WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

EFMD
EQUIS
ACCREDITED

Bachelor Thesis

Learning Deep Reinforcement Learning

Thomas Schmied

Date of Birth: 08.04.1997
Student ID: 1553816

Subject Area: Information Business

Studienkennzahl: 033 561

Supervisor: Dr. Vadim Savenkov

Date of Submission: 15.08.2019

Department of Information Systems and Operations, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR INFORMATIONS-VERARBEITUNG UND PROZESS-MANAGEMENT DEPARTMENT OF INFORMATION SYSTEMS AND OPERATIONS

Contents

1 Introduction 1
1.1 Research questions . . . 1
1.2 Motivation . . . 1
1.3 Methods and idea . . . 2
1.4 Why is that important? . . . 2
1.5 What is the contribution? . . . 2

2 Overview 3

3 Reinforcement Learning 4
3.1 Key concepts . . . 4
3.2 Markov decision processes . . . 5
3.3 Dynamic Programming . . . 10
3.3.1 Overview . . . 10
3.3.2 Policy evaluation . . . 10
3.3.3 Policy improvement . . . 11
3.3.4 Policy iteration . . . 12
3.3.5 Value iteration . . . 12
3.4 Monte carlo methods . . . 13
3.4.1 Overview . . . 13
3.4.2 Monte Carlo prediction for state values . . . 13
3.4.3 Monte Carlo prediction for action values . . . 14
3.4.4 Monte Carlo control . . . 14
3.5 Temporal Difference Learning . . . 14
3.5.1 TD Prediction . . . 14
3.5.2 On-policy vs. Off-policy . . . 15
3.5.3 Sarsa . . . 16
3.5.4 Q-Learning . . . 16

4 Deep Learning 17
4.1 Overview . . . 17
4.2 Deep Feedfoward Networks . . . 17
4.3 Loss function . . . 19
4.4 Backpropagation . . . 20
4.5 Optimization . . . 20
4.6 Convolutional Neural Networks . . . 21
4.7 Further architectures . . . 21

5 Deep Reinforcement Learning **22**
5.1 Overview . . . 22
5.2 Q-Learning . . . 23
5.2.1 DQN . . . 24
5.2.2 Double DQN . . . 26
5.2.3 Prioritized experience replay . . . 27
5.2.4 Dueling DQN . . . 28
5.2.5 Multi-step learning . . . 29
5.2.6 Distributional RL . . . 29
5.2.7 NoisyNet . . . 30
5.2.8 Putting it all together: Rainbow-DQN . . . 31
5.3 Policy Optimization . . . 32
5.3.1 Policy gradient . . . 32
5.3.2 Actor-Critic Methods . . . 34
5.3.3 A3C, A2C . . . 36
5.3.4 Trust Region Policy Optimization - TRPO . . . 37
5.3.5 Proximal Policy Optimization - PPO . . . 38

6 Conclusion **40**

A Algorithms **48**

List of Figures

1 The agent-environment interaction in a Markov decision process. 6
2 Feedforward Neural Network 17
3 ReLU 18
4 Reinforcement Learning algorithms 22

List of Algorithms

1 Policy evaluation 11
2 Policy improvement 12
3 First Visit Monte Carlo Prediction for state values 13
4 $TD(0)$ 15
5 DQN 26
6 Categorical Algorithm 30
7 Advantage actor-critic algorithm 36
8 PPO 39
9 Policy Iteration 48
10 Value iteration 49
11 First Visit Monte Carlo Prediction for action values 49
12 Monte Carlo Control with exploring starts 50
14 Q-Learning 51
13 Sarsa 51

Abstract

This thesis explores how Deep Reinforcement Learning works. Deep Reinforcement learning combines two approaches to Artificial Intelligence, Deep Learning and Reinforcement Learning. For this reason, the first part of this thesis discusses the fundamentals of Reinforcement Learning. The second part continues with the theoretical minimum of Deep Learning. The last part examines the landscape of modern Deep Reinforcement Learning and discusses the most important algorithms in depth. In summary, this thesis provides a solid understanding of the fundamentals of Deep Reinforcement Learning and also explains the most important innovations and breakthroughs of recent years.

1 Introduction

1.1 Research questions

How does Deep Reinforcement Learning work? How can we apply Deep Reinforcement Learning?

1.2 Motivation

Deep Reinforcement Learning is a branch of Artificial Intelligence and refers to the combination of Deep Learning and Reinforcement Learning.
Deep Learning is a subset of Machine Learning that uses Neural Networks to learn data representations.
Reinforcement Learning is a machine learning approach which aims to teach agents how to solve tasks by trial and error.
For many years scientist have been working on those three areas of *Artificial Intelligence*, but only in the recent years they have become widely applicable. This happened mainly because of the availability of two driving forces: the necessary computing power and enormous amounts of data. [Goodfellow et al., 2016, p.19-20] Nowadays, AI is increasingly becoming part of our everyday lives. Deep learning based intelligent devices find their ways into households. Smartphones and laptops are transitioning from pure working devices to intelligent assistants and support humans.
In 2015 Deep Reinforcement Learning algorithms were applied to Atari games and outperformed human players [Mnih et al., 2015]. Since then, there have been many advances in the field of Deep RL.
In 2016 the world champion in the game of Go, Lee Sedol, was defeated by AlphaGo, a Deep RL based agent developed by Google DeepMind. [Silver et al., 2016]
One year later, in 2017, AlphaZero, the successor of AlphaGo, beat the best models in Chess and Shogi and of course also its predecessor, AlphaGo, at the game of Go. AlphaZero achieved super-human performance through pure self-play within twenty-four hours. [Silver et al., 2017]
In 2018, OpenAI‘s bot defeated the world‘s reigning champions in Dota2, a real-time combat arena video game. [OpenAI, 2018] Furthermore, most recently in 2019 the same happened to the world‘s best players in the game of StarCraft, another strategic video game. [Vinyals et al., 2019]
Moving from simple and naive Atari games to complex strategy games like Dota2 or StarCraft is a huge progress within less than four years.
Those achievements show that Deep Reinforcement learning is a powerful multi-purpose technique. There is no doubt that the progress we have witnessed has only been the beginning of what is to come. Deep Reinforcement

Learning will presumably play a central role in developing even more powerful AI.

1.3 Methods and idea

In my bachelor thesis I will explore how Deep Reinforcement Learning works. First, I will give an overview of the key mathematical underlying concepts. This part will cover the basics of conventional Reinforcement learning (i.e. *Markov Decision Processes*, *Dynamic Programming*, *Monte Carlo Methods*, *Temporal Difference Learning*) and of Deep Learning (i.e. *Neural Networks*, *Gradient Descent*, *Backpropagation*).
Then I will explore the landscape of modern algorithms used in Deep Reinforcement Learning. Here I will cover the underlyings for algorithms like *DQN* (+ variants), *A3C* and *PPO*.

1.4 Why is that important?

As mentioned before, Artificial Intelligence has become pervasive in recent years and this trend is very likely to continue. The architectures used have become better and more specialized over time, as the field progressed. Most current deep learning based systems learn from examples. They learn to approximate a certain function very well, but can only do just that. However, those systems do not generalize well, which is a thing humans are certainly really good at. None of the previously discussed systems would be able to perform well in any other task, except the one they were trained and specialized on. Not a single speech recognition system could identify a single cat in an image and neither OpenAI‘s Dota2 bot nor DeepMinds‘s StarCraft playing AI would be able to make sense of a word in this thesis. Unfortunately, they are not yet capable of building abstractive knowledge, i.e. they cannot use knowledge gained from a previous task and apply it to another different task. Current AI systems are narrow in their capabilities. Of course, we want them to become broader. Deep Reinforcement Learning might be a central component that helps us get there.

1.5 What is the contribution?

It is clear, that Deep Reinforcement Learning has huge potential. Currently, Deep Reinforcement Learning has higher entry barriers than other research areas. A lot of prerequisites are necessary and unfortunately most of the essential knowledge is hidden in research papers. This Bachelor thesis, among other goals, aims to facilitate the entry. Furthermore, as Deep Reinforcement

Learning is a promising field, the more people enter it, the more can contribute and the faster progress will be made. This thesis is my entry point to the field, so the reader accompanies me on my journey of *learning Deep Reinforcement Learning.*

2 Overview

Deep Reinforcement Learning is a branch of *Machine Learning*. Machine learning enables computer systems to acquire knowledge by extracting patterns from data. [Goodfellow et al., 2016, p. 2] Machine Learning can be decomposed into three main categories: *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning*.
Supervised Learning techniques learn by example, i.e. they learn from labelled data. The term “supervised” originates from the fact that the labels are assigned by a supervisor (a human being). [Goodfellow et al., 2016, p. 105] Given (millions of) different images with corresponding labels the neural networks learns to classify them. For example, given an image of a cat with a corresponding label "cat", it shall learn to classify this image and related images as "cat". If it predicts "dog" it is told, that is wrong and given the right answer, "cat". This is where the supervision and learning takes place. In recent years there has been enormous progress with respect to those techniques. Today they are widely applied and work very well in narrow tasks, such as Image Classification, Object Detection, Speech Recognition or Language Translation. Unfortunately, they require large amounts of training signals, which implies human effort to annotate the data.
Unsupervised Learning is a paradigm, in which agents learn through observation. In contrast to supervised learning, no supervisor is involved. The algorithm has to learn to make sense of the data on its own. [Goodfellow et al., 2016, p. 105] Typically, the goal of unsupervised learning is to find a structure hidden in unlabelled data points. [Sutton and Barto, 1998, p.2]
“*Reinforcement Learning* is a computational approach to understanding and automating goal-directed learning and decision making.” [Sutton and Barto, 1998, p.13] Here learning takes place through direct interaction with the environment. Whereas unsupervised learning tries to find hidden structure in data, reinforcement learning tries to maximize the reward signal received. [Sutton and Barto, 1998, p.2] Reinforcement Learning also differs from Supervised Learning. As already mentioned, Supervised learning algorithms learn from labelled training examples given (and labelled) by a supervisor (human). The label clearly specifies the correct action to take (e.g. "cat" or "dog"), given the example (e.g. image). Then the system learns to generalize

from the training data to situations that were previously not seen. However, in Reinforcement learning it is hardly possible to accurately determine the correctness of an action. They only are good or bad with respect to the outcome. [Sutton and Barto, 1998, p.2]
In the following we will first explain the fundamentals of Reinforcement Learning. Then we will continue with the theoretical minimum of Deep Learning. Finally, we will discuss how these two approaches are connected by illustrating some of the most important methods and most recent innovations in Deep Reinforcement Learning.

