1} = 0$, and so forth since then the sum in the definition of G_t is empty and hence equal to zero. This formula is very useful to quickly compute returns from reward sequences.

At this point, we can formalize what (classical) problems in reinforcement learning are.

Definition 2.5 (reinforcement-learning problem)**.** Given the states $\mathcal{S}$, the actions $\mathcal{A}(s)$, and the opaque transition function $\mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathcal{R}$ of the environment, a reinforcement-learning problem consists of finding policies for selecting the actions of an agent such that the expected discounted return is maximized.

The random transition provided by the MDP of the environment, namely going from a state-action pair to a state-reward pair, being opaque means that we consider it a black box. For any state-action pair as input, it is only required to yield a state-reward pair as output. In particular, the transition probabilities are considered to be unknown. Examples of such opaque environments are

- functions $\mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathcal{R}$ defined in a programming language,
- more complex pieces of software such as computer games or simulation software,
- historic data from which states, actions, and rewards can be observed.

The fact that the problem class in Definition 2.5 is so large adds to the appeal of reinforcement learning.

2.4 Standard Environments

At this point, we have defined the basic concepts of RL that are necessary to translate applications into the language of RL. In this section, various standard environments are defined. They serve several purposes. First, they serve as leading examples of RL problems in different application areas. Several environments in this collection have a long history in RL. Second, most of these environments have well-known solutions, and thus they easily serve as test cases to verify implementations of algorithms. Third, these environments embody a wide variety of problems useful for the evaluation of algorithms. The challenge in developing learning algorithms is that they should be general; if a learning algorithm performs well on a wide variety of problems, a useful algorithm has been found.

The state and action spaces can be discrete or continuous. This is indicated in the titles below in this order; for example, "continuous/discrete" indicates a continuous state space and a discrete action space.

A popular type of environment is the grid world. A grid world is an environment that typically consists of a (two-dimensional) grid, i.e., the states, on which the agent moves. There are usually four actions: up, down, left, and right; sometimes there are eight, including the diagonal directions. If the agent is near the edge of the grid and the action would make it leave the grid, the state remains unchanged; in other words, the agents cannot leave the grid. Often the grid world is a maze, i.e., there is a start state and a goal or a terminal state. In the case of a maze, the goal state must be reached as quickly as possible. Therefore there is a reward of -1 in each step until the goal state is reached, and the learning task is episodic and undiscounted ($\gamma = 1$). At first glance, it would also make sense to give out zero rewards during the episode and a positive one when the goal is reached; however, such rewards provide no incentive to reach the goal state as quickly as possible.

Usually there are complications built into the environment, or additional tasks defined via rewards must be performed. In this manner, grid worlds become a versatile class of environments and are well-suited to elucidate certain behaviors of algorithms.

2.4.1 Simple Grid World (Discrete/Discrete)

Implement a 4×4 grid world with two terminal states in the upper left and lower right corners, resulting in 14 non-terminal states (see [23, Example 4.1]. The four actions $\mathcal{A} = \{\text{up}, \text{down}, \text{left}, \text{right}\}$ act deterministically, the discount factor is $\gamma = 1$, and the reward is always equal to -1. Ensure that a maximum number of time steps can be specified.

2.4.2 Windy Grid World (Discrete/Discrete)

Windy Grid World is [23, Example 6.5]. The underlying grid has size 10×7, see Figure 2.1. The start is in the middle of the first column, and the goal is in the middle of the eighth column. The four actions are up, down, left, and right, and the learning task is an undiscounted ($\gamma = 1$) episodic task. As is usual in mazes, there is a reward of -1 in each time step until the goal or terminal state is reached.

This grid world is called windy because of its complication of a wind that sometimes displaces the agent (deterministically). In the middle region of the grid, the next states are shifted upward by the wind by one or two cells as

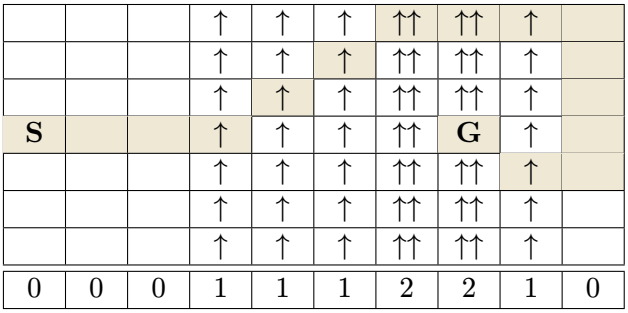


Figure 2.1: Windy Grid World. The colored cells indicate an optimal policy, resulting in an episode of length 15.

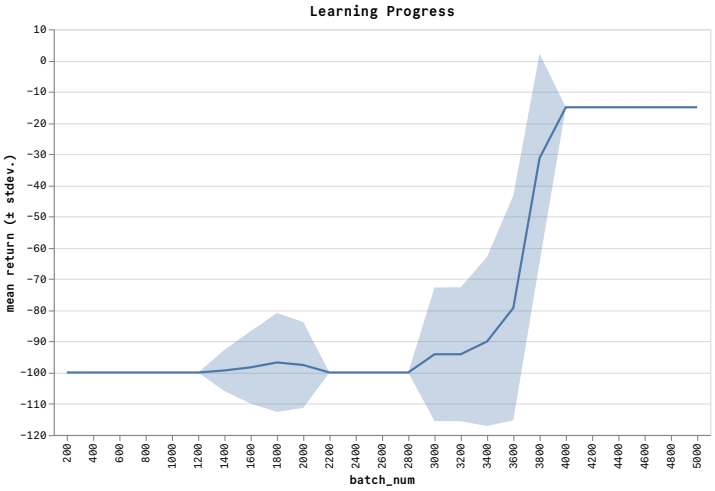


Figure 2.2: Learning progress of Q-learning for Windy Grid World.

indicated in the bottom row in the figure. The number below each column indicates the number of cells shifted upward. For example, if the agent is located to the right of the goal and the action is going left, then the agent is taken to the cell just above the goal. Note that the agent never leaves the grid world (as usual in grid worlds), even if it is displaced by the wind.

Learning progress is shown in Figure 2.2.

2.4.3 Cliff Walking (Discrete/Discrete)

Cliff Walking is another grid world [23, Example 6.6]. The underlying grid has size 12×4, see Figure 2.3. The start is in the bottom left, and the goal is in

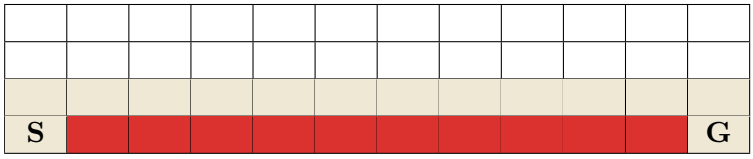


Figure 2.3: Cliff Walking. The cliff is located at the bottom between the start and goal states. The colored cells indicate an optimal policy, resulting in an episode of length 13.

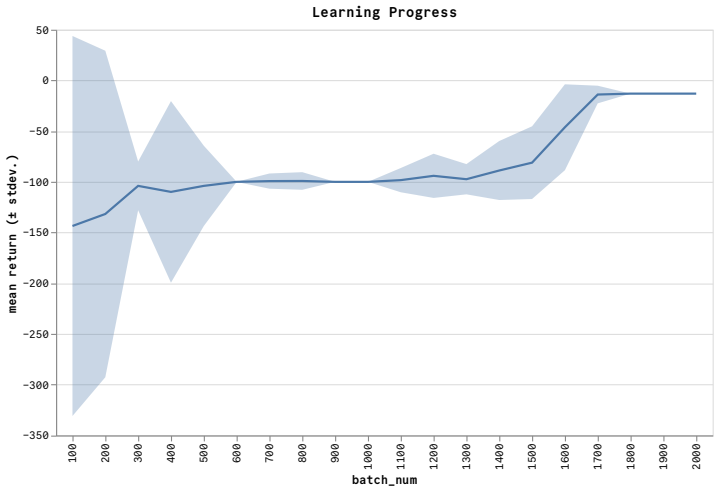


Figure 2.4: Learning progress of Q-learning for Cliff Walking.

the bottom right. Between the start and goal there is a cliff, representing the complication of this environment. The four actions are up, down, left, and right, and the learning task is an undiscounted ($\gamma = 1$) episodic task. The reward is -1 as usual in mazes, but falling down the cliff incurs a reward of -100 and instantly teleports the agent back to the start state.

Learning progress is shown in Figure 2.4.

2.4.4 Frozen Lake (Discrete/Discrete)

Frozen Lake[1] is a stochastic grid world. There are two versions regarding the size of the grid, namely 4×4 (see Figure 2.5) and 8×8 (see Figure 2.6). The start state is in the top left, and the goal state (a terminal state) is in the bottom right. The four actions are up, down, left, and right, and the learning task is an

[1] https://www.gymlibrary.dev/environments/toy_text/frozen_lake/

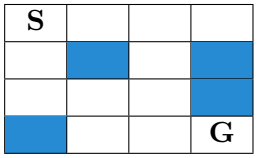


Figure 2.5: Frozen Lake 4×4. The colored cells are holes in the ice.

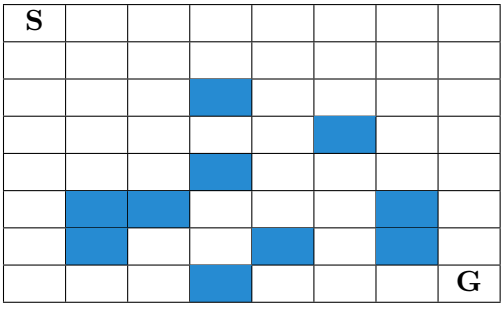


Figure 2.6: Frozen Lake 8×8. The colored cells are holes in the ice.

undiscounted ($\gamma = 1$) episodic task. There is a reward of $+1$ when reaching the goal, otherwise the rewards are zero. The complications are the holes in the ice, which are terminal states.

Another complication, which can be used optionally, is slippery ice. This is a complication that introduces lots of randomness. In the slippery version, the intended direction is followed with probability $1/3$ only. Otherwise, the agent will move in either direction perpendicular to the intended direction with equal probability of $1/3$ in both directions. For example, if the action is up, then the actions up, left, and right all have probability $1/3$.

Learning progresses are shown in Figure 2.7 and Figure 2.8.

2.4.5 Rock Paper Scissors (Discrete/Discrete)

Rock paper scissors is a well-known simultaneous, zero-sum game for two players. In the first variant considered here, the opponent uses a fixed, stochastic policy. Each action (rock, paper, or scissors) is chosen with equal probability $1/3$. As usual, the task of the agent is to learn an optimal policy.

In the second variant, the opponent uses a stochastic policy that varies with time. At time $t \in \mathbb{N}$, the probabilities are

$$\mathbb{P}[\text{rock}] = \sin^2 \alpha t \cos^2 \beta t,$$
$$\mathbb{P}[\text{paper}] = \sin^2 \alpha t \sin^2 \beta t,$$

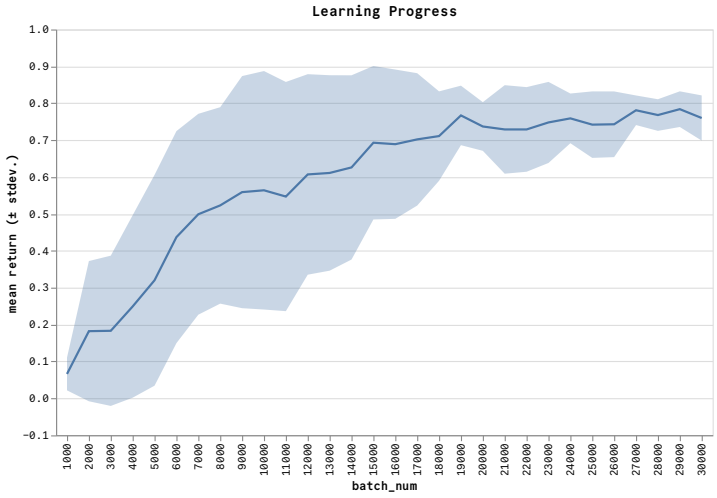


Figure 2.7: Learning progress of Q-learning for Frozen Lake (size 4×4, slippery).

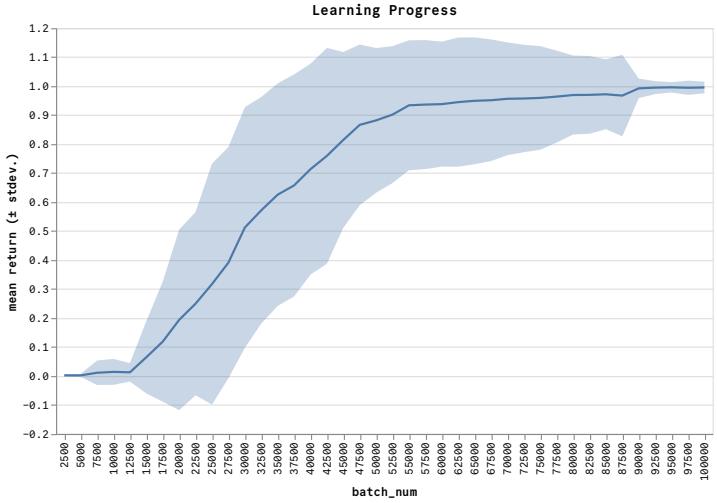


Figure 2.8: Learning progress of Q-learning for Frozen Lake (size 8×8, slippery).

$$\mathbb{P}[\text{scissors}] = \cos^2 \alpha t,$$

where $\alpha := 2\pi/100$ and $\beta := 2\alpha$. Again, the task of the agent is to learn an optimal policy.