3 Reinforcement Learning

3.1 Key concepts

Before we delve deeper into the more formal and mathematical part of this chapter, we first need to define the most important elements and notions of a reinforcement learning system. In Reinforcement Learning, the central components are the *agent* and the *environment*. The agent is the decision-maker or learner. The environment is the world, which the agent lives in and interacts with. [Sutton and Barto, 1998, p.48]
The agent is situated in certain *states* within the environment and performs *actions* to interact with it. Besides these main elements, there are four important sub-elements: a *policy*, a *reward*, a *value* and a optional *model*. [Sutton and Barto, 1998, p.6]
The *policy* defines how the agent behaves at a certain point in time. It maps states in the environment to actions, i.e. it determines which action to take at a particular state.
At each time step the agent receives a certain number, the *reward* signal, from the environment. His objective is to maximize the cumulative reward he obtains in the long run. Hence, the reward determines which events are good and which are bad events for the agent. Therefore, the reward directly influences the policy.[Sutton and Barto, 1998, p.6]
In contrast to the reward, which indicates what is beneficial for the agent in an immediate sense, the *value* determines what is good in the long run. The value of a state is the total cumulative reward the agent can expect to obtain, starting at a certain state.
Another distinction: rewards are provided directly by the environment - values have to be estimated based on what the agents experiences over its entire lifetime. [Sutton and Barto, 1998, p.6]
Finally, we have the concept of the *model* which is learned from the envi-

ronment. The model makes predictions about state transitions and rewards, i.e. how the environment will behave. Models are used to plan ahead into the future. They allow the agent to consider situations that have not yet occurred.
Whether an agent does or does not learn and use a model of the environment is one of the most important distinctions between different kinds of Reinforcement learning algorithms. Methods that learn and use a model are called *model-based*. Methods that do not learn and use a model are called *model-free*. [Sutton and Barto, 1998, p.7] More on that in chapter 5.
We have now defined the most crucial terms. However, we will define further terms as we progress and as they become necessary.

In this chapter, we will first explain *tabular solution methods*, i.e. methods for which the *state space* (all possible states) and the *action space* (all possible actions to perform) are small, so that the *value functions* can be represented as arrays and tables. For such methods it is often possible to find exact solutions.
Approximate solution methods will be the topic of the chapter *Deep Reinforcement Learning*.

3.2 Markov decision processes

The general problem of reinforcement learning is formalized by *Markov Decision Processes*. [Bellman, 1957] [Howard, 1960] The basic idea is to capture the most important aspects of the real problem which an actor who interacts with his environment in order to achieve a certain goal, faces over time. [Sutton and Barto, 1998, p.2] To achieve this, the agent must be able to sense the state of the environment he interacts with, at least to some extent. In addition, the agent must be able to take actions within the environment, so that he can influence the state of it. Finally, the agent needs to pursue a particular goal (or multiple goals) which is (are) related to a specific state of the environment. [Sutton and Barto, 1998, p.2]
These 3 aspects (sensation, action, goal) are formalized by Markov Decision Processes and are the subject of this chapter.
In the previous chapter we have already defined the necessary terms on a conceptual level. Now we will define them in a more formal way.

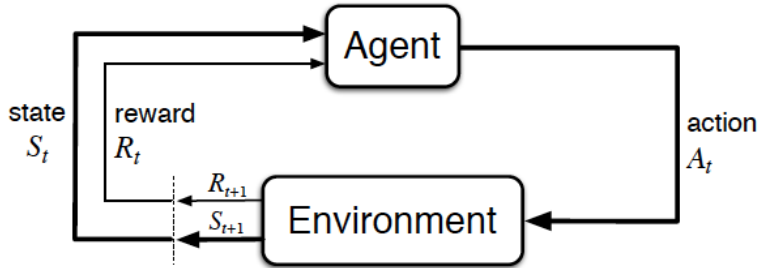

Figure 1: The agent-environment interaction in a Markov decision process, from [Sutton and Barto, 1998, p. 48]

As stated before, the *agent* is situated in an *environment* and interacts with it at every time step, t. At each step t the agent finds itself in a certain *state*, $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the *state space*. Based on S_t the agent selects *action*, $A_t \in \mathcal{A}$, where $\mathcal{A}$ is the *action space*.
As a result of the selected action, in $t+1$ the agent receives a *reward*, $R_t \in \mathcal{R} \subset \mathbb{R}$ and finds itself in a *new state*, S_{t+1}. Repeating this decision process several times gives the *trajectory*, τ, a sequence of state-action-reward triplets:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{3.1}$$

In the case of *finite* Markov Decision Processes $\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$ are finite. Hence R_t and S_t have well defined probability distributions, which only depend on previous state and action. Therefore, at time step t, given previous state and action, the random variables $s' \in \mathcal{S}$ and $r \in \mathcal{R}$ have a certain probability of occurring:

$$p(s', r \mid s, a) \doteq Pr\left\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a)\right\} \tag{3.2}$$

For $\forall s', s \in \mathcal{S}, r \in \mathcal{R}$ and $a \in \mathcal{A}$. Here s and a denote the observed state and taken action at time step t. s' and r denote the observed state and received reward in $t+1$, i.e. those are the results of taking action a in state s at time step t. p denotes the *dynamics* of our Markov Decision Process. In a Markov Decision Process, the probability of each value for S_t and R_t only depend on S_{t-1} and R_{t-1}. This is called the *markov property*. Similar as above, we can compute the *state-transition* probabilities:

$$p(s' \mid s, a) \doteq Pr\left\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\right\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \tag{3.3}$$

Furthermore, a state s is *markovian* (i.e. "posesses" the markov property) iff:

$$p(s' \mid s, a) = p(s' \mid \tau, a) \tag{3.4}$$

We can also compute the expected rewards for state-action pairs:

$$r(s, a) \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a) \tag{3.5}$$

Before it was stated, that the agent's goal is to maximize its cumulative expected reward. The cumulative reward is denoted as the *return*, G_t. Mathematically the return is defined as:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.6}$$

In fact (3.6) defines the expected *discounted return*, where γ denotes the *discount factor* with $0 \leq \gamma \leq 1$. The intuition behind discounting, is that immediate rewards are better than rewards received multiple time steps in the future. An analogy from finance would be that is better to receive 10 Million Euros today, than to receive 10 Million Euros in 1000 years. Since in 1000 years one could be dead.

The *policy*, π, maps states to probabilities of selecting each possible action. Hence, if the agent follows policy π the probability at time step t of action $A_t = a$ given, that $S_t = s$ is defined as $\pi(a \mid s)$. [Sutton and Barto, 1998, p.58]
The *value function* tells how good a certain state is and is defined in terms of the expected return. Of course, the expected return is influenced by which actions are taken and which actions are taken is defined by the policy, π. Therefore, value function,$V_\pi(s)$, of state s following policy π is defined as:

$$V_\pi(s) \doteq \mathop{\mathbb{E}}_{\pi}\left[G_t \mid S_t = s\right] = \mathop{\mathbb{E}}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \tag{3.7}$$

Equation (3.7) is called the *state-value function* for policy, π.
A important property of value functions in general is, that they satisfy recursive relationships. G_t could be re-expressed as the sum of the immediate

reward R_{t+1} and the return at the next time step G_{t+1}:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.8}$$

Now, we can exchange G_t in Equation (3.7) for our newly defined G_t in (3.8). Then:

$$\begin{aligned} V_\pi(s) &\doteq \mathop{\mathbb{E}}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[r + \gamma \mathop{\mathbb{E}}_\pi \left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V_\pi(s')\right] \\ &= \mathop{\mathbb{E}}_\pi \left[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s\right] \end{aligned} \tag{3.9}$$

The important thing is that in the last line we could exchange $\mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s'\right]$ for $V_\pi(s')$, which specifies the recursive relationship. This means that the state-value function is defined in terms of itself. Equation (3.9) is known as the *Bellman equation* for V_π. [Bellman, 1954]

Similar to Equation (3.7), we can define the *action-value function* for policy π, which is denoted by Q_π. Q_π represents the value of taking action a in state s following policy π. Intuitively Q_π determines how good it is to choose a certain action, when situated in a given state:

$$Q_\pi(s, a) \doteq \mathop{\mathbb{E}}_\pi \left[G_t \mid S_t = s, A_t = a\right] = \mathop{\mathbb{E}}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \tag{3.10}$$

Both, the state-value function and the action-value function are crucial concepts in Reinforcement Learning, and will appear throughout this thesis. Fortunately, both of them, V_π and Q_π can be estimated from experience. This will be the topic of the section on *Monte Carlo Methods* and the chapter on *Deep Reinforcement Learning*.

Furthermore, as in Equation (3.9) we can exchange G_t for (3.8) and we get the *Bellman equation* for *action-values*:

$$
\begin{aligned}
Q_\pi(s,a) &\doteq \mathop{\mathbb{E}}_{\pi}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \mathop{\mathbb{E}}_{\pi}\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\
&= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V_\pi(s')\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a\right]
\end{aligned} \tag{3.11}
$$

In order to achieve the best possible outcome, i.e. to receive the highest reward in the long run, one has to find the *optimal policy*. For a policy π to be better than or equal to another policy π', its expected return has to be greater than or equal for all states. Therefore, $\pi \geq \pi'$ iff $V_\pi(s) \geq V_{\pi'}(s)$.
The optimal policy has to better than any other policy and is denoted by π_*. Likewise, the *optimal state-value function* is denoted V_*:

$$
V_*(s) \doteq \max_\pi V_\pi(s) \tag{3.12}
$$

The *optimal action-value function*, Q_* is defined similarly:

$$
Q_*(s,a) \doteq \max_\pi Q_\pi(s,a) \tag{3.13}
$$

Using Equation (3.11) we can rewrite the Equation (3.13) in terms of V_* as:

$$
Q_*(s,a) \doteq \mathop{\mathbb{E}}_{\pi}\left[R_{t+1} + \gamma V_*(S_{t+1}) \mid S_t = s, A_t = a\right] \tag{3.14}
$$

Now, we can use the recursive property again to get the *Bellman optimality equation* for V_*:

$$
\begin{aligned}
V_*(a) &\doteq \max_\pi V_\pi(s) \\
&= \max_a Q_{\pi_*}(s,a) \\
&= \max_a \mathop{\mathbb{E}}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a\right] \\
&= \max_a \mathbb{E}\left[R_{t+1} + \gamma V_*(S_{t+1}) \mid S_t = s, A_t = a\right] \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V_*(s')\right]
\end{aligned} \tag{3.15}
$$

Likewise, we can express the *Bellman optimality equation* for Q_* as:

$$\begin{aligned} Q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma \max_{a'} Q_*(s', a')\right] \end{aligned} \tag{3.16}$$

The Bellman equations allow us to solve a Markov Decision Process in a optimal manner. *Dynamic Programming* is a technique to compute the solution and will be the topic of the next chapter. Unfortunately, however, the availability of that solution relies on three assumptions, which are hardly given in reality [Sutton and Barto, 1998, p. 66]:

- dynamics of the environment are known
- enough computational resources are available
- the Markov property

For this reason, it usually has to be approximated, which will be the topic of later chapters.

3.3 Dynamic Programming

3.3.1 Overview

"The term dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process." [Sutton and Barto, 1998, p. 73] However, they assume that a *perfect model* of the environment is available (i.e. that its true dynamics are known) and require enormous computational efforts. Nevertheless, they are theoretically very important and enable us to understand further topics. The basic idea of Dynamic Programming methods is to use value functions to search for good policies. [Sutton and Barto, 1998, p. 73].

3.3.2 Policy evaluation

The first important algorithm is called *Policy evaluation* and allows us to compute the state-value function V_π for an arbitrary policy π [Sutton and Barto, 1998, p. 74]. Policy evaluation applies the Bellman equation for state values as

defined in Equation (3.9). In the beginning, V is initialized arbitrarily, for example to zero. Then one iterates all states and computes the state value using (3.9). The resulting value can be compared to the previous value and the difference between them, Δ , is stored. The smaller Δ , the better, since if the values do not change (much), one assumes that the true value is reached. Once Δ gets below a certain threshold, θ, we stop iterating. Table 1 shows the algorithm for policy evaluation.

Algorithm 1: Policy evaluation

Input: policy π , threshold parameter θ
Output: $V \approx V_\pi$
Initialize $V(s)$ arbitrarily, $\forall s \in \mathcal{S}^+$, $V(s) = 0$
repeat
  $\Delta \leftarrow 0$
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end
until $\Delta < \theta$

Table 1: Policy evaluation

3.3.3 Policy improvement

The next important algorithm is called *policy improvement*, which as the name already hints, helps to find better policies. [Sutton and Barto, 1998, p. 76]
Its idea is to consider selecting at state s an action $a \neq \pi(s)$ (i.e. an action that does not follow the current policy) and thereafter again following π. This can be expressed as:

$$
\begin{aligned}
Q_\pi(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V_\pi(s')\right]
\end{aligned}
\tag{3.17}
$$

For it to be better this value has to be greater than $V_\pi(s)$. Now, in the more general case for *all* possible states $s \in (S)$, for π' to be better than π, $Q_\pi(s, \pi'(s)) \geq V_\pi(s)$.
This can be further generalized to include all possible actions. If one than

acts greedily, i.e. chooses the best possible action w.r.t. $Q_\pi(s,a)$ one obtains π', the new policy:

$$\begin{aligned}\pi'(s) &\doteq arg\max_a Q_\pi(s,a) \\ &= arg\max_a \mathbb{E}\left[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a\right] \\ &= arg\max_a \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma V_\pi(s')\right]\end{aligned} \tag{3.18}$$

The algorithm is shown in Table 2

Algorithm 2: Policy improvement

Input: value function V
Output: policy π'
for $s \in \mathcal{S}$ **do**
  for $a \in \mathcal{A}(s)$ **do**
    $Q(s,a) \leftarrow \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma V(s')\right]$
  end
  $\pi'(s) \leftarrow arg\max_a Q(s,a)$
end
return π'

Table 2: Policy improvement

3.3.4 Policy iteration

The third important algorithm in dynamic programming combines Policy evaluation and Policy improvement and is called *Policy Iteration*. First policy evaluation is performed followed by policy improvement. This process is repeated until the policy is stable which leads to a sequence of improving policies and value functions.

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{I} v_* \tag{3.19}$$

The policy iteration algorithm is shown in Table 9 in the Appendix.

3.3.5 Value iteration

Furthermore, we could use the Bellman Optimality equation for V_* (3.15) as an update in policy evaluation. The resulting algorithm, that contains this modification, is called *value iteration*. It is shown in Table 10 in the Appendix.

3.4 Monte carlo methods

3.4.1 Overview

In the previous chapter it was mentioned that the value function can be estimated using Monte Carlo Methods. Monte Carlo Methods do not require a complete model of the environment. They learn purely from actual or simulated experience. The idea of Monte Carlo Methods is simple: sample from the experience, then average the returns obtained for each state-action pair. [Sutton and Barto, 1998, p. 91]

3.4.2 Monte Carlo prediction for state values

First, we will explore how to estimate the state-value function V_π for a given policy π. There are two methods to accomplish that.
First visit Monte Carlo Prediction, as shown in Algorithm 3, estimates $V_\pi(s)$ as the average of the obtained returns following *first visits* to a state s.

Algorithm 3: First Visit Monte Carlo Prediction for state values

```
Input: policy π , num_of_episodes
Output: V ≈ v_π
Initialize V(s) arbitrarily, ∀s ∈ S
Initialize Returns(s) ← empty list, for all s ∈ S
for i ← 0 to num_of_episodes do
    Generate episode following π:
     S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T−1}, A_{T−1}, R_T
    G ← 0
    for t ← 0 to T − 1 do
        G ← γG + R_{t+1}
        if S_t in S_0, S_1, ..., S_{t−1} then
            Append G to Returns(S_t)
            V(S_t) ← average(Returns(S_t))
    end
end
return V
```

Table 3: First Visit Monte Carlo Prediction for state values

Similarly, *Every visit Monte Carlo Prediction* estimates $V_\pi(s)$ as the average of the obtained returns following *all visits* to a state s. [Sutton and Barto, 1998, p.92]

3.4.3 Monte Carlo prediction for action values

Monte Carlo Methods are typically used when no model of the environment is available. If this is the case, estimating action values is especially useful. Therefore, Monte Carlo Methods aim to estimate Q_*. As mentioned earlier, we can do this by *first visit*. This means, that the value of a state action pair is estimated as the average of all the returns following the first time in each episode that the state was visited and the action selected. The *every visit* method works similarly, but for every visit, not just the first visit. The algorithm for *first visit Monte Carlo prediction for action values* is shown in Algorithm 11 in the Appendix.
As the episodes are generated by following policy π and only the state-action pairs that were already visited have a value so far, those will be favoured again in the next iteration. Therefore, other state-action pairs may never be visited at all. For this reason, we can specify that the episodes must start in a certain state action pair and that each pair has a non-zero probability of being selected as the start pair. This is called *exploring starts* and is used in the next algorithm.

3.4.4 Monte Carlo control

Of course, as in Section 3.3.4 the goal is to find the optimal policy. The algorithm to find the optimal policy is called *Monte carlo control* and is shown in Table 12.