2.4.6 Prisoner's Dilemma (Discrete/Discrete)

2.4.7 Blackjack (Discrete/Discrete)

Blackjack is the most widely played casino banking game. It is a descendent of the family of casino banking games known as "twenty-one," whose progenitor is recorded in Spain in the early 17th century. In the context of card games, a banking game is a gambling card game in which one or more players, the so-called punters, play against a banker or dealer who controls the game.

The objective of Blackjack is to draw cards such that the sum of their values is as great as possible without exceeding 21. An ace has a value of 1 or 11, whichever is more expedient to the player. All face cards (King, Queen, and Jack) have a value of 10. [23, Example 5.1] is the basic version of the game in which each player plays independently against the dealer and has only two choices, namely to "hit" or to "stick." In other versions, the player can also "double down," "split," or "surrender."

All cards are dealt from an infinite deck of cards, which means that there is no advantage in counting cards. At the start of a each game, two cards are dealt to both the player and the dealer. Both cards of the player are only known to the player, but one card of the dealer is dealt face up and the other face down.

If a player or the dealer has 21 immediately, it is called a "natural." If the player has a natural immediately, the game is a draw if the dealer also has a natural, or the player wins if the dealer does not have a natural. Later, if the player does not have a natural immediately, the player has two choices in each time step: he can request an additional card ("hits") or he can stop requesting additional cards ("sticks"). The episode continues until he sticks or exceeds 21 ("goes bust"). If he goes bust, he loses the game. Otherwise, if the sticks, it becomes the dealer's turn.

Whenever it is the dealer's turn, he also requests cards one by one, but his strategy or policy is fixed without any choice. The dealer sticks on any sum of 17 or greater and hits otherwise.

If the dealer goes bust, then the player wins. Otherwise, the result of the game (win, draw, or lose) and hence the terminal reward ($+1$, 0, -1, respectively) is determined by whose sum is closer to 21. All rewards are 0 except the terminal rewards. The learning task is an undiscounted ($\gamma = 1$) episodic task, and therefore the terminal rewards are the returns.

An ace that can count as 11 without going bust is called a "usable" ace. A usable ace always counts as 11.

Learning progress is shown in Figure 2.9.

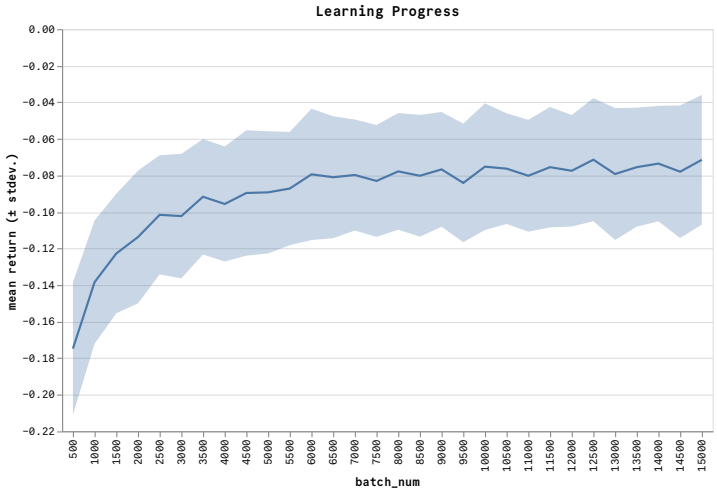


Figure 2.9: Learning progress of Q-learning for Blackjack.

2.4.8 Schnapsen (Discrete/Discrete)

The rules of Schnapsen[2] are available from the playing-card and board-game manufacturer Wiener Spielkartenfabrik Ferd. Piatnik & Söhne.

2.4.9 Autoregressive Trend Process (Continuous/Discrete)

Autoregressive trend processes such as the following serve as models for stock prices. We define the stochastic process

$$\begin{aligned} a_0 &:= 1, \\ b_0 &:= N(0,1), \\ a_k &:= a_{k-1} + b_{k-1} + \lambda N(0,1), \\ b_k &:= \kappa b_{k-1} + N(0,1). \end{aligned}$$

Here $N(\mu, \sigma^2)$ means drawing a random number from the normal distribution with mean μ and variance σ^2. We use $\lambda := 3$, $\kappa := 0.9$, and $k \in \{0, \dots, 99\}$. Having calculated 100 values $\{a_0, \dots, a_{99}\}$, the scaled values

$$p_k := \exp\left(\frac{a_k}{\max_{0 \le k < 100} a_k - \min_{0 \le k < 100} a_k}\right)$$

serve as simulated stock prices.

[2] https://www.piatnik.com/karten-spielregel

The actions in this episodic task are taking long, neutral, or short positions. The transactions costs are fractions of the stock prices; it is instructive to learn policies for cheap and expensive transaction costs.

2.5 Policies, Value Functions, and Bellman Equations

After the discussion of environments, rewards, returns, and episodes, we focus on concepts that underlie learning algorithms.

Learning optimal policies is the goal.

Definition 2.6 (policy)**.** A *policy* is a function $\mathcal{S} \times \mathcal{A}(s) \to [0,1]$. An agent is said to follow a policy π at time t, if $\pi(a|s)$ is the probability that the action $A_t = a$ is chosen if $S_t = s$.

Like the dynamics function p of the environment, a policy π is a function despite the notation that is reminiscent of a conditional probability. We denote the set of all policies by $\mathcal{P}$.

The following two functions, the (state-)value function and the action-value function, are useful for the agent because they indicate how expedient it is to be in a state or to be in a state and to take a certain action, respectively. Both functions depend on a given policy.

Definition 2.7 ((state-)value function)**.** The *value function* of a state s under a policy π is

$$v\colon \mathcal{P} \times \mathcal{S} \to \mathbb{R}, \qquad v_\pi(s) := \mathbb{E}_\pi[G_t \mid S_t = s],$$

i.e., it is the expected discounted return when starting in state s and following the policy π until the end of the episode.

Definition 2.8 (action-value function)**.** The *action-value function* of a state-action pair(s,a) under a policy π is

$$q\colon \mathcal{P} \times \mathcal{S} \times \mathcal{A}(s) \to \mathbb{R}, \qquad q_\pi(s,a) := \mathbb{E}_\pi[G_t \mid S_t = s,\ A_t = a],$$

i.e., it is the expected discounted return when starting in state s, taking action a, and then following the policy π until the end of the episode.

Recursive formulae such as (2.4) are fundamental throughout reinforcement learning and dynamic programming. We now use (2.4) to find a recursive formula for the value function v_π by calculating

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{2.5a}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \tag{2.5b}$$
$$= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r \mid s, a)\big(r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\big) \tag{2.5c}$$
$$= \sum_{a\in\mathcal{A}(s)} \pi(a|s) \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r \mid s, a)\big(r + \gamma v_\pi(s')\big) \qquad \forall s \in \mathcal{S}. \tag{2.5d}$$

This equation is called the Bellman equation for v_π, and it is fundamental to computing, approximating, and learning v_π. The solution v_π exists uniquely if $\gamma < 1$ or all episodes are guaranteed to terminate from all states $s \in \mathcal{S}$ under policy π.

In the case of a finite MDP, the optimal policy is defined as follows. We start by noting that state-value functions can be used to define a partial ordering over the policies: a policy π is defined to be better than or equal to a policy π' if its value function v_π is greater than or equal to the value function $v_{\pi'}$ for all states. We write

$$\pi \geq \pi' \quad :\iff \quad v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}.$$

An optimal policy is a policy that is greater than or equal to all other policies. The optimal policy may not be unique. We denote optimal policies by π_*.

Optimal policies share the *same* state-value and action-value functions. The optimal state-value function is given by

$$v_*\colon \mathcal{S} \to \mathbb{R}, \qquad v_*(s) := \max_{\pi\in\mathcal{P}} v_\pi(s),$$

and the optimal action-value function is given by

$$q_*\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}, \qquad q_*(s, a) := \max_{\pi\in\mathcal{P}} q_\pi(s, a),$$

These two functions are related by the equation

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s,\ A_t = a] \qquad \forall (s, a) \in \mathcal{S} \times \mathcal{A}(s). \tag{2.6}$$

Next, we find a recursion for these two optimal value functions similar to the Bellman equation above. Similarly to the derivation of (2.5), we calculate

$$v_*(s) = \max_{a\in\mathcal{A}(s)} q_{\pi_*}(s, a) \tag{2.7a}$$
$$= \max_{a\in\mathcal{A}(s)} \mathbb{E}_{\pi_*}[G_t \mid S_t = s,\ A_t = a] \tag{2.7b}$$
$$= \max_{a\in\mathcal{A}(s)} \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s,\ A_t = a] \tag{2.7c}$$
$$= \max_{a\in\mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s,\ A_t = a] \tag{2.7d}$$

$$= \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma v_*(s')) \qquad \forall s \in \mathcal{S}. \tag{2.7e}$$

The last two equations are both called the Bellman optimality equation for v_*. Analogously, two forms of the Bellman optimality equation for q_* are

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} q_*(S_{t+1}, a') \mid S_t = s,\ A_t = a] \tag{2.8a}$$

$$= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma \max_{a' \in \mathcal{A}(s)} q_*(s', a')) \qquad \forall (s, a) \in \mathcal{S} \times \mathcal{A}(s). \tag{2.8b}$$

One can try to solve the Bellman optimality equations for v_* or q_*; they are just systems of algebraic equations. If the optimal action-value function q_* is known, an optimal policy π_* is easily found; we are still considering the case of a *finite* MDP. However, there are a few reasons why this approach is seldomly expedient for realistic problems:

- The Markov property may not hold.
- The dynamics of the environment, i.e., the function p, must be known.
- The system of equations may be huge.

2.6 On-Policy and Off-Policy Learning

2.7 Policy Evaluation (Prediction)

Policy evaluation means that we evaluate how well a policy π does by computing its state-value function v_π. The term "policy evaluation" is common in dynamic programming, while the term "prediction" is common in reinforcement learning. In policy evaluation, a policy π is given and its state-value function v_π is calculated.

Instead of solving the Bellman equation (2.5) for v_π directly, we follow an iterative approach. Starting from an arbitrary initial approximation $v_0 \colon \mathcal{S} \to \mathbb{R}$ (whose terminal state must have the value 0), we use (2.5) to define the iteration

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \tag{2.9a}$$

$$= \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma v_k(s')) \tag{2.9b}$$

for $v_{k+1} \colon \mathcal{S} \to \mathbb{R}$. This iteration is called iterative policy evaluation.

If $\gamma < 1$ or all episodes are guaranteed to terminate from all states $s \in \mathcal{S}$ under policy π, then this operator is a contraction and hence the approximations v_k converge to the state-value function v_π as $k \to \infty$, since v_π is the fixed point by the Bellman equation (2.5) for v_π.

The updates performed in (2.9) and in dynamic programming in general are called expected updates, since the expected value over all possible next states is computed in contrast to using a sample next state.

Algorithm 2 shows how this iteration can be implemented with updates performed in place. It also shows a common termination condition that uses the maximum norm and a prescribed accuracy threshold.

Algorithm 2 iterative policy evaluation for approximating $\mathbf{v} \approx v_\pi$ given $\pi \in \mathcal{P}$.

Initialization:
choose the accuracy threshold $\delta \in \mathbb{R}^+$,
initialize the vector $\mathbf{v}$ of length $|\mathcal{S}|$ arbitrarily
except that the value of the terminal state is 0.

repeat
  $d := 0$
  for all $s \in \mathcal{S}$ **do**
    $w := v_s$ ▷ save the old value
    $v_s := \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma v_{s'})$
    $d := \max(d, |v_s - w|)$
  end for
until $d < \delta$

return v

2.8 Policy Improvement

Having seen how we can evaluate a policy, we now discuss how to improve it. To do so, the value functions show their usefulness. For now, we assume that the policies we consider here are deterministic.

Similarly to (2.6), the action value of selecting action a and then following policy π can be written as

$$q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s,\ A_t = a] \tag{2.10a}$$

$$= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma v_\pi(s')) \qquad \forall (s, a) \in \mathcal{S} \times \mathcal{A}(s). \quad (2.10b)$$

This formula helps us determine if the action $\pi'(s)$ of another policy π' is an improvement over $\pi(s)$ in this time step. In order to be an improvement in this time step, the inequality

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

must hold. If this inequality holds, we also expect that selecting $\pi'(s)$ instead of $\pi(s)$ every time the state s occurs is an improvement. This is the subject of the following theorem.

Theorem 2.9 (policy improvement theorem)**.** *Suppose π and π' are two deterministic policies such that*

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \qquad \forall s \in \mathcal{S}.$$

Then

$$\pi' \geq \pi,$$

i.e., the policy π' is greater than or equal to π.

Proof. By the definition of the partial ordering of policies, we must show that

$$v_{\pi'}(s) \geq v_\pi(s) \qquad \forall s \in \mathcal{S}.$$

Using the assumption and (2.10), we calculate

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s,\ A_t = \pi'(s)] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) \mid S_{t+1}] \mid S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \mid S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) \mid S_t = s] \\
&\ \ \vdots \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \mid S_t = s] \\
&= v_{\pi'}(s),
\end{aligned}$$

which concludes the proof. □

In addition to making changes to the policy in single states, we can define a new, greedy policy π' by selection the action that appears best in each state according to a given action-value function q_π. This greedy policy is given by

$$\pi'(s) :\in \arg\max_{a\in\mathcal{A}(s)} q_\pi(s,a), \tag{2.11a}$$

$$= \arg\max_{a\in\mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s,\ A_t = a] \tag{2.11b}$$

$$= \arg\max_{a\in\mathcal{A}(s)} \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r \mid s, a)\big(r + \gamma v_\pi(s')\big). \tag{2.11c}$$

Any ties in the arg max are broken in a random manner. By construction, the policy π' satisfies the assumption of Theorem 2.9, implying that it is better than or equal to π. This process is called policy improvement. We have created a new policy that improves on the original policy by making it greedy with respect to the value function of the original policy.