3.5 Temporal Difference Learning

In this chapter we will explore *Temporal Difference Learning*, which combines the ideas of Section 3.3 Dynamic Programming and Section 3.4 Monte Carlo Methods. Temporal Difference Learning is *model free* as Monte Carlo Methods. Furthermore, it updates the estimates based on other estimates (*bootstrapping*), like Dynamic Programming methods do. [Sutton and Barto, 1998, p.119]

3.5.1 TD Prediction

As in previous chapters, the goal is to estimate the value function V_π for a given policy π. [Sutton and Barto, 1998, p. 119] Both, Monte Carlo Methods and Temporal Difference Learning methods learn from experience. While Monte Carlo Methods have to wait until the end of the episode to update $V(S_t)$, Temporal Difference Learning methods only wait until the next time

step. This is called *online learning*. The update of $V(S_t)$ can be expressed as:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{3.20}$$

Where α is a parameter for the step-size (i.e. it indicates how strong the update will be). This is called *one-step Temporal difference* or $TD(0)$. It can be extended to *n-steps* $TD(\lambda)$, where λ represents the number of steps. An important thing to mention is that the expression $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ denotes the *TD-Error*, δ. It measures the difference between the previously estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$ and is important throughout reinforcement learning. [Sutton and Barto, 1998, p. 121]
The algorithm itself is simple. As always, we take an action following π and observe the new state obtained reward. Then we perform the update for $V(S)$ as defined in Equation 3.20. Those steps are repeated until the observed state is terminal. $TD(0)$ is shown in Table 4:

Algorithm 4: $TD(0)$

```
Input: policy π, step-size α ∈ (0, 1], num_of_episodes
Output: V ≈ V_π
Initialize V(s) arbitrarily, ∀s ∈ S+
for i ← 0 to num_of_episodes do
    Initialize S
    foreach step in episode do
        A ← action given by π for S
        Take action A, observe R and S'
        V(S) ← V(S) + α [R + γV(S') − V(S)]
        S ← S'
    end
end
return V
```

Table 4: $TD(0)$

3.5.2 On-policy vs. Off-policy

In the next two chapters we will have a look at two different algorithms, *Sarsa* and *Q-Learning*. While Sarsa is *on-policy*, Q-Learning is an *off-policy* algorithm. Therefore, it is time to define those two terms.
In general, learning control methods try to learn the optimal policy. But in order to learn the optimal policy, they have to behave in a non-optimal way, such that they can explore the whole action space. [Sutton and Barto, 1998,

p.103]
On-policy methods try to learn action values for the optimal policy, but for a near-optimal policy that still explores. [Sutton and Barto, 1998, p. 103]
All algorithms we have discussed so far were on-policy.
Off-policy methods use two separate policies, a *behaviour policy* and a *target policy*. The behaviour policy is used to generate behaviour and to explore. The target policy is the policy that is learned and ultimately becomes the optimal policy. Since the learning process is separated from data generation, this approach is off-policy. [Sutton and Barto, 1998, p. 104]

3.5.3 Sarsa

Now that we know the conceptual difference between on-policy and off-policy, we can move on to another on-policy algorithm called *Sarsa*. [Sutton and Barto, 1998, p. 129]
Previously, in Section 3.5.1 we defined the update for state-values. But Sarsa uses the update for state-action pairs, which is defined as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{3.21}$$

The update rule, as defined in Equation 3.21, uses the current state-action pair, the reward received and the next state-action pair, i.e. $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Hence, the name *Sarsa*. The rest of the algorithm is similar to the previous one, as Table 13 shows.

3.5.4 Q-Learning

Moving on to the off-policy control version, *Q-Learning*. [Watkins and Dayan, 1992] [Sutton and Barto, 1998, p. 131] In fact, the algorithm for *Q-Learning* in Table 14 differs from *Sarsa* only in two important points. First, the update is different:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{3.22}$$

The second subtle difference is, that we select A at S using Q within the second for-loop and not within the first loop as before.
In Section 5.2 we will discuss the ideas of Q-Learning in more detail.

4 Deep Learning

4.1 Overview

Deep Learning is as Reinforcement Learning a part of the broader family of Machine Learning. Even though the origins of Deep Learning date back to the 1940s [Mcculloch and Pitts, 1943], it has only recently become widely applicable, as mentioned in the beginning of this thesis. Today, Deep Learning is at the heart of most of the major developments in Artificial Intelligence, such as Computer Vision [Real et al., 2018], [Li et al., 2019], Speech Recognition [Park et al., 2019] and Natural Language Understanding [Radford et al., 2019]. In this chapter the essentials of Deep Learning will be explained, s.t. we can establish the link between Reinforcement Learning and Deep Learning in the next chapter.

4.2 Deep Feedfoward Networks

"*Deep feedforward networks*, also often called *feedforward neural networks*, or *multilayer perceptrons* (MLPs), are the quintessential deep learning models." [Goodfellow et al., 2016, p. 167]
In general, Deep feedforward networks (and Neural Networks in general) aim to approximate a certain function f^*. In fact, they are also called *Universal function approximators* due to their ability to approximate any arbitrary function. [Hornik et al., 1989]

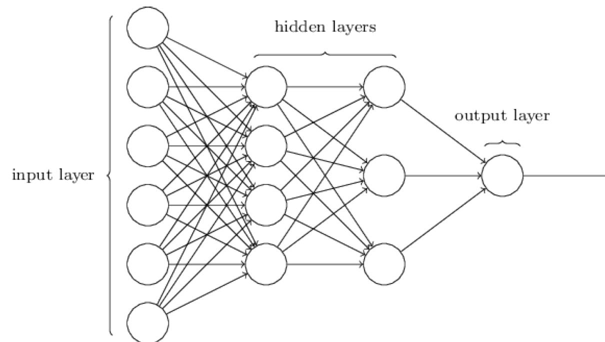

Figure 2: Feedforward Neural Network, from [Nielsen, 2018, p.11]

Typically, feedforward neural networks are arranged in layers, which is why they are called *networks*. Mathematically this can be represented as a function composition, e.g. for three functions $f^{(3)}, f^{(2)}, f^{(1)}$ as $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$

(or like this $f_3(f_2(f_1(x)))$). [Goodfellow et al., 2016, p. 167] In general, this composition can be represented as $f(x) = f^{(n)}(f^{(n-1)}(...(f^{(1)}(x))))$. Composing many functions makes it possible to represent very complicated functions. The innermost function is called *input layer* (here $f^{(1)}$). It receives the input to the network. The outermost function *output layer* (here $f^{(3)}$) returns the output of the network. The functions in between are called *hidden layers* (here only $f^{(2)}$).
The information flows from the input layer forward through the hidden layers to the output layer. Therefore, *feedforward*. Furthermore, the multilayer structure defines the *depth* of the network. Combined with the fact that Neural Networks aim to *learn* a certain function f^*, we arrive at the term Deep Learning. [Goodfellow et al., 2016, p.167]

In general, a simple layer is defined as:

$$y = g(\sum_i w_i x_i + b) \tag{4.1}$$

Or in matrix notation:

$$y = g(W^T x + b) \tag{4.2}$$

Where $y \in \mathbb{R}^{q \times m}$ is the *output* of the layer. $W \in \mathbb{R}^{q \times p}$ is the weight matrix and $b \in \mathbb{R}^{q \times m}$ a *bias* term, which are both learned during training. $x \in \mathbb{R}^{p \times m}$ is the *input* to the layer, and g is a non-linear differentiable *activation function*. Equation 4.2 is also referred to as a *fully connected layer*. Essentially, it multiplies the input times the weights, adds a bias and then applies the activation function ("input times weight, add a bias, activate." [Raval, 2017])
Many different activation functions are available. For hidden layers the simplest choice seems to work best in practice: *Rectified Linear Unit*, short *ReLU*. [Glorot et al., 2011]
ReLU is defined as:

$$g(z) = \max\{0, z\} \tag{4.3}$$

i.e. simply activating if the input is greater than 0.

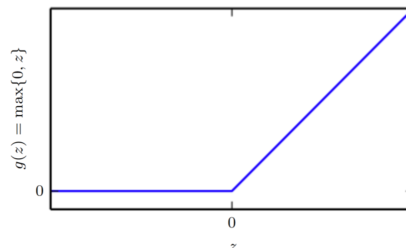

Figure 3: ReLU, from [Goodfellow et al., 2016, p. 174]

Other activation functions, that are heavily used include the *Sigmoid function*, $\sigma(z) = \frac{1}{1+e^{-z}}$, or the *Hyperbolic tangent function*, $g(z) = tanh(z)$. [Goodfellow et al., 2016, p. 194]
The choice of activation function is also crucial in the output layer. Depending on what problem has to be solved, different activation functions may be suitable. If the aim is to predict the value of a binary variable y (i.e. $0, 1$ or $true, false$) the sigmoid function is a prominent choice. In this case our output is a single number $\hat{y} \in [0, 1]$. $\hat{y}$ could represent the probability that y is of class 1 given input x:

$$\hat{y} = P(y = 1 \mid x) \tag{4.4}$$

If we would like to output a probability distribution over n possible classes instead, we can use the *softmax function*. In this case $\hat{y}$ is a n-dimensional vector, representing the probabilities of y being a certain class.

$$\hat{y} = P(y = i \mid x) \tag{4.5}$$

The *softmax function* is defined as:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{4.6}$$

The class with the highest probability is then the prediction of the output layer. In practice, however, we are using maximum likelihood to train neural networks. Therefore, we have to take the logarithm of the softmax function and obtain:

$$\log softmax(z)_i = \log z_i - \log \sum_j e^{z_j} \tag{4.7}$$

4.3 Loss function

In order to make Neural Networks learn, one first has to define a method to measure how good or bad the outputs produced by the network were. The *loss function* is this measure. On the basis of the *loss*, one can then calculate the *gradients* using *backpropagation* (the next chapter), which tell how the networks parameters have to be updated, so that the loss is minimized and the predictions improved.

Nowadays, most neural networks are trained using *maximum likelihood*. Therefore, the cost function can be described by the *negative log-likelihood*, or equivalently by the *cross entropy* between the training data and the outputs

of the neural network. [Goodfellow et al., 2016, p. 177] So, this loss function is given by:

$$J(\theta) = -\mathop{\mathbb{E}}_{x,y\sim\hat{p}_{data}}\left[\log p_{model}(y \mid x;\theta)\right] \tag{4.8}$$

In this case, p_{model} is the model distribution (i.e. the distribution represented by the neural network) and $\hat{p}_{data}$ the true distribution.
Another cost function that is heavily used is the *mean squared error*, which is defined as:

$$J(\theta) = \mathop{\mathbb{E}}_{x,y\sim\hat{p}_{data}}\left[(y - f(x;\theta))^2\right] \tag{4.9}$$

Here, $f(x;\theta)$ is the predictor, which produces the output $\hat{y}$ for a given input x. In both cases θ represents the parameters within the network.

4.4 Backpropagation

The *Backpropagation algorithm* allows us to compute the gradients of the parameters in the network based on the loss value. [Rumelhart et al., 1986] Later, these gradients are used to update the parameters, which is when the network "learns". In fact, Backpropagation actually only refers to computing the gradients, while other algorithms, such as stochastic gradient descent, are used to perform learning. [Goodfellow et al., 2016, p. 203]
Backpropagation relies on the *Chain Rule* to compute $\nabla_\theta J(\theta)$, the gradient of the cost function, $J(\theta)$, with respect to θ, the parameters in the network. In practice, computing the gradients and applying them is taken care of by Deep Learning Frameworks such as *TensorFlow* or *PyTorch*. [Abadi et al., 2015], [Paszke et al., 2017]

4.5 Optimization

As stated previously, back-propagation refers only to the method for computing the gradient. Once computed, they can be applied to θ, the parameters within the network. In general, this update rule is defined as:

$$\theta = \theta + \alpha\nabla_\theta J(\theta) \tag{4.10}$$

α is again the parameter which defines the *step-size*, also called *learning rate*. In practice, *Stochastic Gradient Descent* and its variants are the most used algorithms for optimization. [Goodfellow et al., 2016, p. 294]. Especially the so called *Adam* optimizer is widely in use. [Kingma and Ba, 2014]

4.6 Convolutional Neural Networks

A *Convolutional Neural Network* (CNN) is a type of architecture that was originally developed to perform well on images. [LeCun et al., 1989]
Today, CNNs are the standard building blocks for Computer Vision applications. [Real et al., 2018], [Li et al., 2019] They also turned out to deliver particularly good results on Natural Language tasks, such as Speech Recognition. [Abdel-Hamid et al., 2014] ,[Zeghidour et al., 2018]
In Deep Reinforcement Learning, convolutional neural networks act as the visual cortex of the agent. They allow the agent to perceive the environment which it interacts with. For example, they enable the agent to recognize Go boards, StarCraft maps or the robotic arm that it has to control. [Silver et al., 2017] [Vinyals et al., 2019] For this reason, these architectures will play an essential role in the next chapter.

4.7 Further architectures

Finally, we would like to mention that there are of course other architectures which have proven to perform very well in their domains.
Reccurent Neural Networks (RNN) are used to process sequential data. RNNs [Rumelhart et al., 1986] and its variants, i.e. *Long Short-Term Memory Networks* (LSTM) [Hochreiter and Schmidhuber, 1997] and *Gated Recurrent Networks* (GRU) [Cho et al., 2014] are typically used for *Natural Language Processing* (NLP) tasks.
Another type of architecture which was introduced recently is called *Transformer*. [Vaswani et al., 2017] *Transformers* use attention mechanisms and have proven to outperform recurrent networks on specific domains.
[Devlin et al., 2018], [Radford et al., 2019].

5 Deep Reinforcement Learning

5.1 Overview

Deep Reinforcement Learning methods use neural networks to approximate the state-value function V, the action-value function Q, the policy π and/or a model of the environment. In Chapter 3 *Reinforcement Learning* these were represented by a table, but as state space $\mathcal{S}$ or action space $\mathcal{A}$ become larger, this becomes intractable. Therefore, we aim to approximate them by a parametric function which can be accomplished by various techniques, such as Decision trees or different kinds of multivariate regression. [Sutton and Barto, 1998, p. 198] However, in recent years Neural networks have proven to outperform other approaches. As mentioned in chapter *Deep Learning*, Neural networks are capable of approximating any function, due to them being "Universal function approximators". [Hornik et al., 1989] In fact, we are trying to approximate the optimal functions or policy, i.e:

$$V(s;\theta) \approx V^*(s) \tag{5.1}$$

for the state-value function - likewise for the action value function:

$$Q(s,a;\theta) \approx Q^*(s,a) \tag{5.2}$$

And the optimal policy:

$$\pi(a \mid s,\theta) \approx \pi^*(a \mid s) \tag{5.3}$$

In all cases the variable θ denotes the parameters within the network which are learned. Approximating using neural networks establishes the connection between Reinforcement Learning and Deep Learning and gives rise to the term *Deep Reinforcement Learning*.

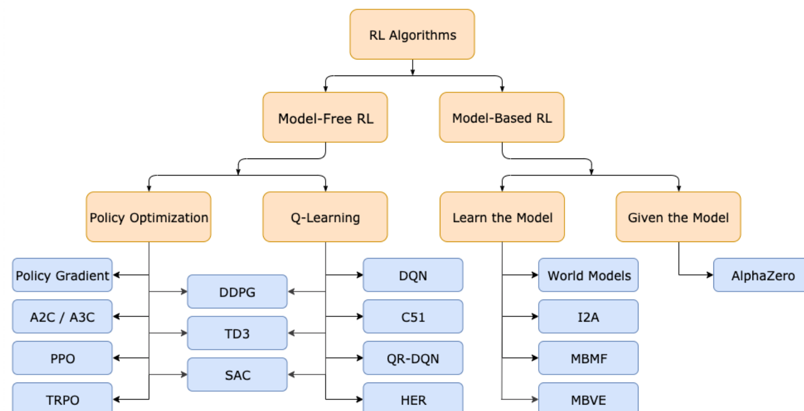

Figure 4: Reinforcement Learning algorithms, from [Achiam, 2018]

Figure 4 illustrates the main different types of Reinforcement Learning algorithms, without being exhaustive. The first major distinction is, whether a model of the environment is learned or not. Methods that learn and use a model of the environment are called *model-based*, whereas methods that do not learn a model are referred to as *model-free*, defined in Section 3.1.
More precisely, model-free methods learn the state-value function, the action-value function or the policy from experience without an estimation of a world model. [Dayan and Niv, 2008]
In contrast, in model-based methods a model of the environment is either given or the experience is used "to construct and internal model of the transitions and immediate outcomes in the environment." [Dayan and Niv, 2008]
Model-free methods are less sample efficient than model based methods, since the availability of a model allows the agent to plan ahead and to search for the appropriate action to take. [Dayan and Niv, 2008]. In most situations, however, a ground truth model is not available or hard to learn. For this reason, model-free methods currently are more popular and more widely applicable.
Both approaches can be further decomposed. *Q-Learning* and *Policy Optimization* are the two main model-free methods. Q-Learning on the one hand aims to approximate the optimal value function $Q^*(s, a)$ by $Q(s, a; \theta)$. On the other hand, Policy Optimization methods directly aim to approximate by a parametrized policy $\pi(a \mid s, \theta)$. [Sutton and Barto, 1998, p.321]
In model-based RL we can further distinguish between methods that actually learn the model and methods for which a model is already available and given, i.e. does not have to be learned.
In the following chapters, we will focus on model-free methods due to page restrictions.

5.2 Q-Learning

The tabular case of Q-Learning has already been introduced in Section 3.5.4. In this section we will first discuss a Q-learning method enhanced by Deep Learning, Deep Q-Networks (DQN).
DQN was originally proposed by researchers at DeepMind in 2013. It learned to play Atari games [Bellemare et al., 2015] through pure self-play and surpassed human performance in many of these games.[Mnih et al., 2013, Mnih et al., 2015]
Since then, several enhancements to the original DQN algorithm have been proposed, which significantly improved the performance. The most significant improvements include *Double Q-learning* [Hasselt et al., 2016], *Prioritized replay* [Schaul et al., 2016], *Dueling networks* [Wang et al., 2016c] , *Mutli-step learning* [Sutton, 1988, Sutton and Barto, 1998, Mnih et al., 2016],

Distributional RL [Bellemare et al., 2017] and *Noisy DQN* [Fortunato et al., 2017]. It turned out that integrating these individual components into a single agent, which was called *Rainbow-DQN*, yields particularly good results. [Hessel et al., 2017] These extensions will also be discussed later in this chapter.