In the case that the $\pi' = \pi$, i.e., the new, greedy policy is as good as the original one, the equation $v_\pi = v_{\pi'}$ follows, and using the definition (2.11) of π', we find

$$v_{\pi'}(s) = \max_{a\in\mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t = s,\ A_t = a] \tag{2.12a}$$

$$= \max_{a\in\mathcal{A}(s)} \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r \mid s, a)\big(r + \gamma v_{\pi'}(s')\big) \qquad \forall s \in \mathcal{S}, \tag{2.12b}$$

which is the Bellman optimality equation (2.7). Since $v_{\pi'}$ satisfies the optimality equation (2.7), $v_{\pi'} = v_*$ holds, meaning that both π and π' are optimal policies. In other words, policy improvement yields a strictly better policy unless the original policy is already optimal.

So far we have considered only deterministic policies, but these ideas can be extended to stochastic policies, and Theorem 2.9 also holds for stochastic policies. When defining the greedy policy, all maximizing actions can be assigned some nonzero probability.

2.9 Policy Iteration

Policy iteration is the process of using policy evaluation and policy improvement to define a sequence of monotonically improving policies and value functions. We start from a policy π_0 and evaluate it to find its state-value function v_{π_0}. Using v_{π_0}, we use policy improvement to define a new policy π_1. This policy is evaluated, and so forth, resulting in the sequence

$$\pi_0 \overset{\text{eval.}}{\longrightarrow} v_{\pi_0} \overset{\text{impr.}}{\longrightarrow} \pi_1 \overset{\text{eval.}}{\longrightarrow} v_{\pi_1} \overset{\text{impr.}}{\longrightarrow} \pi_2 \overset{\text{eval.}}{\longrightarrow} v_{\pi_2} \overset{\text{impr.}}{\longrightarrow} \cdots \overset{\text{impr.}}{\longrightarrow} \pi_* \overset{\text{eval.}}{\longrightarrow} v_*.$$

Unless a policy π_k is already optimal, it is a strict improvement over the previous policy π_{k-1}. In the case of a finite MDP, there is only a finite number of policies, and hence the sequence converges to the optimal policy and value function within a finite number of iterations.

A policy-iteration algorithm is shown in Algorithm 3. Each policy evaluation is started with the value function of the previous policy, which speeds up convergence. Note that the update of v_s has changed.

2.10 Value Iteration

A time-consuming property in Algorithm 3 is the fact that the repeat loop for policy evaluation is nested within the outer loop. Nesting these two loops may be quite time consuming. (The other inner loop is a for loop with a fixed number of iterations.) This suggests to try to get rid of these two nested loops while still guaranteeing convergence. An important simple case is to perform only one iteration policy evaluation, which makes it possible to combine policy evaluation and improvement into one loop. This algorithm is called value iteration, and it can be shown to convergence under the same assumptions that guarantee the existence of v_*.

Turning the fixed-point equation (2.12) into an iteration, value iteration becomes the update

$$\begin{aligned} v_{k+1}(s) &:= \max_{a \in \mathcal{A}(s)} \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s,\ A_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma v_k(s')). \end{aligned}$$

The algorithm is shown in Algorithm 4. Note that the update of v_s has changed again, now incorporating taking the maximum from the policy-improvement part.

2.11 Bibliographical and Historical Remarks

The classic textbook on dynamic programming is [35] by Richard Bellman. The most influential introductory textbook on RL is [23]. An excellent summary of the theory of Markov decision processes is [27].

Algorithm 3 policy iteration for calculating $\mathbf{v} \approx v_*$ and $\pi \approx \pi_*$.

Initialization:
choose the accuracy threshold $\delta \in \mathbb{R}^+$,
initialize the vector $\mathbf{v}$ of length $|\mathcal{S}|$ arbitrarily
  except that the value of the terminal state is 0,
initialize the vector π of length $|\mathcal{S}|$ arbitrarily.

loop
  policy evaluation:
  repeat
    $d := 0$
    for all $s \in \mathcal{S}$ **do**
      $w := v_s$ ▷ save the old value
      $v_s := \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, \pi(s))(r + \gamma v_{s'})$
        ▷ note the change: π_s is an optimal action
      $d := \max(d, |v_s - w|)$
    end for
  until $d < \delta$

  policy improvement:
  policyIsStable := true
  for all $s \in \mathcal{S}$ **do**
    old_action := $\pi[s]$ ▷ save the old value
    $\pi_s :\in \arg\max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma v_{s'})$
      ▷ $\pi_s :\in \arg\max_{a \in \mathcal{A}(s)} q(s, a)$
    if old_action $\neq \pi(s)$ **then**
      policy_is_stable := false
    end if
  end for
  if policy_is_stable **then**
    return $\mathbf{v} \approx v_*$ and $\pi \approx \pi_*$
  end if
end loop

Algorithm 4 value iteration for calculating $\mathbf{v} \approx v_*$ and $\pi \approx \pi_*$.

Initialization:
choose the accuracy threshold $\delta \in \mathbb{R}^+$,
initialize the vector $\mathbf{v}$ of length $|\mathcal{S}|$ arbitrarily
  except that the value of the terminal state is 0,
initialize the vector π of length $|\mathcal{S}|$ arbitrarily.

policy evaluation and improvement:
repeat
  $d := 0$
  for all $s \in \mathcal{S}$ **do**
    $w := v_s$ ▷ save the old value
    $v_s := \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma v_{s'})$
    ▷ note the change
    $d := \max(d, |v_s - w|)$
  end for
until $d < \delta$

calculate deterministic policy:
for all $s \in \mathcal{S}$ **do**
  $\pi_s :\in \arg\max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a)(r + \gamma v_{s'})$
  ▷ $\pi_s :\in \arg\max_{a \in \mathcal{A}(s)} q(s, a)$
end for

return $\mathbf{v} \approx v_*$ and $\pi \approx \pi_*$

2.12 Exercises

2.12.1 Applications and Environments

Exercise 2.1. Think of another (preferably creative) application of reinforcement learning. Specify the environments, agents, states, actions, and rewards. Which assumptions are needed to satisfy the Markov property?

Exercise 2.2. Try to find a goal-directed learning task that cannot be represented by a Markov decision process.

Exercise 2.3 (Simple Grid World)**.** Implement Simple Grid World (see Section 2.4.1).

Exercise 2.4 (Windy Grid World)**.** Implement Windy Grid World (see Section 2.4.2).

Exercise 2.5 (Cliff Walking)**.** Implement Cliff Walking (see Section 2.4.3).

Exercise 2.6 (Frozen Lake)**.** Implement Frozen Lake (see Section 2.4.4).

Exercise 2.7 (variants of Frozen Lake)**.**

1. In Frozen Lake (see Section 2.4.4), there is a positive reward only when the goal is reached. However, in mazes, the rewards are usually -1 until the goal is reached. Implement the usual rewards as a variant and investigate any differences when learning the two variants.

2. Another variant is that holes do not end the episode, but incur a large negative reward and teleport the agent to the start. Implement this variant and investigate any differences in learning.

Exercise 2.8 (Rock paper scissors)**.** Implement Rock paper scissors (see Section 2.4.5).

Exercise 2.9 (Blackjack)**.** Implement Blackjack (see Section 2.4.7).

Exercise 2.10 (Schnapsen)**.** $*$ Implement Schnapsen (see Section 2.4.8).

Exercise 2.11 (autoregressive trend process)**.** Implement an autoregressive trend process (see Section 2.4.9).

Exercise 2.12 (ϵ-greedy action selection)**.** Assume that ϵ-greedy action selection is used and that there is a single greedy action.

1. Suppose $|\mathcal{A}| = 4$ and $\epsilon = 1/5$. What is the probability that the greedy action is selected?

2. Give a formula for calculating the probability of selecting the greedy action for any $|\mathcal{A}|$ and any ϵ.

2.12.2 Multi-Armed Bandits

Exercise 2.13 (multi-armed bandits with ϵ-greedy action selection (programming))**.** You play against a 10-armed bandit, where at the beginning of each episode the true value $q_*(a)$, $a \in \{1, \dots, 10\}$, of each of the 10 actions is chosen to be normally distributed with mean zero and unit variance. The rewards after choosing action/bandit a are normally distributed with mean $q_*(a)$ and unit variance. Using the simple bandit algorithm and ϵ-greedy action selection, you have 1000 time steps or tries in each episode to maximize the average reward starting from zero knowledge about the bandits.

Which value of ϵ maximizes the average reward? Which value of ϵ maximizes the percentage of optimal actions taken?

Exercise 2.14 (multi-armed bandits with upper-confidence-bound action selection (programming))**.** This exercise is the same as in Exercise 2.13, but now the actions

$$A_t :\in \arg\max_a \left(Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right)$$

are selected according to the upper-confidence bound.

Which value of c yields the largest average reward?

Exercise 2.15 (multi-armed bandits with soft-max action selection (programming))**.** This exercise is the same as Exercise 2.13, but now the actions $A_t \in \mathcal{A} = \{1, \dots, |\mathcal{A}|\}$ are selected with probability

$$\mathbb{P}[a] = \frac{\exp(Q_t(a)/\tau)}{\sum_{i=1}^{|\mathcal{A}|} \exp(Q_t(i)/\tau)},$$

where the parameter τ is called the temperature. This probability distribution is called the soft-max or Boltzmann distribution.

What are the effects of low and high temperatures, i.e., how does the temperature influence the probability distribution all else being equal? Which value of τ yields the largest average reward?

2.12.3 Step Sizes

Exercise 2.16 (harmonic step sizes)**.** Show that the step sizes

$$\alpha_n := \frac{1}{an + b}, \qquad a, b \in \mathbb{R},$$

(where $a \in \mathbb{R}^+$ and $b \in \mathbb{R}$ are chosen such that $an+b > 0$) satisfy the convergence conditions

$$\sum_{n=1}^{\infty} \alpha_n = \infty, \qquad \sum_{n=1}^{\infty} \alpha_n^2 < \infty.$$

Exercise 2.17 (unbiased step sizes)**.** We use the iteration

$$\begin{aligned} Q_1 &\in \mathbb{R}, \\ Q_{n+1} &:= Q_n + \alpha_n(R_n - Q_n), \qquad n \geq 1, \end{aligned}$$

to estimate Q_n using R_n, where

$$\alpha_n := \frac{\alpha}{\beta_n}, \qquad \alpha \in (0,1), \quad n \geq 1,$$

and

$$\begin{aligned} \beta_0 &:= 0, \\ \beta_n &:= \beta_{n-1} + \alpha(1 - \beta_{n-1}), \qquad n \geq 1. \end{aligned}$$

Show that the iteration for Q_n above yields an exponential recency-weighted average *without initial bias* (i.e., the Q_n do not depend on the initial value Q_1).

2.12.4 Basic Definitions

Exercise 2.18 (returns and episodes)**.** Suppose $\gamma := 1/2$ and the rewards $R_1 := 1$, $R_2 := -1$, $R_3 := 2$, $R_4 := -1$, and $R_5 := 2$ are received in an episode with length $T := 5$. What are $G_0, \ldots, G_5$?

Exercise 2.19 (returns and episodes)**.** Suppose $\gamma := 0.9$ and the reward sequence starts with $R_1 := -1$ and $R_2 := 2$ and is followed by an infinite sequence of 1s. What are G_0, G_1, and G_2?

Exercise 2.20 (change of return)**.** In episodic tasks and in continuing tasks, how does the return G_t change if a constant c is added to all rewards R_t? How does it change if all rewards R_t are multiplied by a constant c?

2.12.5 Dynamic Programming

Exercise 2.21 (equation for v_π)**.** Give an equation for v_π in terms of q_π and π.

Exercise 2.22 (equation for q_π)**.** Give an equation for q_π in terms of v_π and the four-argument p.

Exercise 2.23 (Bellman equation for q_π)**.** Analogous to the derivation of the Bellman equation for v_π, derive the Bellman equation for q_π.

Exercise 2.24 (equation for v_*)**.** Give an equation for v_* in terms of q_*.

Exercise 2.25 (equation for q_*)**.** Give an equation for q_* in terms of v_* and the four-argument p.

Exercise 2.26 (equation for π_*)**.** Give an equation for π_* in terms of q_*.