5.2.1 DQN

The goal of a Deep Q-Network is to learn control policies directly from high dimensional sensory input (e.g. images) which are used to select actions that ultimately maximize the cumulative future rewards G_t (3.6). [Mnih et al., 2013] The optimal action-value function $Q^*(s,a)$, was defined in Equation 3.16 in the chapter on Reinforcement Learning:

$$Q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \tag{5.4}$$

As already stated, we aim to approximate it: $Q^*(s,a) \approx Q(s,a;\theta)$. The approximate value function is represented by the Q-network, a convolutional neural network (4.6) with θ, the parameters in the network. This can be achieved by training the Q-network to minimize the following mean squared error (4.9):

$$L_i(\theta_i) = \mathop{\mathbb{E}}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s,a;\theta_i)\right)^2\right] \tag{5.5}$$

Equation 5.5 contains the two most important ideas of DQN: experience replay [Lin, 1992] and a separate target-network. [Mnih et al., 2013, Mnih et al., 2015] Sequences of observations (e.g. consecutive frames) are highly correlated i.e. they are not i.i.d. This causes unstable learning. Maintaining an *experience replay*, D, which stores the last N experiences diminishes this issue by randomizing experiences and therefore breaking the correlation between them. At every time step t the agent's current experience $e_t = (s_t, a_t, r_t, s_{t+1})$ is saved to D. Then during the learning process experiences are sampled uniformly from the experience replay, $(s_t, a_t, r_t, s_{t+1}) \sim D$.
Furthermore, θ_i^- denotes the parameters, which are actually used to compute the target at time step t. In fact, the target is given by:

$$Y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) \tag{5.6}$$

Therefore, this is also called *target-network*. On the other hand, θ_i, the parameters of the *online network*, $Q(s,a;\theta_i)$, are kept up to date. This illustrates the off-policy nature (3.5.2) of DQN. Every C time steps θ_i^- is

updated by θ_i, but kept fixed during other iterations. C is known as the target network update frequency.
$L_i(\theta_i)$ is then optimized by stochastic gradient descent. Differentiating with respect to θ gives the gradient:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right] \tag{5.7}$$

Another important fact is that the agent follows a ϵ-greedy policy. This means that at every step the agent acts either according to the policy or randomly with probabilities $1 - \epsilon$ and ϵ respectively.

As mentioned earlier, DQN was originally tried on Atari games. Originally those games have a resolution of 210×160 pixels with a 128 colour palette. Since this can be computationally expensive the frames have to be preprocessed. The preprocessing step is performed by a function ϕ. Among other steps ϕ extracts the Y-channel and rescales it to 84×84. Furthermore, it stacks the $m = 4$ most recent frames, which are then input to the network. Stacking frames is crucial, since otherwise the game dynamics might not be represented correctly. [Mnih et al., 2015] For example, in the game of Breakout, one frame is not even enough to decide whether the ball will move upwards or downwards.

Finally, we will briefly discuss the architecture of the neural network used in [Mnih et al., 2015]. Generally, it consists of three convolutional layers (4.6) followed and two fully connected layers (4.2). All layers except the output layer use a ReLU as activation function.(4.3) As already mentioned, the input to the network is a $84 \times 84 \times 4$ stack, produced by ϕ. The stack is received by the first convolutional layer, which has 32 8×8 filter maps with $stride = 4$. The second convolutional layer has 64 4×4 filters and a stride of 2. The third convolutional layer has 64 3×3 filters and $stride = 1$. The first fully connected layer consists of 512 units. Finally, the output layer has n output units, with n corresponding to the number of possible actions (depending on the game).
Considering all these tricks, the DQN-algorithm is given by:

Algorithm 5: DQN

Input: capacity of experience replay N, number of episodes M, ϵ-greedy probability ϵ, target network update frequency C

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for *episode* $\leftarrow 0$ **to** M **do**
  Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t \leftarrow 1$ **to** T **do**
    With probability ϵ select random action a_t
    Otherwise select $a_t = arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to θ
    Every C steps $\hat{Q} = Q$
  end
end

Table 5: DQN, from [Mnih et al., 2015]

In the following chapters, Algorithm 5 will be extended and refined.

5.2.2 Double DQN

Based on earlier work on Double Q-learning [Hasselt, 2010], Double-DQN was proposed.[Hasselt et al., 2016]
In Equation 5.5 the target is given by:

$$Y^{DQN} = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) \tag{5.8}$$

Since a' is chosen by the target network, it can be rewritten as:

$$Y^{DQN} = r + \gamma Q(s', arg\max_{a'} Q(s', a'; \theta_i^-); \theta_i^-) \tag{5.9}$$

From 5.9 we see that the max operator uses the same values to select and evaluate an action which leads overoptimistic value estimates. The idea of Double DQN is simple: split up selection and evaluation of the actions. Therefore, it was proposed to evaluate the greedy policy according to the online network and to use the target network to estimate its value. [Hasselt et al., 2016]
So, we only have to exchange θ_i^- for θ_i in Equation 5.9:

$$Y^{DoubleDQN} = r + \gamma Q(s', arg\max_{a'} Q(s', a'; \theta_i); \theta_i^-) \tag{5.10}$$

This simple change results in severe performance improvements. It reduces overestimation and stabilizes the learning process. [Hasselt et al., 2016]

5.2.3 Prioritized experience replay

In the original DQN-paper [Mnih et al., 2015] the experiences were uniformly sampled from the experience replay. Intuitively, however, it would make sense to sample important experiences more frequently than irrelevant ones, i.e. to prioritize them. For this reason, *prioritized experience replay* was proposed.[Schaul et al., 2016]
Obviously a mechanism is needed which determines the importance of experiences. This measure is based on the TD-error, δ, as used in Equation 5.5. Using the Double-DQN extension gives:

$$\delta_i = r + \gamma Q(s', arg\max_{a'} Q(s', a'; \theta_i); \theta_i^-) - Q(s, a; \theta_i) \tag{5.11}$$

The magnitude of δ_i determines how much the actual value differs from the next estimate, i.e. it indicates how surprising the transition is. This measure is denoted by p_i and called the priority of transition i. It is defined by $p_i = |\delta_i| + \epsilon$, where ϵ is a small constant which assures that no experience has a probability of 0 being selected again, i.e. $p_i \geq 0$.
However, greedy prioritization based on p_i would lead to selecting the same experiences over and over again. Therefore, a stochastic prioritization method was proposed that interpolates between greedy prioritization and uniform sampling [Schaul et al., 2016]:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{5.12}$$

The hyper parameter α determines how much prioritization is used. Where $\alpha = 1$ and $\alpha = 0$ correspond to only selecting greedily and to only selecting in a uniform way respectively. The problem with priority sampling is that the

experiences used to compute the stochastic updates to estimate the expected value, come from a different distribution than the actual expected value. Therefore, importance-sampling weights are used additionally:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta \tag{5.13}$$

Where N denotes the size of the experience replay, and hyperparameter β determines how strongly the weight affects learning. β is annealed from its initial value ($[0, 1]$) to 1. The gradient descent step in DQN is then performed on $w_i\delta_i$ (instead of only on δ_i) with respect to θ.
To conclude this section, it was found that *prioritized replay* increases learning speed by a factor of two and significantly improved performance compared with the original DQN and Double DQN. [Schaul et al., 2016]

5.2.4 Dueling DQN

In [Wang et al., 2016c] a new architecture was proposed, Dueling DQN. In section 3.2, I already defined the state-value function (3.9), the action-value function (3.11) and their optimality versions (3.15, 3.16). Another important measure is called the *advantage function*, given by the difference of the action-value function and state-value function:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s) \tag{5.14}$$

Subtracting the value of a state from the value of choosing a certain action in this state, gives the relative measure of the importance of each action, the *advantage*. Now, the action-value function can be expressed as the sum of state-value function and advantage function:

$$Q_\pi(s, a) = A_\pi(s, a) + V_\pi(s) \tag{5.15}$$

The above directly relates to the idea of Dueling DQN, which is to use one estimator for the state-value function and another separate one for the state-dependent advantage function. This can be achieved by using two streams of fully connected layers (each stream consisting of two layers), instead of one (as used in [Mnih et al., 2015]). One stream represents the state-value function, $V(s; \theta, \beta)$, the other one represents the advantage function, $A(s, a; \theta, \alpha)$. With α and β denoting the parameters of the fully connected layers in the two separate streams and θ denoting the shared parameters of the convolutional layers. In the end, both streams are combined to produce the output Q-function:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s, a'; \theta, \alpha)\right) \tag{5.16}$$

The big advantage of Dueling DQN is, that the proposed changes are implemented as part of the network. For this reason, during the training process the parameters are automatically computed by backpropagation and optimized by gradient descent. [Wang et al., 2016c] In practice, this is taken care of by TensorFlow or PyTorch.

5.2.5 Multi-step learning

When looking at equation 5.5 and 5.6 we see that only the immediate reward r contributes to the target value. For this reason, Equation 5.6 is called the one-step target. Actually, however, we can use future rewards from multiple steps. This is the idea of multi-step learning, [Sutton, 1988], [Sutton and Barto, 1998] and was used to extend DQN. [Mnih et al., 2016] [Hessel et al., 2017] The n-step target is given by:

$$Y_{t:t+n} = \sum_{k=0}^{n-1} \gamma_t^k R_{t+k+1} + \gamma_t^n \max_{a'} Q(S_{t+n}, a'; \theta_t^-) \tag{5.17}$$

Consequently, $\Big(Y_{t:t+n} - Q(s,a;\theta)\Big)^2$ has to be minimized.

5.2.6 Distributional RL

So far the approach was, to maximize the return via the optimal action-value function 5.4. The Bellman equation for action-values can alternatively be written as:

$$Q(s,a) = \mathbb{E}(R(s,a)) + \gamma\,\mathbb{E}(Q(s',a')) \tag{5.18}$$

[Bellemare et al., 2017] argues for modelling the full distribution of the return, to get a much richer picture of the situation:

$$Z(s,a) \overset{D}{=} R(s,a) + \gamma Z(S',A')) \tag{5.19}$$

Equation 5.19 is the distributional version of the Bellman equation for action-values. The variable Z represents the distribution of the future rewards and replaces Q. It is called the value distribution and is characterized by the random variables R, S', A' and $Z(S',A')$.
Furthermore, the authors proposed to use a discrete distribution to model the value distribution. This distribution is based on the parametrized support z, with $z_i = V_{min} + i\Delta z$ and $\Delta z = \frac{V_{max} - V_{min}}{N_{atoms} - 1}$. Where $N_{atoms} \in \mathbb{N}$, $V_{min}, V_{max} \in \mathbb{R}$ and $i \in 1, ..., N_{atoms}$.
The probabilities of each atom, i.e. each z_i are given by:

$$Z_\theta(s,a) = z_i \;\text{ with probability }\; p_i(s,a) = \frac{e^{\theta_i(s,a)}}{\sum_j e^{\theta_j(s,a)}} \tag{5.20}$$

The authors then propose, to project the sample Bellman update $\hat{\mathcal{T}}Z_\theta$ onto the support of Z_θ. The bellman update for each atom z_j is computed by $\hat{\mathcal{T}}z_j = r + \gamma z_j$ and its probability $p_j(s', \pi(s'))$ is distributed to the immediate neighbours of $\hat{\mathcal{T}}z_j$. We denote this projected update by $\Phi\hat{\mathcal{T}}Z_\theta(s,a)$. The loss function $L_{s,a}(\theta)$ is the cross entropy term of the Kullback-Leibler divergence:

$$D_{KL}(\Phi\hat{\mathcal{T}}Z_{\tilde{\theta}}(s,a) \;||\; Z_\theta(s,a)) \tag{5.21}$$

The whole algorithm is then given by:

Algorithm 6: Categorical Algorithm

Input: A transition s_t, a_t, r_t, s_{t+1}, $\gamma_t \in [0,1]$
$Q(s_{t+1,a}) = \sum_i z_i p_i(s_{t+1}, a)$
$a* \leftarrow arg\max_a Q(s_{t+1}, a)$
$m_i = 0,\ i \in [0, ..., N-1]$
for $j \in [0, ..., N-1]$ **do**
  $\hat{\mathcal{T}}z_j \leftarrow [r_t + \gamma_t z_j]_{V_{min}}^{V_{max}}$
  $b_j \leftarrow \hat{\mathcal{T}} - V_{min})/\Delta z$
  $l \leftarrow \lfloor b_j \rfloor$
  $u \leftarrow \lceil b_j \rceil$
  $m_l \leftarrow m_l + p_j(s_{t+1}, a^*)(u - b_j)$
  $m_u \leftarrow m_u + p_j(s_{t+1}, a^*)(b_j - l)$
end
return $-\sum_i m_i log(p_i(s_t, a_t))$

Table 6: Categorical Algorithm, from [Bellemare et al., 2017]

In the paper they used $V_{min} = -10, V_{max} = 10$ and $N_{atoms} = 51$. The choice for the number of atoms gave the algorithm its name, *C51*.
Of all the extensions to DQN mentioned, Distributional DQN gave the strongest improvements in performance.

5.2.7 NoisyNet

The idea of NoisyNet is simple: Replace the fully connected layers in the network with noisy linear layers. This leads to the network learning to explore (and to stop exploring) on its own directly by gradient descent, without the need for an ϵ-greedy policy. [Fortunato et al., 2017]
In Section 4 *Deep Learning*, Equation 4.2 essentially defined a fully connected layer. It is modified as follows. The bias term b is exchanged for:

$$b = \mu^b + \sigma^b \odot \epsilon^b \tag{5.22}$$

Likewise, for the weight matrix:

$$W = \mu^w + \sigma^w \odot \epsilon^w \tag{5.23}$$

$\mu^w \in \mathbb{R}^{q \times p}, \sigma^w \in \mathbb{R}^{q \times p}, \mu^b \in \mathbb{R}^q$ and $\sigma^b \in \mathbb{R}^q$ are learnable parameters. $\epsilon^w \in \mathbb{R}^{q \times p}$ and $\epsilon^b \in \mathbb{R}^q$ are random noise variables. To be more specific, the noise variables ϵ^w and ϵ^b introduce factorized Gaussian noise. $\odot$ represents the element-wise multiplication operation (Hadamard product). As a result, we get for the linear layer:

$$y = (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b \tag{5.24}$$

The additional parameters are part of the network and can therefore be learned during the training process. (Similar to Dueling DQN modifications)

5.2.8 Putting it all together: Rainbow-DQN

All previously mentioned extensions to the original DQN algorithm have individually improved its performance. Additionally, they address different issues, but build on the same framework. Therefore, it is natural to ask, whether they could be combined. This was already the case in some of the papers discussed. For example, [Wang et al., 2016c] combined Double DQN with the Dueling architecture and prioritized experience replay and observed that they were very compatible. In fact, it turned out that all of them integrate well and are "indeed largely complementary."[Hessel et al., 2017] The resulting approach was called *Rainbow* and exceeded not only in performance, but also in data efficiency.

5.3 Policy Optimization

Instead of relying on a value function, policy optimization methods aim directly at learning a parametrized policy $\pi(a \mid s, \theta)$ (alternatively written as $\pi_\theta(a \mid s)$). [Sutton and Barto, 1998, p.321] [Sutton et al., 1999] This policy can be both deterministic or stochastic. In the first part of this chapter, we begin by defining the most important quantities. All of the more advanced concepts which follow in the next chapters will be based on the ideas we derive and will require only minor modifications.

5.3.1 Policy gradient

The parameters θ which again represent the weights in the network, are learned by performing gradient ascent on a performance measure, $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{5.25}$$

$\nabla_\theta J(\theta)$ is called the *policy gradient* and α is the learning rate. [Sutton and Barto, 1998, p.321] The goal is to maximize the expected total (not yet discounted) reward following the parametrized policy. So, we define:

$$J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} [r(\tau)] = \int \pi_\theta(\tau) r(\tau) d\tau \tag{5.26}$$

Where $r(\tau)$ is the sum of the rewards received for trajectory τ, i.e. $r(\tau) = \sum_{t=1} r(s_t, a_t)$. Here $r(s_t, a_t)$ denotes the reward received for performing action a_t at state s_t. Now taking the gradient of $J(\theta)$, we get the policy gradient:

$$\begin{aligned} \nabla_\theta J(\theta) &= \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau \\ &= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau \\ &= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} [r(\tau) \nabla_\theta \log \pi_\theta(\tau)] \end{aligned} \tag{5.27}$$

In the second line in Equation 5.27 we just applied the log-derivative trick $\nabla_\theta \pi_\theta = \pi_\theta \frac{\nabla_\theta \pi_\theta}{\pi_\theta} = \pi_\theta \nabla_\theta \log \pi_\theta(\tau)$. Now, since

$$\pi_\theta(\tau) = \pi_\theta(s_t, a_t, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t) \tag{5.28}$$

When we take the logarithm we have:

$$\log \pi_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t \mid s_t) + \log p(s_{t+1} \mid s_t, a_t) \tag{5.29}$$

So, then, as $p(s_1)$ and $p(s_{t+1} \mid a_{t+1})$ do not depend on θ we get for the policy gradient:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[r(\tau) \nabla_\theta \log \pi_\theta(\tau) \right] \\
&= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[r(\tau) \nabla_\theta \Big(\log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t \mid s_t) + \log p(s_{t+1} \mid s_t, a_t) \Big) \right] \\
&= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right]
\end{aligned} \tag{5.30}$$

Equation 5.30 is just another way to express the policy gradient. Now, we can simply collect N trajectories, by letting the agent interact with the environment. Then we can estimate the above quantity by taking the sample mean. [Levine, 2018] Therefore, we have:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t}) \right] \tag{5.31}$$

Having derived $\nabla_\theta J(\theta)$, we can now use it to update θ, as in 5.25. This update rule was used in the original REINFORCE algorithm. [Williams, 1992]. The effect of multiplying by the reward is essentially to make good trajectories more likely, while making bad trajectories less likely.
However, using the total expected reward to compute the estimate of the policy gradient is of high variance. [Levine, 2018] Therefore, Equation 5.30 can be modified. First of all, rewards that were received before an action was taken, should not contribute to how good that action was. [Baxter and Bartlett, 2001] This intuitively makes sense. We can modify Equation 5.31 as follows:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} \sum_{t'=t}^{T} r(s_{t'}, a_{t'}) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \tag{5.32}$$

Now only the rewards starting at t are summed. The quantity $\sum_{t'=t}^{T} r(s_{t'}, a_{t'})$ is called *reward to go*.[Levine, 2018] Note, that the reward to go is the undiscounted version of $Q_\pi(s_t, a_t)$, the action-value function in Equation 3.11. But we will reintroduce the discount factor later in this chapter.
Another way to reduce the variance, is to use a *baseline*, b, which is subtracted from the reward. [Williams, 1992] This results in following modification:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} \Big(\sum_{t'=t}^{T} r(s_{t'}, a_{t'}) - b \Big) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \tag{5.33}$$

For example, we could use the average reward as baseline. This has the effect that trajectories which are more rewarding than usual (i.e. that differ positively from the average reward) become more likely, while less rewarding trajectories than usual become less likely.
Both those tricks, reward to go and baselines significantly reduce the variance.