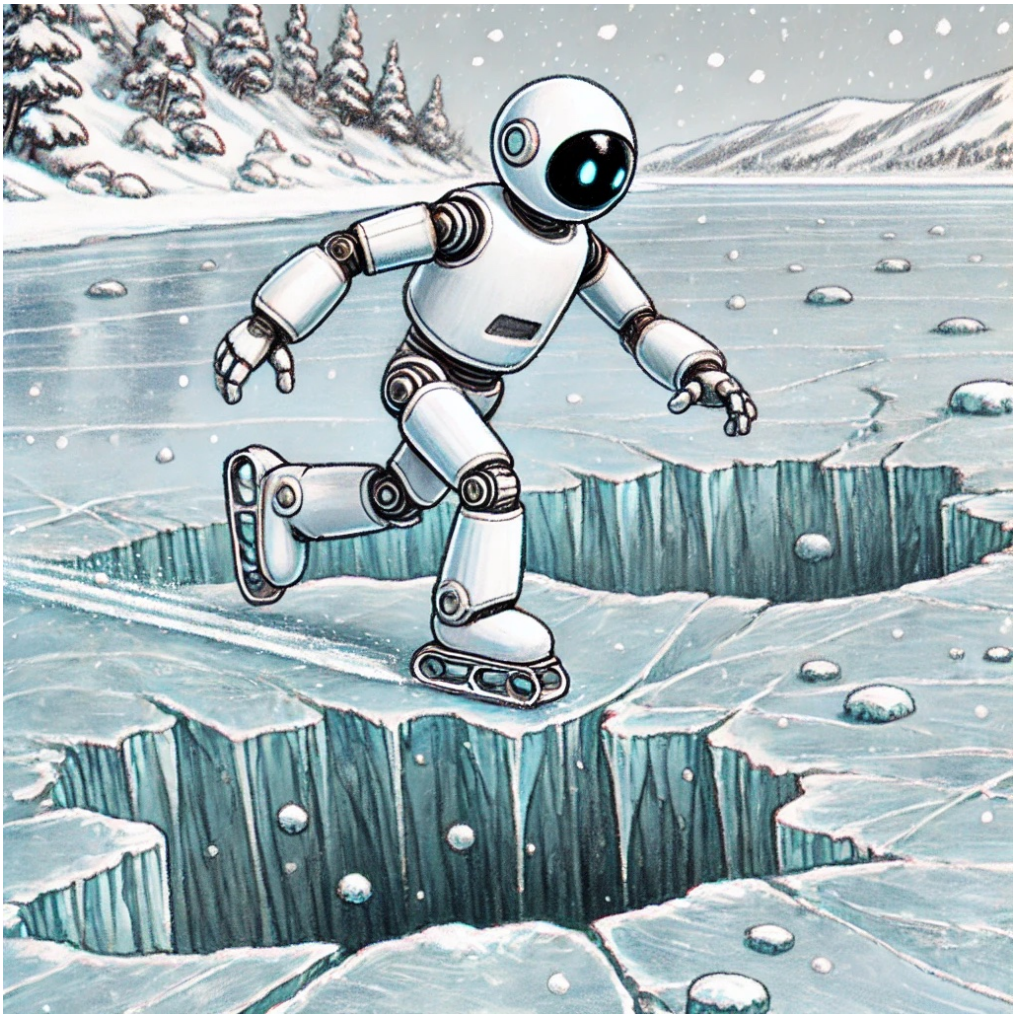
Exercise 2.27 (equation for π_*)**.** Give an equation for π_* in terms of v_* and the four-argument p.

Exercise 2.28 (update rule for q_π)**.** Using the Bellman equation for q_π (see Exercise 2.23), find an update rule for approximations q_{k+1} of q_π (in terms of q_k, π, and p) analogous to the update rule for v_{k+1}.

Exercise 2.29 (iterative policy evaluation (programming))**.** Implement iterative policy evaluation and use it to estimate v_π for the grid world in Exercise 2.3, where π is the equiprobable random policy.

Exercise 2.30 (policy iteration (programming))**.** Implement policy iteration and use it to estimate π_* for the grid world in Exercise 2.3.

Exercise 2.31 (value iteration (programming))**.** Implement value iteration and use it to estimate π_* for the grid world in Exercise 2.3.



Chapter 8

Policy-Gradient Methods

8.1 Introduction

A large class of reinforcement-learning methods are action-value methods, i.e., methods that calculate action values and then choose the best action based on these action values. On the other hand, we present methods for calculating policies more directly in this chapter. The policies here are parameterized, i.e., we write them in the form

$$\pi\colon \mathcal{A}(s) \times \mathcal{S} \times \Theta \to [0,1],$$
$$\pi_\theta(a \mid s) := \pi(a \mid s, \theta) := \mathbb{P}\{A_t = a \mid S_t = s,\ \theta_t = \theta\},$$

where the parameter $\theta \in \Theta \subset \mathbb{R}^{d'}$ is a vector. We seek a parameter that corresponds to an optimal policy.

Commonly, the parameters are learned such that a scalar performance measure called

$$J\colon \Theta \to \mathbb{R},$$

defined on the parameters, is maximized. A leading example is the definition

$$J(\theta) := \mathbb{E}[v_{\pi_\theta}(S_0) \mid S_0 \sim \iota],$$

where $S_0 \sim \iota$ is the initial state chosen according to the probability distribution ι.

The general assumption is that the policy π and the performance measure J are differentiable with respect to θ such that gradient based optimization can be employed. Such methods are called *policy-gradient methods*. The performance can be maximized for example by using stochastic gradient ascent with respect to the performance measure J, i.e., we define the iteration

$$\theta_{t+1} := \theta_t + \alpha_t \mathbb{E}[\nabla_\theta J(\theta_t)].$$

The expected value of the gradient of the performance measure J in the last term is usually approximated.

The question whether action-value or policy-gradient methods are preferable depends on the problem. It may be the case that the action-value function has a simpler structure and is therefore easier to learn, or it may be the case that the policy itself has a simpler structure.

8.2 Finite and Infinite Action Sets

8.2.1 Finite Action Sets

If the action sets $\mathcal{A}(s)$ are finite, then a common form of the policy is based on the so-called *preference function*

$$h \colon \mathcal{S} \times \mathcal{A}(s) \times \Theta \to \mathbb{R}.$$

The preferences of state-action pairs (s, a) are translated into probabilities and hence into a policy via the exponential soft-max function

$$\pi(a \mid s, \theta) := \frac{\exp(h(s, a, \theta))}{\sum_{a' \in \mathcal{A}(s)} \exp(h(s, a', \theta))}.$$

Many choices for the representation of the preference functions are possible. Two popular ones are the following.

- The preference function is an artificial neural network. Then the parameter vector $\theta \in \Theta \subset \mathbb{R}^{d'}$ contains all weights and biases of the artificial neural network.

- The preference function has the linear form

  $$h(s, a, \theta) := \theta^\top x(s, a),$$

  where the *feature function*

  $$x \colon \mathcal{S} \times \mathcal{A}(s) \to \mathbb{R}^{d'}$$

  yields the feature vectors.

8.2.2 Infinite Action Sets

If the action sets $\mathcal{A}(s)$ are infinite, it is possible to simplify the problem of learning the probabilities of all actions by reducing it to learning the parameters of a probability distribution. The parameters of the distribution are represented by functions. For example, we define a policy of the form

$$\pi(a \mid s, \theta) := \frac{1}{\sqrt{2\pi}\sigma(s,\theta)} \exp\left(-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}\right),$$

where now the two functions $\mu\colon \mathcal{S} \times \Theta \to \mathbb{R}$ and $\sigma\colon \mathcal{S} \times \Theta \to \mathbb{R}^+$ need to be learned. To do that, we split the parameter vector $\theta \in \Theta$ into two vectors θ_μ and θ_σ, i.e., $\theta = (\theta_\mu, \theta_\sigma)^\top$, and write the functions μ and σ as a linear and a positive function

$$\begin{aligned}\mu(s,\theta) &:= \theta_\mu^\top x_\mu(s),\\ \sigma(s,\theta) &:= \exp(\theta_\sigma^\top x_\sigma(s)),\end{aligned}$$

respectively, where the features x_μ and x_σ are vector valued functions as usual.

Representing policies as the soft-max of preferences for actions makes it possible to approximate deterministic policies, which is not immediately possible when using ϵ-greedy policies. If the optimal policy is deterministic, the preferences of the optimal actions become unbounded, at least if this behavior is allowed by the kind of parameterization used.

Action preferences can represent optimal stochastic policies well in the sense that the probabilities of actions may be arbitrary. This is in contrast to action-value methods and it is an important feature whenever the optimal policy is stochastic such as in rock-paper-scissors and poker.

8.3 The Policy-Gradient Theorem

Before stating the policy-gradient theorem, we define the (discounted) state distribution. In the case of an episodic environment or learning task, we consider discount rates $\gamma < 1$. In the case of a continuing environment or learning task, it can be shown that a discount rate $\gamma < 1$ only results in the factor $1/(1-\gamma)$ in the performance measure and hence we assume that $\gamma = 1$ in this case without loss of generality.

Definition 8.1 (discounted state distribution)**.** The *discounted state distribution*

$$\mu_\pi\colon \mathcal{S} \to [0,1],$$

$$\mu_\pi(s) := \mathbb{E}_{\text{all episodes}}[\lim_{t\to\infty} \mathbb{P}\{S_t = s \mid S_0 \sim \iota,\ A_0, \dots, A_{t-1} \sim \pi\}]$$

is the probability of being in state s in all episodes under a given policy $\pi \in \mathcal{P}$ and a given initial state distribution ι and discounted by γ in the episodic case.

In order to simplify notation, we write μ_π instead of $\mu_{\pi,\iota,\gamma}$.

By definition, $\mu_\pi(s) \geq 0$ for all $s \in \mathcal{S}$ and $\sum_{s\in\mathcal{S}} \mu_\pi(s) = 1$. We discount the state distribution by the discount rate γ, because this is the form that is necessary for the calculations in Theorem 8.3 below. It is also consistent with the appearance of the discount rate in the definitions of the state- and action-value functions, which are also used in the calculations in the theorem.

If the environment or learning task is continuing, the state distribution is just the stationary distribution under the policy π. If the environment or learning task is episodic, however, the distribution of the initial distribution of the states plays a role as seen in the following lemma, which gives the state distribution in both cases.

Lemma 8.2 (discounted state distribution)**.** *If the environment is continuing, the* state distribution *is the stationary distribution under the policy π.*

If the environment is episodic, the discounted state distribution *is given by*

$$\mu_\pi(s) = \frac{\eta(s)}{\sum_{s'\in\mathcal{S}} \eta(s')} \qquad \forall s \in \mathcal{S},$$

where the $\eta(s)$ are the solution of the system of equations

$$\eta(s) = \iota(s) + \gamma \sum_{s'\in\mathcal{S}} \eta(s') \sum_{a\in\mathcal{A}(s)} \pi(a|s') p(s \mid s', a) \qquad \forall s \in \mathcal{S},$$

where $\iota\colon \mathcal{S} \to [0,1]$ is the initial distribution of the states in an episode.

To simplify notation, we write η instead of η_π or $\eta_{\pi,\iota,\gamma}$.

Proof. We start by noting that $\eta(s)$ is the average number of time steps spent in state s over all episodes discounted by γ. It consists of two terms, namely the probability $\iota(s)$ to start in state s and the discounted average number of times the state s occurs coming from all other states $s' \in \mathcal{S}$.

This linear system of equations has a unique solution, yielding the $\eta(s)$. In order to find the state distribution $\mu_\pi(s)$, the $\eta(s)$ must be scaled by the mean length

$$L := \sum_{s\in\mathcal{S}} \eta(s) \tag{8.1}$$

of all episodes. □

The following theorem [36] is fundamental for policy-gradient methods. In the continuing case, the performance measure is

$$r(\pi_\theta) := \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t \mid S_0 \sim \iota,\ A_0, \dots, A_{t-1} \sim \pi_\theta],$$

which is assumed to exist and to be independent of the initial state distribution ι, i.e., the stochastic process defined by the policy π_θ and the transition probability p is assumed to be ergodic.

Theorem 8.3 (policy-gradient theorem)**.** *Suppose that the performance measure is defined as*

$$J(\theta) := \begin{cases} \mathbb{E}[v_{\pi_\theta}(S_0) \mid S_0 \sim \iota], & \textit{episodic environment,} \\ r(\pi_\theta), & \textit{continuing environment,} \end{cases}$$

where $S_0 \sim \iota$ is the initial state chosen according to the probability distribution ι. Then its gradient is given by

$$\nabla_\theta J(\theta) = L \sum_{s\in\mathcal{S}} \mu_{\pi_\theta}(s) \sum_{a\in\mathcal{A}(s)} q_{\pi_\theta}(s,a) \nabla_\theta \pi_\theta(a \mid s, \theta),$$

where

$$L := \begin{cases} \textit{mean episode length}, & \textit{episodic environment}, \\ 1, & \textit{continuing environment} \end{cases}$$

and μ is the state distribution.

Proof. We prove the episodic case first and start by differentiating the state-value function v_{π_θ}. By the definitions of the value functions v and q, we have

$$\nabla_\theta v_{\pi_\theta}(s) = \nabla_\theta \Big(\sum_{a\in\mathcal{A}(s)} \pi(a \mid s, \theta) q_{\pi_\theta}(s,a) \Big) \qquad \forall s \in \mathcal{S}.$$

By (2.10), we find

$$\begin{aligned} \nabla_\theta v_{\pi_\theta}(s) = \sum_{a\in\mathcal{A}(s)} \Big(& \nabla_\theta \pi(a \mid s, \theta) q_{\pi_\theta}(s,a) \\ & + \pi(a \mid s, \theta) \nabla_\theta \sum_{s'\in\mathcal{S}} \sum_{r\in\mathcal{R}} p(s', r \mid s, a)(r + \gamma v_{\pi_\theta}(s')) \Big) \qquad \forall s \in \mathcal{S}, \end{aligned}$$

which simplifies to

$$\nabla_\theta v_{\pi_\theta}(s) = \sum_{a\in\mathcal{A}(s)} \Big(\nabla_\theta \pi(a \mid s,\theta) q_{\pi_\theta}(s,a) + \gamma\pi(a \mid s,\theta) \sum_{s'\in\mathcal{S}} p(s', r \mid s, a) \nabla_\theta v_{\pi_\theta}(s')\Big) \qquad \forall s \in \mathcal{S}.$$

Performing a second time step by applying the recursive formula we just found to $\nabla_\theta v_{\pi_\theta}(s')$, we find

$$\nabla_\theta v_{\pi_\theta}(s) = \sum_{a\in\mathcal{A}(s)} \Big(\nabla_\theta \pi(a \mid s,\theta) q_{\pi_\theta}(s,a) + \gamma\pi(a \mid s,\theta) \sum_{s'\in\mathcal{S}} p(s', r \mid s, a) \sum_{a'\in\mathcal{A}(s)} \Big(\nabla_\theta \pi(a' \mid s',\theta) q_{\pi_\theta}(s',a') + \gamma\pi(a' \mid s',\theta) \sum_{s''\in\mathcal{S}} p(s'', r \mid s', a') \nabla_\theta v_{\pi_\theta}(s'')\Big)\Big) \qquad \forall s \in \mathcal{S}.$$

Hence the recursive expansion of this formula can be written as

$$\nabla_\theta v_{\pi_\theta}(s) = \sum_{s'\in\mathcal{S}} \sum_{k=0}^{\infty} \gamma^k \mathbb{P}\{s' \mid s, k, \pi\} \sum_{a\in\mathcal{A}(s')} \nabla_\theta \pi(a \mid s',\theta) q_{\pi_\theta}(s',a),$$

where $\mathbb{P}\{s' \mid s, k, \pi\}$ is the probability of transitioning to state s' after following policy π for k steps after starting from state s.

The gradient of the performance measure J thus becomes

$$\begin{aligned}
\nabla_\theta J(\theta) &:= \nabla_\theta \mathbb{E}[v_{\pi_\theta}(S_0) \mid S_0 \sim \iota] \\
&= \mathbb{E}\Big[\sum_{s\in\mathcal{S}} \sum_{k=0}^{\infty} \gamma^k \mathbb{P}\{s \mid S_0, k, \pi\} \sum_{a\in\mathcal{A}(s)} \nabla_\theta \pi(a \mid s,\theta) q_{\pi_\theta}(s,a) \mid S_0 \sim \iota\Big] \\
&= \sum_{s\in\mathcal{S}} \eta(s) \sum_{a\in\mathcal{A}(s)} \nabla_\theta \pi(a \mid s,\theta) q_{\pi_\theta}(s,a) \\
&= L \sum_{s\in\mathcal{S}} \frac{\eta(s)}{L} \sum_{a\in\mathcal{A}(s)} \nabla_\theta \pi(a \mid s,\theta) q_{\pi_\theta}(s,a) \\
&= L \sum_{s\in\mathcal{S}} \mu_{\pi_\theta}(s) \sum_{a\in\mathcal{A}(s)} \nabla_\theta \pi(a \mid s,\theta) q_{\pi_\theta}(s,a),
\end{aligned}$$

where we have used the definition of $\eta(s)$ in Lemma 8.2 and (8.1).

In the continuing case, similar calculations for the differential return

$$G_t := \sum_{k=1}^{\infty} (R_{t+k} - r(\pi_\theta))$$

can be done.

□

Interestingly enough, although the performance measure depends on the state distribution which depends on the policy parameter θ, the derivative of the state distribution does not appear in the expression found in the policy-gradient theorem. This is the usefulness of the theorem.

Remark 8.4 (experience replay). The expression for the gradient of the performance measure found in the theorem includes the state distribution μ and hence motivates experience replay. Experience replay is a method which keeps a cache of states and actions that have been visited and which are replayed during learning in order to ensure that the whole space is sampled in an equidistributed manner. Cf. Remark 6.8.

8.4 Monte-Carlo Policy-Gradient Method: REINFORCE

Having the gradient of the performance measure available from Theorem 8.3, we can use gradient based stochastic optimization. The most straightforward way is the iteration

$$\theta_{t+1} := \theta_t + \alpha \sum_{a \in \mathcal{A}(S_t)} \hat{q}_w(S_t, a) \nabla_\theta \pi_\theta(a \mid S_t, \theta),$$

where $\hat{q}_w$ is an approximation of q_{π_θ} and parameterized by a vector $w \in \mathbb{R}^d$. This iteration is called an all-actions method.

The more classical REINFORCE algorithm is derived as follows. We start from state S_t in time step t and use Theorem 8.3 to write

$$\begin{aligned}
\nabla_\theta J(\theta) &= L\mathbb{E}_{\pi_\theta}\Big[\gamma^t \sum_{a \in \mathcal{A}(S_t)} q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a \mid s, \theta)\Big] \\
&= L\mathbb{E}_{\pi_\theta}\Big[\gamma^t \sum_{a \in \mathcal{A}(S_t)} \pi_\theta(a \mid S_t, \theta) q_{\pi_\theta}(S_t, a) \frac{\nabla_\theta \pi_\theta(a \mid S_t, \theta)}{\pi_\theta(a \mid S_t, \theta)}\Big],
\end{aligned}$$

since the discounted state distribution μ_{π_θ} includes a factor of γ for each time step. Next, we replace the sum over all actions by the sample $A_t \sim \pi_\theta$. Then the gradient of the performance measure is approximately proportional to

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\Big[\gamma^t q_{\pi_\theta}(S_t, A_t) \frac{\nabla_\theta \pi_\theta(A_t \mid S_t, \theta)}{\pi_\theta(A_t \mid S_t, \theta)}\Big].$$

Having selected the action A_t, we use Definition 2.8 to find

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\gamma^t G_t \frac{\nabla_\theta \pi_\theta(A_t \mid S_t, \theta)}{\pi_\theta(A_t \mid S_t, \theta)}\right].$$

This yields the gradient used in the REINFORCE update

$$\begin{aligned}\theta_{t+1} &:= \theta_t + \alpha\gamma^t G_t \frac{\nabla_\theta \pi_\theta(A_t \mid S_t, \theta_t)}{\pi_\theta(A_t \mid S_t, \theta_t)} \\ &= \theta_t + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t \mid S_t, \theta_t).\end{aligned}$$

Since the return G_t until the end of an episode is used as the target, this is an MC method.

The algorithm for this update is shown in Algorithm 16.

Algorithm 16 REINFORCE for calculating $\pi_\theta \approx \pi_*$.

initialization:
choose a representation the policy π_θ
choose learning rate $\alpha \in \mathbb{R}^+$
initialize policy parameter $\theta \in \Theta \subset \mathbb{R}^{d'}$

loop ▷ for all episodes
  generate an episode $(S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T)$ following π_θ
  for $t \in (0, 1, \dots, T-1)$ **do** ▷ for all time steps
    $G := \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\theta := \theta + \alpha\gamma^t G \nabla_\theta \ln \pi_\theta(A_t \mid S_t, \theta)$
  end for
end loop

return θ

REINFORCE is a stochastic gradient ascent method. By the construction based on Theorem 8.3, the expected update over time is in the same direction as the performance measure J. Under the standard stochastic approximation conditions (2.2), the algorithm converges to a local optimum. On the other hand, REINFORCE is a MC algorithm and thus may be of high variance.

8.5 Monte-Carlo Policy-Gradient Method: REINFORCE with Baseline

The right side in Theorem 8.3 can be changed by subtracting an arbitrary so-called baseline b, a function or a random variable of the state, from the action-value function, i.e.,

$$\begin{aligned}\nabla_\theta J(\theta) &= L \sum_{s\in\mathcal{S}} \mu_{\pi_\theta}(s) \sum_{a\in\mathcal{A}(s)} q_{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a \mid s,\theta) \\ &= L \sum_{s\in\mathcal{S}} \mu_{\pi_\theta}(s) \sum_{a\in\mathcal{A}(s)} (q_{\pi_\theta}(s,a) - b(s))\nabla_\theta \pi_\theta(a \mid s,\theta).\end{aligned}$$

The last equation holds true because

$$\sum_{a\in\mathcal{A}(s)} b(s)\nabla_\theta \pi_\theta(a \mid s,\theta) = b(s)\nabla_\theta \underbrace{\sum_{a\in\mathcal{A}(s)} \pi_\theta(a \mid s,\theta)}_{=1} = 0.$$

With this change, the update becomes

$$\theta_{t+1} := \theta_t + \alpha\gamma^t (G_t - b(S_t))\nabla_\theta \ln \pi_\theta(A_t \mid S_t,\theta_t).$$

What is the purpose of adding a baseline? It leaves the expected value of the updates unchanged, but it is a method to reduce their variance. The natural choice is an approximation of the expected value of G_t, i.e., the state-value function. This approximation

$$\hat{v}_w(s) \approx v_{\pi_\theta}(s)$$

of the state-value function $v_{\pi_\theta}(s)$, where $w \in W \subset \mathbb{R}^d$ is a parameter vector, can be calculated by any suitable method, but since REINFORCE is an MC method, an MC method is used for calculating the approximation in Algorithm 17 as well.

The general rule of thumb for choosing the learning rate α_w is

$$\alpha_w := \frac{0.1}{\mathbb{E}[\|\nabla_w \hat{v}_w(S_t)\|_\mu^2]},$$

which is updated while the algorithm runs. Unfortunately, no such general rule is available for the learning rate α_θ, since the learning rate depends on the range of the rewards and on the parameterization of the policy.

REINFORCE with baselines is unbiased and its approximation of an optimal policy converges to a local minimum. As an MC method, it converges slowly with high variance and inconvenient to implement for continuing environments or learning tasks.

Algorithm 17 REINFORCE with baseline for calculating $\pi_\theta \approx \pi_*$.

initialization:
choose a representation for the policy π_θ
choose a representation for the state-value function $\hat{v}_w$
choose learning rate $\alpha_\theta \in \mathbb{R}^+$
choose learning rate $\alpha_w \in \mathbb{R}^+$
initialize policy parameter $\theta \in \Theta \subset \mathbb{R}^{d'}$
initialize state-value parameter $w \in W \subset \mathbb{R}^d$

loop ▷ for all episodes
  generate an episode $(S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T)$ following π_θ
  for $t \in (0, 1, \dots, T-1)$ **do** ▷ for all time steps
    $G := \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\delta := G - \hat{v}_w(S_t)$
    $w := w + \alpha_w \delta \nabla_w \hat{v}_w(S_t)$
    $\theta := \theta + \alpha_\theta \gamma^t \delta \nabla_\theta \ln \pi_\theta(A_t \mid S_t, \theta)$
  end for
end loop

return θ and w

8.6 Temporal-Difference Policy-Gradient Methods: Actor-Critic Methods

REINFORCE with baseline is an MC method, since the target value in the update is the return till the end of the episode. In other words, no bootstrapping is performed, i.e., no previous approximation of a value function is used to update the policy. Although the state-value function is used in the iteration, it is used to only for the state that is currently being updated; it serves for variance reduction, not for bootstrapping.

Bootstrapping (e.g., by going from MC to TD methods) introduces a bias. Still, this is often useful as it reduces the variance in the value function or policy and hence accelerates learning. Going from MC methods to TD methods is analogous to going from REINFORCE with baseline to actor-critic methods. Just as TD methods, actor-critic methods use the return calculated over a certain number of time steps; the simplest case being to use the return $G_{t:t+1}$ using only one time step.

Here we introduce the one-step actor-critic method that uses the one-step return $G_{t:t+1}$ as the target in the update and that still uses the learned state-value function $\hat{v}_w$ as the baseline $b := \hat{v}_w$. This yields the iteration

$$\begin{aligned}\theta_{t+1} &:= \theta_t + \alpha\gamma^t(G_t - b(S_t))\nabla_\theta \ln \pi_\theta(A_t \mid S_t, \theta_t) \\ &= \theta_t + \alpha\gamma^t(R_t + \gamma\hat{v}_w(S_{t+1}) - \hat{v}_w(S_t))\nabla_\theta \ln \pi_\theta(A_t \mid S_t, \theta_t)\end{aligned}$$

The update of the baseline is now performed by TD(0) in order to be consistent in the methods that are used to learn the baseline and the policy.

The name comes from the intuition that the policy is the actor (answering the question what to do) and the baseline, i.e., the state-value function is the critic (answering the question how well it is done).

This one-step actor-critic method is shown in Algorithm 18.

Of course, instead of the one-step return, the n-step return $G_{t:t+n}$ or the λ-return G_t^λ can be used.

8.7 Bibliographical and Historical Remarks

Problems

Algorithm 18 one-step actor critic for calculating $\pi_\theta \approx \pi_*$.

initialization:
choose a representation for the policy π_θ
choose a representation for the state-value function $\hat{v}_w$
choose learning rate $\alpha_\theta \in \mathbb{R}^+$
choose learning rate $\alpha_w \in \mathbb{R}^+$
initialize policy parameter $\theta \in \Theta \subset \mathbb{R}^{d'}$
initialize state-value parameter $w \in W \subset \mathbb{R}^d$

loop ▷ for all episodes
  initialize $s \sim \iota$
  $G := 1$
  while s is not terminal **do** ▷ for all time steps
    choose action a according to $\pi_\theta(\cdot|s)$
    take action a and receive the new state s' and the reward r
    $\delta := R + \gamma \hat{v}_w(s') - \hat{v}_w(s)$
    $w := w + \alpha_w \delta \nabla_w \hat{v}_w(s)$
    $\theta := \theta + \alpha_\theta G \delta \nabla_\theta \ln \pi_\theta(a \mid s, \theta)$
    $G := \gamma G$
    $s := s'$
  end while
end loop

return θ and w

Chapter 12

Large Language Models

Large Language Models are artificial neural networks that generate texts in natural languages following commands or prompts. In this chapter, the training and functioning of large language models, and ChatGPT in particular, is explained. Transformers and their attention mechanism provide the basis for text generation token by token. But generating grammatically correct and beautifully formulated text is not enough. The output of large language models should be aligned to the intentions of the user. This is achieved by various training steps, one of them being reinforcement learning from human feedback.

12.1 Introduction

The processing of natural language has been part of the goals of the field of artificial intelligence since its inception. The proposal for the Dartmouth Summer Research Project on Artificial Intelligence, dated August 31, 1955, mentions

- natural language,
- artificial neuronal networks,
- self-improvement, and
- creativity

among "some aspects of the artificial intelligence problem." (The Dartmouth Workshop took place one year later, in the summer of 1956.) These four aspects are certainly essential for large language models (LLM), showing how prescient the proposal was. The proposal says the following about the second of the seven aspects mentioned.

> "How Can a Computer be Programmed to Use a Language
>
> It may be speculated that a large part of human thought consists of manipulating words according to rules of reasoning and rules of conjecture. From this point of view, forming a generalization consists of admitting a new word and some rules whereby sentences containing it imply and are implied by others. This idea has never been precisely formulated nor have examples been worked out."

ChatGPT was launched on November 30, 2022. It revolutionized natural-language processing and has achieved at least some of the goals broadly outlined in the proposal for the Dartmouth Workshop. To which extent is ChatGPT intelligent and creative? The answer to this question is in the eye of the beholder, but there can be no doubt that ChatGPT shows intelligence and creativity.

Why did it take nearly seventy years until computers could be programmed to use natural language at a level comparable to humans or even at a superhuman level? There are two complications.

The first is that any program that can usefully deal with natural language requires a world model or common sense, thus knowing how the world works and how the subjects and objects in a text may interact. This information must be conveyed to the program somehow. Common sense is also required to be able to deal with ambiguities.

The second complication is that natural language is full of references, sometimes over large distances in a text. In order to resolve references correctly, context must be understood. A common example are personal pronouns that reference nouns.

LLM such as ChatGPT address the first complication by being training on huge, high-quality text corpora. This is an example of self-improvement: the learning algorithm extracts information (including grammar) from the text corpora and stores it in a new form, i.e., the neural network, for later use. No grammatical rules or facts are programmed into an LLM directly. Self-improvement and the use of huge text corpora necessitate a probabilistic approach to learning and to generating textual output in order to deal with contradictions that are most likely present in large text corpora. The probabilistic approach to generating output is linked to the so-called temperature parameter, which in turn is linked to creativity.

The second challenge is addressed by the attention mechanism of transformers.

12.2 Transformers

Transformers are a certain kind of deep neural network.[1] Their defining feature is their sole use of the so-called attention mechanism, which makes it possible to learn references within a text or a time series, in a rather simple architecture. Transformers were introduced in [37] for use in machine translation in 2017. Before this publication, the dominant sequence transduction models for use in machine translation were recurrent or convolutional neural networks, but already used attention mechanisms. The advantages of transformers reported in [37] are superior quality, more parallelizable models, and shorter training times.

The following discussion of transformers is based on [37] and pertains mostly to the features that set them apart from other neural-network architectures, i.e., scaled dot-product attention, multi-head attention, and positional encoding. A simplified version of transformers for machine translation are the basis of ChatGPT, discussed later. Much of the design of neural-network architectures is empirical; after the success of transformers in machine translation, their use has expanded to many other application areas, and many variations have been proposed and implemented.

A schematic diagram of the original transformer architecture for machine translation is shown in Figure 12.1. We start with an overview, and go into the details later. The left part is the encoder stack, and the right part is the decoder stack. The input text in the source language is converted into a vector of so-called tokens by the input embedding. Positional encoding is added. The light gray box on the left side in the figure contains the encoder and consists of six identical layers. In each layer, there are two sublayers, i.e., multi-head attention and feed forward. At the end of each sublayer, the two indicated vectors are added, and their sum is normalized.

The input to the encoder is a window, i.e., a vector of tokens with fixed length, that slides along the whole input text. The length of this vector is called the model dimension and has the value $d_{\text{model}} = 512$ in [37]. To facilitate all connections, all embeddings and sublayers have the same length, i.e., the model dimension.

In the first iteration, the vector containing the previous output in the target language shown at the bottom right contains only padding. The decoder shown on the right is similar to the encoder shown on the left. Its layers are also repeated six times, but it contains three sublayers, i.e., masked multi-head attention, multi-head attention, and feed forward. The masking in the first sub-

[1] A visualization of transformers can be found at https://poloclub.github.io/transformer-explainer/.

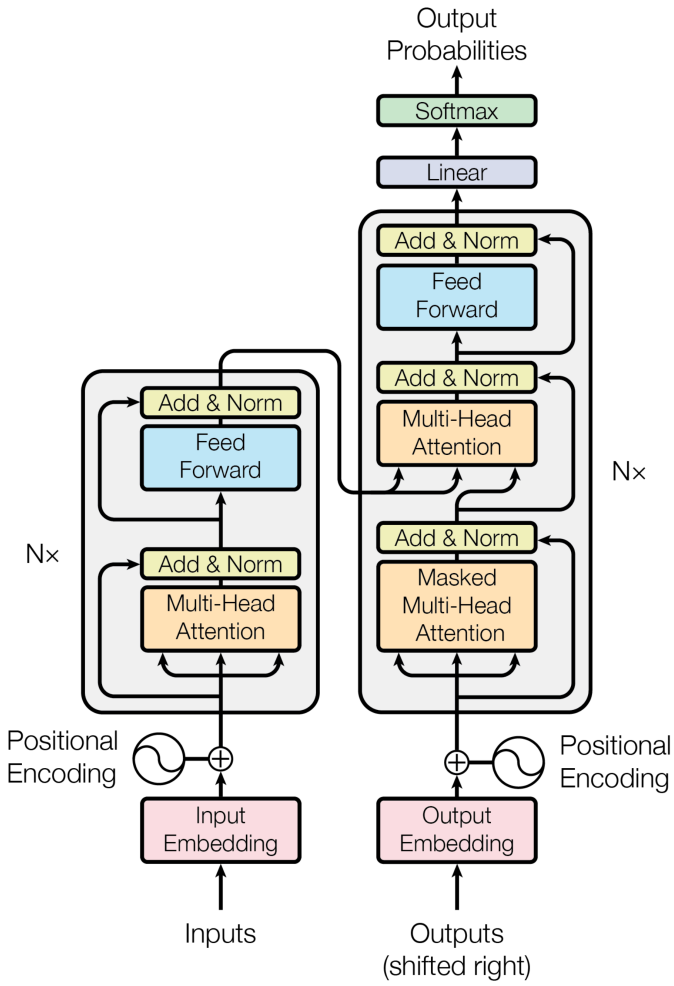


Figure 12.1: Transformer architecture for machine translation. Source: [37, Figure 1].

layers prevents positions from attending to subsequent positions, and hence the predictions only depend on known output tokens. The output of the encoder is passed to the second sublayers, i.e., the multi-head attention layers, of the six decoder layers, and conveys all meaning from the input to the output text.

Finally, a linear transformation is applied, and a softmax layer generates a single output token in a probabilistic manner depending on the temperature

parameter T by assigning the probability

$$\text{softmax}(x_i) := \frac{\mathrm{e}^{x_i/T}}{\sum_i \mathrm{e}^{x_i/T}}$$

to the i-th element of the real-valued input vector x to the softmax layer.

In this manner, a single token is generated in each iteration, which is an evaluation of the transformer, and appended to the output sequence. The transformer model is auto-regressive, as the previously generated tokens serve as additional input when generating the next token.

All textual inputs to a transformer are converted by embeddings to vectors of tokens, and the output of the transformer is another token. Why are tokens more useful than characters or words? Experience has shown that characters as the smallest textual units do not carry enough information. On the other hand, using words as the smallest units results in large vocabularies that contain tens of thousands of entries and is inflexible. Tokens are a useful compromise, and very good mental model of tokens are syllables.

The process of converting a text to tokens is called tokenization and is performed by a tokenizer. Each token has an ID, and the set of all tokens is called the vocabulary. A tokenizer can parse, i.e., convert a string to a list of characters; it can encode, i.e., convert a vector of tokens to their IDs; and it can decode, i.e., convert a vector of token IDs to a string.

Embeddings convert tokens to real valued vectors of dimension d_{model}. This makes it possible, for example, that tokens of similar meaning or use are converted to points that are close in the real vector space. to In [37], the embeddings are learned linear transformation, although other choices exist [38].

Next, we turn our attention to… attention. Figure 12.2 shows schematic diagrams for scaled dot-product attention and multi-head attention.

In the mechanism of scaled dot-production attention, queries and keys of dimension d_k as well as values of dimension d_v are used. In practice, the attention function works on a set of queries simultaneously, and hence the queries, keys, and values are stored in matrices Q, K, and V. Scaled dot-product attention is defined as

$$\text{attention}(Q, K, V) := \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V.$$

Elements of the inner product $QK^\top$ are large whenever queries and keys point into the same direction and vanish whenever they are orthogonal. Thus, whenever the queries and keys are aligned, the values V are the result of the attention, otherwise the attention is (close to) zero. Therefore this mechanism enables the

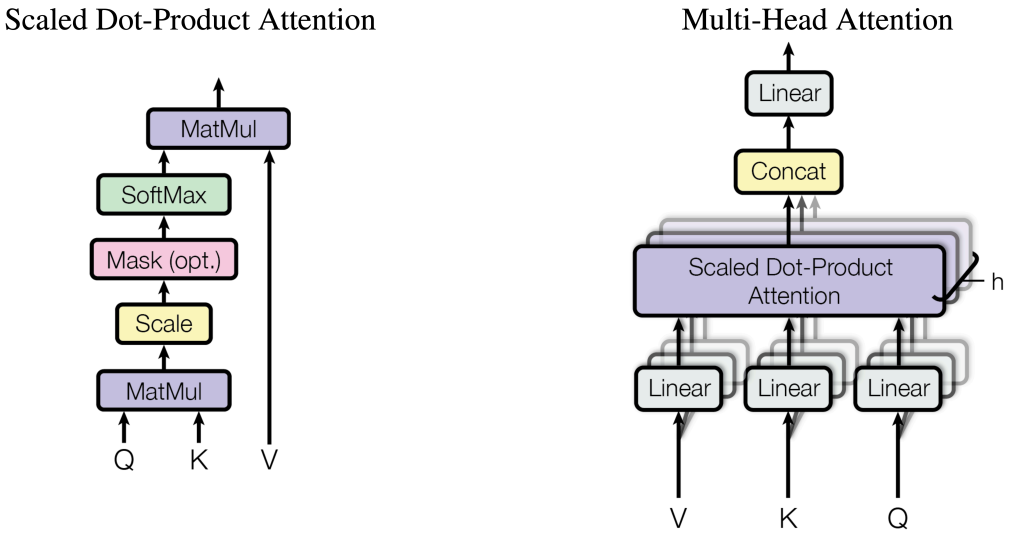


Figure 12.2: Scaled dot-product attention (left) and multi-head attention (right). Source: [37, Figure 2].

learning of grammar. For example, queries and keys make it possible to match nouns and pronouns.

All operations in this definition are standard functions and can be implemented using highly optimized code. The scaling factor $\sqrt{d_k}$ in the denominator is important, because the inner product in the argument to the softmax function may become large in magnitude for large values of d_k. The value of the scaling factor is motivated by the following fact. Suppose the elements of the two vectors $\mathbf{q} \in \mathbb{R}^{d_k}$ and $\mathbf{k} \in \mathbb{R}^{d_k}$ are independent random variables with expectation 0 and variance 1. Then their inner product $\mathbf{q} \cdot \mathbf{k}$ has mean 0 and variance d_k, which means that after scaling by $1/\sqrt{d_k}$ the variance of the inner product is 1.

In the mechanism of multi-head attention [37], the d_{model}-dimensional keys, values, and queries are projected h times using learned linear projections to d_k, d_k, and d_v dimensions, respectively. The attention function is applied to the projected versions in parallel, yielding d_v-dimensional vectors. These are concatenated and projected again, i.e.,

$$\begin{aligned} \text{head}_i &:= \text{attention}(QW_i^Q, KW_i^K, VW_i^V) \in \mathbb{R}^{d_v}, \\ \text{multiHead}(Q,K,V) &:= \text{concat}(\text{head}_1, \ldots, \text{head}_h)W^O \end{aligned}$$

with the parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, and $W_i^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$. In [37], $h = 8$ and $d_k = d_v = d_{\text{model}}/h = 64$ was used.

Since each of the multiple heads is smaller, the total computational cost is similar to that of single-head attention with full dimensionality.

The advantage of multi-head attention is that the model can pay attention to information from different representation subspaces at different positions. This is inhibited by averaging if a single attention head is used.

There are three applications of attention in [37]. Firstly, in the encoder-decoder attention layers, the queries come from the previous decoder layer, while the keys and the values come the output of the encoder. In this manner, all positions in the decoder can attend to all positions in the input sequence.

Secondly, the attention layers in the encoder are self-attention layers. In the encoder, all keys, values, and queries come from the output of the previous encoder layer. Hence, all positions in the encoder can attend to all positions in the previous layer in the encoder.

Thirdly, analogously to the attention layers in the encoder, the attention layers in the decoder are also self-attention layers, but with the slight difference that all positions in the decoder attend to all positions in the decoder up to and including that position. Otherwise, information could flow leftward and the auto-regressive property would be violated.

Finally, we discuss how positional information is encoded in transformers [37]. It is advantageous for the neural network to be able to use the order of the sequence and to know the distances between tokens. Such information about the relative and absolute positions of tokens is made available by positional encoding, which is added to the input embeddings at the bottoms of the encoder and decoder stacks. Both the embeddings and the positional embeddings have dimension d_{model} and are added.

Many approaches to positional embeddings exist. In [37], the sine and cosine functions

$$\begin{aligned} \mathrm{PE}(p, 2i) &:= \sin(p/10\,000^{2i/d_{\text{model}}}), \\ \mathrm{PE}(p, 2i+1) &:= \cos(p/10\,000^{2i/d_{\text{model}}}) \end{aligned}$$

of different frequencies, where p is the position and i the dimension, were used. The wavelengths are a geometric progression from 2π to $10\,000 \cdot 2\pi$.

Such a positional encoding has two advantages. Firstly, all values of PE are between -1 and $+1$, which is advantageous for training neural networks. Secondly, attention to relative positions is easily learned, since $\mathrm{PE}(p+k, i)$ is a linear function of $\mathrm{PE}(p, i)$ for any fixed k.

The authors compared this sinusoidal positional encoding with learned positional encodings, finding that both yielded nearly identical results, and eventually chose the sinusoidal positional encoding, because its behavior for sequence

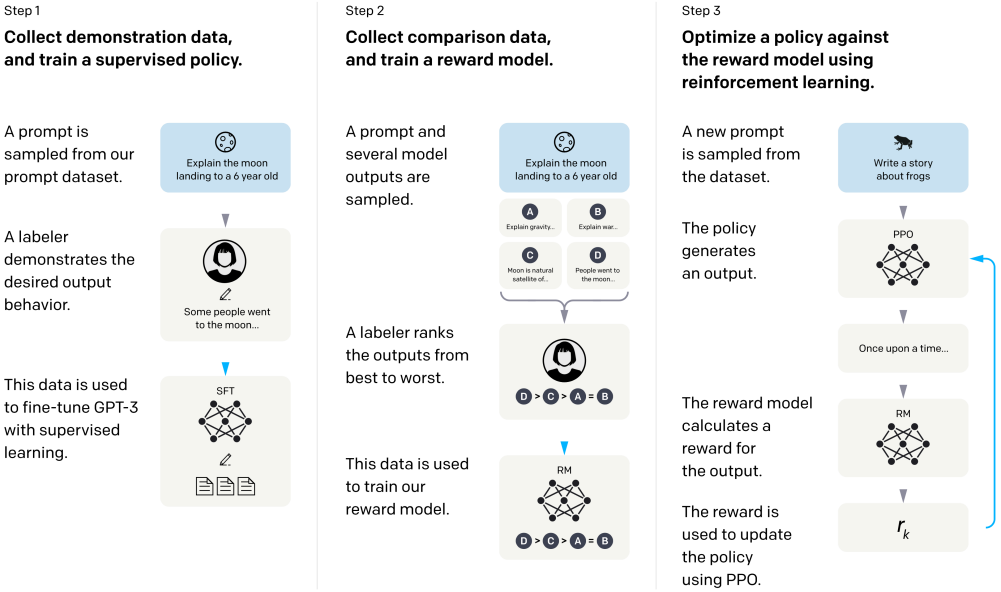


Figure 12.3: The three steps used in training InstructGPT [39, Figure 2].

lengths longer than the ones encountered during training is more predictable.

12.3 InstructGPT and ChatGPT

G: generative, P: pretrained, T: transformer.

The predecessor of ChatGPT is called InstructGPT [39].

A model is called aligned if it is

- helpful,
- honest, and
- harmless.

Figure 12.3 shows the three steps used in training InstructGPT.

Table 12.1 shows the criteria given to the labelers.

12.4 Rewards

This section is about finding rewards that can be used to train large language models.

Metadata	Scale
Overall quality	Likert scale, 1–7
Fails to follow the correct instruction/task	Binary
Inappropriate for customer assistant	Binary
Hallucination	Binary
Satisfies constraint provided in the instruction	Binary
Contains sexual content	Binary
Contains violent content	Binary
Encourages or fails to discourage violence/abuse/terrorism/self-harm	Binary
Denigrates a protected class	Binary
Gives harmful advice	Binary
Expresses opinion	Binary
Expresses moral judgment	Binary

Table 12.1: The criteria the human labelers used to judge model output [39, Table 3].

12.4.1 Direct Assignment of Rewards

Rewards may be assigned directly. First, multiple responses to a prompt are collected. Then human annotators rank these responses, and these rankings are converted to the numerical rewards that are needed in RL.

Unfortunately, rewards based on rankings come with issues such as ambiguity, the quality of the human feedback, and how ties or small differences are handled.

How can rankings be converted to rewards? Various approaches are possible. The simplest is to assign fixed rewards to the ranks in order; e.g., the first, second, and third place are assigned rewards of $+1$, 0, and -1 or rewards of $+1$, 0.5, and 0.

Another approach is to use normalized rewards, i.e., to assign rewards based on a linear scale from zero to one, for example. Suppose there are n responses in total and the ranks denoted by a are $a \in \{1, \dots, n\}$. Then the reward $(n - a)/(n - 1) \in [0, 1]$ is assigned to rank a.

A third approach is to use a nonlinear scaling such as logarithmic or exponential scaling. Using logarithmic scaling, the reward for rank a is $-\ln a$; using exponential scaling, the reward for rank a is $\mathrm{e}^{-\lambda a}$. Here $\lambda \in \mathbb{R}^+$ is a positive constant that is responsible for how much the differences between the ranks are scaled.

12.4.2 Reward Functions by Supervised Learning

Instead of directly assigning rewards from rankings, a reward function can be learned by supervised learning. The reward model, which usually is an artificial neural network, predicts the reward given the prompt and the response.

A starting point for defining a loss function for the reward model is the Bradley-Terry model, first presented in 1952. It assigns a probabilities to the outcome of pairwise comparisons between items. In the nomenclature of the Bradley-Terry model, the preference $i > j$ of two items i and j with scores $s_i \in \mathbb{R}^+$ and $s_j \in \mathbb{R}^+$ is assigned the preference value or probability

$$p_{i>j} := \frac{s_i}{s_i + s_j} \in \mathbb{R}^+,$$

where the preference $i > j$ has different meanings according to the applications. Note that the equation $p_{i>j} + p_{j>i} = 1$ supports the interpretation of the values as probabilities.

If the scores are scaled exponentially, the original Bradley-Terry model

$$p_{i>j} := \frac{\exp(s_i)}{\exp(s_i) + \exp(s_j)} \in \mathbb{R}^+$$

is obtained using more general scores $s_i \in \mathbb{R}$ and $s_j \in \mathbb{R}$. Note that the equation $p_{i>j} + p_{j>i} = 1$ still holds and that factors can be used in the exponents.

While the Bradley-Terry model works with pairwise preferences, the more general Plackett-Luce model works for preferences of items taken from arbitrarily long lists. If there are n items $\{i_1, \dots, i_n\}$ with scores $\{s_1, \dots, s_n\}$, the preference value of $i_1 > i_2 > \cdots > i_n$ is

$$p_{i_1>i_2>\cdots>i_n} := \prod_{k=1}^{n} \frac{s_k}{\sum_{j=k}^{n} s_j} = \frac{s_1}{s_1 + \cdots + s_n} \frac{s_2}{s_2 + \cdots + s_n} \cdots \frac{s_n}{s_n}.$$

Next, we denote the output of the reward model with parameters θ for a prompt x and a response y by $r_\theta(x, y)$. If the response y_1 is preferred to y_2 and if we use the exponentially scaled version of the Bradley-Terry model, we have

$$p_{y_1>y_2} = \frac{\exp(r_\theta(x, y_1))}{\exp(r_\theta(x, y_1)) + \exp(r_\theta(x, y_2))}$$

for the probability of this preference. Maximizing this probability leads to a loss function. The optimal parameter is the maximizer

$$\theta_* := \arg\max_\theta \frac{\exp(r_\theta(x, y_1))}{\exp(r_\theta(x, y_1)) + \exp(r_\theta(x, y_2))}$$

$$= \arg\max_\theta \sigma(r_\theta(x, y_1) - r_\theta(x, y_2))$$
$$= \arg\min_\theta(-\log(\sigma(r_\theta(x, y_1) - r_\theta(x, y_2)))).$$

Here we have used the sigmoid function $\sigma(x) := 1/(1 + \exp(-x))$. This consideration for two responses leads to the loss function

$$J(\theta) := -\mathbb{E}_{(x,y_1,y_2)\sim D}[\log(\sigma(r_\theta(x, y_1) - r_\theta(x, y_2)))]$$

for the reward model r_θ in [39, page 8], where D is the dataset of human preferences and y_1 is preferred to y_2 given prompt x.

12.5 Proximal Policy Optimization (PPO)

The third step in training InstructGPT and ChatGPT uses reinforcement learning based on the reward model found in the second step (see Figure 12.3). More precisely, proximal policy optimization (PPO), which is a policy-gradient method, is used.

Whenever local optima of smooth functions are sought, the gradient of the objective function is instrumental in devising efficient optimization algorithms [40, Chapter 12].

In reinforcement learning, gradient methods can be applied to the action-value function or to the policy directly or to both. Policy-gradient methods are based on the policy-gradient theorem (see Section 8.3).

PPO is a family of policy-gradient methods[41]. The algorithms alternate between sampling epochs of minibatches from the environment and optimizing a surrogate objective function using stochastic gradient ascent. According to [41], PPO methods share some of the benefits of trust-region policy optimization (TRPO), are simpler to implement, and empirically have better sample complexity.

Policy-gradient methods maximize the performance

$$J_{\mathrm{PG}}(\theta) = \hat{\mathbb{E}}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t],$$

where $\hat{A}_t(s, a)$ is the estimate of the advantage function

$$A_t(s, a) := Q_t(s, a) - V_t(s).$$

The approximation $\hat{\mathbb{E}}$ of the expectation is the sample mean based on a finite batch of samples. This objective function, i.e., the performance, leads to the estimate

$$\hat{g}(\theta) = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t],$$

of the gradient.

It is not well-justified to perform multiple optimization steps on the performance J_{PG} using the same trajectory and empirically it often leads to destructively large updates [41].

TRPO is a trust-region method [42]. It uses the surrogate objective function

$$\begin{aligned} &\text{maximize}_\theta \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] \\ &\text{subject to} \quad \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta, \end{aligned}$$

which is maximized subject to the constraint on the size of the update, i.e., the difference between the old and new policies. Here KL denotes the Kullback-Leibler divergence

$$\text{KL}(P, Q) := \int p(x) \log \left(\frac{p(x)}{q(x)} \right) \mathrm{d}x$$

of two continuous random variables P and Q, where p and q are their densities.

Alternatively, instead of a constraint, a penalty can be used, which results in the unconstrained optimization problem

$$\text{maximize}_\theta \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t - \beta \, \text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right], \tag{12.1}$$

where β is a non-negative coefficient. Unfortunately, reasonable choices of β vary over the course of learning and among different learning problems.

Next, we define the clipped surrogate objective [41]. We note the probability ratio between the old and the new policies by

$$r_t(\theta) := \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}.$$

TRPO maximizes the objective

$$J_{\text{CPI}}(\theta) := \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t[r_t(\theta) \hat{A}_t].$$

The index refers to conservative policy iteration[2].

[2] [S. Kakade and J. Langford. "Approximately optimal approximate reinforcement learning". In: ICML. Vol. 2. 2002, pp. 267–274]

In order to limit the size of policy updates, the objective function is now modified such that changes in $r_t(\theta)$ away from $r(\theta_{\text{old}}) = 1$ are penalized. In [41], the performance

$$J_{\text{CLIP}}(\theta) := \hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

is hence proposed as the main objective function. The clip function ensures that the value $r_t(\theta)$ remains in the interval $[1-\epsilon, 1+\epsilon]$. Taking the minimum of the clipped and unclipped values ensures that the performance is a lower or pessimistic bound on the unclipped value. In other words, changes in the probability ratio $r_t(\theta)$ are included when they make the objective worse and are ignored when they would improve the objective.

Next, we discuss how the hyperparameter β in the penalized, unconstrained optimization problem (12.1) can be adapted dynamically [41]. The main idea is to adapt the penalty coefficient β so that a prescribed target value d_{targ} of the KL divergence is achieved.

We denote the expectation in (12.1) by $J_{\text{KLPEN}}(\theta)$. It is maximized using several epochs of minibatches. Then the difference

$$d := \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]]$$

between the old and new policies is computed, and the parameter β may be updated according to its value: if $d < d_{\text{targ}}/1.5$, then the updated penalty parameter is $\beta \leftarrow \beta/2$; if $d > d_{\text{targ}} \cdot 1.5$, then it becomes $\beta \leftarrow \beta \cdot 2$. The new value of β is used for the next policy updated, and so on. The parameters 1.5 and 2 are chosen heuristically.

It was found in [41] that J_{KLPEN} performs worse than J_{CLIP}.

Finally, we can formulate the main PPO algorithm. It combines three terms in its performance

$$J_{\text{CLIP+VS+S}}(\theta) := \hat{\mathbb{E}}_t \left[J_{\text{CLIP}} - c_1 J_{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right]$$

to be maximized, where c_1 and c_2 are non-negative coefficients. The second term is the squared-error loss

$$J_{\text{VF}} := \left(V_\theta(s_t) - V_t^{\text{target}} \right)^2,$$

which turns PPO into an actor-critic algorithm. The third term S is the entropy. The entropy is added as a reward in order to ensure sufficient exploration.

The policy-gradient implementation is often based on segments of episodes[3], which is also well-suited for use with recurrent neural networks. The policy is

[3] [V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. "Asynchronous methods for deep reinforcement learning". In: arXiv preprint arXiv:1602.01783 (2016).]

run for T time steps, where T is much less than the episode length, and this truncated segment of an episode is used as the sample for the policy-gradient update. This approach means that the estimator of the advantage function only uses T time steps. A very general choice is

$$\hat{A}_t := \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1},$$
$$\delta_t := r_t + \gamma V(s_{t+1}) - V(s_t).$$

The proximal policy optimization (PPO) algorithm is shown in Algorithm 20.

Algorithm 20 Proximal policy optimization (PPO) for policy optimization [41].

loop iterations
  for all actors from 1 to N **do**
    run policy $\pi_{\theta_{\text{old}}}$ for T time steps
    compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end for
  optimize surrogate performance/objective $J_{\text{CLIP+VS+S}}$ w.r.t. θ
    using K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} := \theta$
end loop

12.6 Group Relative Policy Optimization (GRPO)

Group relative policy optimization (GRPO) [43, 44] aims to improve the reasoning ability of an LLM. It simplifies PPO in the sense that no value function is learned (and hence it is not an actor-critic method). Instead of an approximation of the value function, the average reward of multiple sampled outputs produced in response to the same prompt is used as the baseline. For each question q, a set or group $\{o_1, ..., o_G\}$ of $G \in \mathbb{N}$ outputs is sampled by the old policy $\pi_{\theta_{\text{old}}}$, and then the policy is optimized by maximizing the objective

$$J_{\text{GRPO}}(\theta) := \mathbb{E}_{q\sim P(Q),\, \{o_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(O|q)} \left[\frac{1}{G}\sum_{i=1}^{G} \frac{1}{|o_i|}\sum_{t=1}^{|o_i|} \left(\min\left(\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}\hat{A}_{i,t}, \text{clip}\left(\frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}, 1-\epsilon, 1+\epsilon \right)\hat{A}_{i,t} \right) - \beta\,\text{KL}(\pi_\theta, \pi_{\text{ref}}) \right) \right]$$

(see [43, Section 4.1.1]). Here $\epsilon \in \mathbb{R}^+$ and $\beta \in \mathbb{R}^+$ are hyperparameters and $\hat{A}_{i,t}$ is the advantage calculated based on relative rewards of the outputs inside each group only. Note that the KL divergence between the trained policy and the reference policy is added directly to the loss. In fact, the KL divergence is approximated by the unbiased estimator

$$\mathrm{KL}(\pi_\theta, \pi_{\mathrm{ref}}) \approx \frac{\pi_{\mathrm{ref}}(o_{i,t}|q, o_{i,<t})}{\pi_\theta(o_{i,t}|q, o_{i,<t})} - \log \frac{\pi_{\mathrm{ref}}(o_{i,t}|q, o_{i,<t})}{\pi_\theta(o_{i,t}|q, o_{i,<t})} - 1 > 0.$$

The advantages are computed relative to their group of outputs. For each question q, a group of outputs $\{o_i\}_{i=1}^G$ are sampled from the old policy model $\pi_{\theta_{\mathrm{old}}}$. Then a reward model scores the outputs, which yields the G rewards $\{r_i\}_{i=1}^G$. These rewards are normalized by subtracting the group mean and dividing by the group standard deviation, i.e., the advantages $\hat{A}_{i,t}$ of all tokens in the output are these normalized rewards.

The GRPO algorithm is shown in Algorithm 21.

Algorithm 21 Group relative policy optimization (GRPO) for policy optimization [43].

input: initial policy $\pi_{\theta_{\mathrm{init}}}$, reward models r_ϕ, task prompts D, hyperparameters ϵ, β, μ
policy $\pi_\theta := \pi_{\theta_{\mathrm{init}}}$
for iteration $= 1, \ldots, I$ **do**
  reference model $\pi_{\mathrm{ref}} := \pi_\theta$
  for step $= 1, \ldots, M$ **do**
    sample a batch D_b from D
    update the old policy model $\pi_{\theta_{\mathrm{old}}} := \pi_\theta$
    sample G outputs $\{o_i\}_{i=1}^G \sim \pi_{\theta_{\mathrm{old}}}(.|q)$ for each question $q \in D_b$
    compute rewards $\{r_i\}_{i=1}^G$ for each sampled output o_i by running r_ϕ
    compute $\hat{A}_{i,t}$ for the t-th token of o_i
      through group relative advantage estimation
    for GRPO iteration $= 1, \ldots, \mu$ **do**
      update the policy model π_θ by maximizing the objective J_{GRPO}
    end for
    update r_ϕ through continuous training using a replay mechanism
  end for
end for
output: π_θ

12.7 Bibliographical and Historical Remarks

Transformers were introduced in [37]. Proximal policy optimization (PPO) was introduced in [41]. InstructGPT was introduced in [39].

12.8 Problems

1. Number of letters. Ask one or preferably more LLM how many letters some words contain and compare the results.

2. Implement a character based tokenizer.

3. Implement an embedding layer.

Bibliography

[1] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. "CARLA: an open urban driving simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[2] Peter R. Wurman et al. "Outracing champion Gran Turismo drivers with deep reinforcement learning". In: *Nature* 602 (2022), pp. 223–228. DOI: 10.1038/s41586-021-04357-7.

[3] Helmut Horvath. "Deep reinforcement learning with applications to autonomous driving". MA thesis. Vienna, Austria: TU Wien, 2024. URL: http://Clemens.Heitzinger.name.

[4] Tobias Kietreiber. "Combining maximum entropy reinforcement learning with distributional Q-value approximation methods". MA thesis. Vienna, Austria: TU Wien, 2023. URL: http://Clemens.Heitzinger.name.

[5] Pierrick Lorang, Horvath Helmut, Tobias Kietreiber, Patrik Zips, Clemens Heitzinger, and Matthias Scheutz. "Adapting to the "open world": the utility of hybrid hierarchical reinforcement learning and symbolic planning". In: *Proc. 2024 IEEE International Conference on Robotics and Automation (ICRA 2024)*. 13–17 May 2024. DOI: 10.1109/ICRA57147.2024.10611594. URL: https://ieeexplore.ieee.org/document/10611594.

[6] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68. DOI: 10.1145/203330.203343.

[7] "TD-Gammon". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Springer US, 2010, pp. 955–956. DOI: 10.1007/978-0-387-30164-8_813.

[8] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (2017), pp. 354–359. DOI: 10.1038/nature24270.

[9] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362 (2018), pp. 1140–1144.

[10] Noam Brown and Thomas Sandholm. “Superhuman AI for multiplayer poker”. In: *Science* 365.6456 (2019), pp. 885–890. DOI: 10.1126/science.aay2400.

[11] Tobias Salzer. “Reinforcement learning for games with imperfect information – teaching an agent the game of Schnapsen”. MA thesis. Vienna, Austria: TU Wien, 2023. URL: http://Clemens.Heitzinger.name.

[12] Stephen W. Falken. “Computers and Theorem Proofs: Toward an Artificial Intelligence”. Cited in *War Games* (1983). PhD thesis. Cambridge, MA, USA: MIT, 1960.

[13] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518 (2015), pp. 529–533.

[14] Max Jaderberg et al. “Human-level performance in 3D multiplayer games with population-based reinforcement learning”. In: *Science* 364 (2019), pp. 859–865.

[15] Aniruddh Raghu, Matthieu Komorowski, Imran Ahmed, Leo Celi, Peter Szolovits, and Marzyeh Ghassemi. *Deep reinforcement learning for sepsis treatment*. 2017. arXiv: 1711.09602.

[16] Markus Böck, Julien Malle, Daniel Pasterk, Hrvoje Kukina, Ramin Hasani, and Clemens Heitzinger. “Superhuman performance on sepsis MIMIC-III data by distributional reinforcement learning”. In: *PLOS ONE* 17.11 (2022). Impact factor of *PLOS ONE:* 3.752., e0275358/1–18. DOI: 10.1371/journal.pone.0275358. URL: https://doi.org/10.1371/journal.pone.0275358.

[17] Razvan Bologheanu, Lorenz Kapral, Daniel Laxar, Mathias Maleczek, Christoph Dibiasi, Sebastian Zeiner, Asan Agibetov, Ari Ercole, Patrick Thoral, Paul Elbers, Clemens Heitzinger, and Oliver Kimberger. “Development of a reinforcement learning algorithm to optimize corticosteroid therapy in critically ill patients with sepsis”. In: *Journal of Clinical Medicine* 12.4 (2023). Impact factor of *Journal of Clinicial Medicine:* 4.964., pp. 1513/1–13. DOI: 10.3390/jcm12041513. URL: https://doi.org/10.3390/jcm12041513.

[18] Jongchan Baek, Changhyeon Lee, Young Sam Lee, Soo Jeon, and Soohee Han. “Reinforcement learning to achieve real-time control of triple inverted pendulum”. In: *Engineering Applications of Artificial Intelligence* 128 (2024), p. 107518. DOI: 10.1016/j.engappai.2023.107518.

[19] Carlotta Tubeuf, Jakob aus der Schmitten, René Hofmann, Clemens Heitzinger, and Felix Birkelbach. "Improving control of energy systems with reinforcement learning: application to a reversible pump turbine". In: *Proc. 18th ASME International Conference on Energy Sustainability (ES 2024)*. Anaheim, CA, USA, 15–17 July 2024, ES2024-122475/1–8. DOI: 10.1115/ES2024-122475. URL: https://doi.org/10.1115/ES2024-122475.

[20] Eugene Ie, Chih-wei Hsu, Martin Mladenov, Vihan Jain, Sanmit Narvekar, Jing Wang, Rui Wu, and Craig Boutilier. *RecSim: a configurable simulation platform for recommender systems*. 2019. arXiv: 1909.04847.

[21] M. Mehdi Afsar, Trafford Crump, and Behrouz Far. *Reinforcement learning based recommender systems: a survey*. 2021. arXiv: 2101.06286.

[22] Federico Tomasi, Joseph Cauteruccio, Surya Kanoria, Kamil Ciosek, Matteo Rinaldi, and Zhenwen Dai. "Automatic music playlist generation via simulation-based reinforcement learning". In: *Proc. 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2023)*. 2023, pp. 4948–4957. DOI: 10.1145/3580305.3599777.

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: an Introduction*. 2nd edition. The MIT Press, 2018.

[24] Dimitri P. Bertsekas. *Reinforcement learning and optimal control*. Athena Scientific, 2019.

[25] Dimitri P. Bertsekas. *Rollout, policy iteration, and distributed reinforcement learning*. Athena Scientific, 2020.

[26] Dimitri P. Bertsekas. *Abstract dynamic programming*. Athena Scientific, 2022.

[27] Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010.

[28] Marc G. Bellemare, Will Dabney, and Mark Rowland. *Distributional Reinforcement Learning*. MIT Press, 2023.

[29] Warren B. Powell. *Reinforcement Learning and Stochastic Optimization: a Unified Framework for Sequential Decisions*. John Wiley & Sons, Inc., 2022.

[30] Laura Graesser and Wah Loon Keng. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley, 2020.

[31] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. 2nd edition. Packt Publishing, 2020.

[32] Miguel Morales. *Grokking Deep Reinforcement Learning*. Manning Publications Co., 2020.

[33] Alexander Zai and Brandon Brown. *Deep Reinforcement Learning in Action*. Manning Publications Co., 2020.

[34] Richard Bellman. *Eye of the Hurricane – an Autobiography*. World Scientific Publishing, 1984.

[35] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[36] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in Neural Information Processing Systems 12 (NIPS 1999)*. Ed. by S.A. Solla, T.K. Leen, and K. Müller. MIT Press, 2000, pp. 1057–1063. URL: http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, and I. Polosukhin. "Attention is all you need". In: *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. Ed. by I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 6000–6010. arXiv: 1706.03762. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[38] O. Press and L. Wolf. *Using the output embedding to improve language models*. 2016. arXiv: 1608.05859.

[39] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155.

[40] Clemens Heitzinger. *Algorithms with Julia – Optimization, Machine Learning, and Differential Equations using the Julia Language*. ISBN 978-3-031-16559-7, ISBN 978-3-031-16560-3 (eBook), DOI: 10.1007/978-3-031-16560-3, https://doi.org/10.1007/978-3-031-16560-3. Springer Nature, 2022.

[41] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal policy optimization algorithms*. 2017. arXiv: 1707.06347.

[42] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. *Trust region policy optimization*. 2015. arXiv: 1502.05477.

[43] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang Y.K. Li, Y. Wu, and Daya Guo. *DeepSeekMath: pushing the limits of mathematical reasoning in open language models*. 2024. arXiv: 2402.03300.

[44] DeepSeek-AI. *DeepSeek-R1: incentivizing reasoning capability in LLMs via reinforcement learning*. 2025. arXiv: 2501.12948.

List of Algorithms

Index