In general, the policy gradient has the form [Schulman et al., 2016]:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} \Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \tag{5.34}$$

As we have already seen, Ψ_t may be represented by one of the following:

- $\sum_{t=1}^{T} r_t$
- $\sum_{t'=t}^{T} r_{t'}$
- $\sum_{t'=t}^{T} r_{t'} - b$

But Ψ_t can also be represented using the state-value function, action-value function or the advantage function:

- $Q_\pi(s_t, a_t)$
- $A_\pi(s_t, a_t)$
- $r_t + V_\pi(s_{t+1}) - V_\pi(s_t)$ (the TD-Error as in Equation 3.20)

This leads us the next chapter.

5.3.2 Actor-Critic Methods

Actor-critic methods learn to approximate both, the policy and value functions. “Actor” refers to the learned policy and “critic” to the learned value function. The learned value function assigns credit to the policy’s action selection, i.e. the critic criticises the actor. [Sutton and Barto, 1998] Actor and critic can both be represented by a neural network. [Mnih et al., 2016]
As mentioned earlier, the reward to go in Equation 5.32 is also given by the action-value function $Q_\pi(s_t, a_t)$, hence we can write:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} Q_\pi(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \tag{5.35}$$

In practice, the true action-value function is not available, but as we have seen in the last chapter, it can be approximated by a neural network. So we have $Q_\pi(s_t, a_t; w)$, with w the parameters in the network. Furthermore, from Equation 5.33 we also know that we can subtract a baseline. As baseline we

can use the state-value function $V_\pi(s_t)$. [Sutton and Barto, 1998] This gives the advantage function (5.14):

$$A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t) \tag{5.36}$$

Again we have to approximate the value function which gives $V_\pi(s_t; v)$, with parameters v. However, now we would need two different networks to approximate action-value function and state-value function individually. But according to Equation 3.11, we can express $Q_\pi(s_t, a_t) = \mathbb{E}\left[r_{t+1} + V_\pi(s_{t+1}) \mid s_t, a_t\right]$. So, the advantage function can be rewritten in terms of the TD-Error, as used in Equation 3.20:

$$A_\pi(s_t, a_t) = r(s_t, a_t) + V_\pi(s_{t+1}) - V_\pi(s_t) \tag{5.37}$$

This allows us to only use a single set of weights. In general, choosing the advantage function gives the almost lowest possible variance and is therefore a good choice. [Schulman et al., 2016] Using w as the parameters in the network which approximates the state-value function, we get following version of the policy gradient:

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) + V_\pi(s_{t+1}; w) - V_\pi(s_t; w) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \\ &= \mathop{\mathbb{E}}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} A_\pi(s_t, a_t; w) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \end{aligned} \tag{5.38}$$

This is called the *advantage actor critic*. Multiplying by the approximated advantage function has the effect that the probabilities of actions that are better than average are increased, whereas the probabilities of worse-than-average actions are decreased. [Schulman et al., 2016]
So far we have omitted the discount factor γ. However, to allow for infinite horizons, we can reintroduce it in Equation 5.37:

$$A_\pi(s_t, a_t; w) = r(s_t, a_t) + \gamma V_{\pi;w}(s_{t+1}) - V_\pi(s_t; w) \tag{5.39}$$

We can now finally write down the advantage actor-critic algorithm:

Algorithm 7: Advantage actor-critic algorithm

Input: learning rates $\alpha_\theta > 0$ and $\alpha_w > 0$, discount factor γ, number of episodes N

Initialise policy parameter θ

Initialise state-value function parameter w

for $i = 1$ **to** N **do**

  Initialise s

  $I = 1$

  while *s is not terminal* **do**

    $a \sim \pi_\theta(a \mid s)$

    Take a, observe $r(s, a), s'$

    $\delta = r + \gamma V_\pi(s'; w) - V_\pi(s; w)$

    $w = w + \alpha_w \delta \nabla V_\pi(s; w)$

    $\theta = \theta + \alpha_\theta I \delta \nabla \log \pi_\theta(a \mid s)$

    $I = \gamma I$

    $s = s'$

  end

end

Table 7: Advantage actor-critic algorithm, from [Sutton and Barto, 1998]

In addition, we can, as in Section 5.2.5, allow for multi-step returns with n steps. [Mnih et al., 2016] Then we have:

$$A_\pi(s_t, a_t; w) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) + \gamma^n V_{\pi;w}(s_{t+n}) - V_\pi(s_t; w) \tag{5.40}$$

Those are the fundamentals of actor-critic methods.

5.3.3 A3C, A2C

Policy gradient methods are on-policy methods. However, as we know from Section 5.2.1, successive observations are highly correlated and cause unstable learning. Q-Learning methods encountered that issue by maintaining an experience replay. Another approach to decorrelate the agent's experience is to asynchronously execute multiple agents at the same time, on multiple instances of the same environment, while collecting the gradients. After a specified number of time steps the global network parameters are then synchronized. This methods was proposed in [Mnih et al., 2016] and is known as A3C, *asynchronous advantage actor critic*. A3C outperformed Q-Learning

methods in several games of the Atari suite, requiring only half the training time. In addition, A3C not only performed well on Atari games (as previous methods), but also on continuous motor control tasks provided by the MuJoCo physics engine. [Todorov et al., 2012]
As an alternative to the asynchronous approach A3C, researchers came up with a synchronous version called A2C, *synchronous advantage actor critic*, which outperforms the asynchronous one. [Wang et al., 2016a] [Wu et al., 2017a] Instead of performing the parameter updates asynchronously, A2C waits for all individual actors to complete their experience collection and then performs a cumulative update. This is advantageous, because waiting for the experience collected by all actors results in larger batch sizes and GPUs work better with those. As a result, A2C performs better, especially in single-GPU setups. [Wu et al., 2017b] [Dhariwal et al., 2017]
As already mentioned, policy gradient methods, such as A3C or A2C, are in general on-policy. Nevertheless, there exist off-policy versions of policy gradient methods which do maintain an experience replay. Examples are ACER or SAC. [Wang et al., 2016b] [Haarnoja et al., 2018] Even combinations of both, experience replay and distributed experience collection are possible, as shown by Ape-X and IMPALA.[Horgan et al., 2018] [Espeholt et al., 2018] The latter algorithm was even used to beat the best players in StarCraft, as mentioned in the introduction of this thesis. [Vinyals et al., 2019] We cannot discuss these methods in detail, but look forward to exploring them in the future.

5.3.4 Trust Region Policy Optimization - TRPO

Another different approach to policy optimization is called *Trust Region Policy Optimization* [Schulman et al., 2015]. The idea is to constrain the policy updates using the KL-Divergence (also used in 5.2.6) which measures the difference between the old and the new policy. This results in a more stable training process. The optimization problem that has to be solved, is the following:

$$\underset{\theta}{\text{maximize}} \underset{s\sim\rho_{\theta_{old}},a\sim\pi_{\theta_{old}}}{\mathbb{E}} \left[\frac{\pi_\theta(a \mid s)}{\pi_{\theta_{old}}(a \mid s)} A_{\theta_{old}}(s,a)\right] \tag{5.41}$$

Subject to $\mathbb{E}_{s\sim\rho_{\theta_{old}}}[D_{KL}(\pi_{\theta_{old}}(\cdot \mid s) \mid\mid \pi_\theta(\cdot \mid s))] \leq \delta$.
Where θ_{old} are the policy parameters prior to the update. δ is the bound on the KL-divergence and a hyperparameter. The first line is called the *surrogate objective* and is maximized subject to the second line, the *constraint*. The derivation is based on three important concepts: the minorization-maximization (MM) algorithm [Hunter and Lange, 2000], the trust region

and importance sampling. Only replacing the expectations by sample averages and estimating $A_{\theta_{old}}$ remains.
To test the performance of TRPO, the authors tested the algorithm on Atari games and MuJoCo simulations. It turned out, that the algorithm works especially well on locomotion tasks. Among those tasks were swimming, walking and hopping. But TRPO also proved to deliver impressing results playing Atari games. Furthermore, the concepts introduced in TRPO were highly influential and motivated other algorithms, such as Proximal Policy Optimization.

5.3.5 Proximal Policy Optimization - PPO

The final algorithm we would like to discuss is called Proximal Policy Optimization (PPO). [Schulman et al., 2017] PPO builds on top of the ideas discussed in TRPO. Whereas TRPO is complicated, PPO is much simpler to implement, more general and has better sample complexity.
To derive the PPO algorithm, we have to modify the equation used for TRPO as follows. First we denote the probability ratio $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$, with $r_t(\theta_{old}) = \frac{\pi_{\theta_{old}}(a|s)}{\pi_{\theta_{old}}(a|s)} = 1$. Intuitively, if $r_t(\theta) > 1$, then the action is more likely following the new policy, than before when following the old policy and vice versa. Then the surrogate objective maximized in Equation 5.41 is given by:

$$L^{CPI}(\theta) = \mathop{\mathbb{E}}_{s \sim \rho_{\theta_{old}}} \left[r_t(\theta) A_{\pi_{\theta_{old}}} \right] \tag{5.42}$$

Where CPI means Conservative Policy Iteration. [Kakade and Langford, 2002] Instead of using a hard constraint as in TRPO, the authors propose to modify the objective as:

$$L^{CLIP}(\theta) = \mathop{\mathbb{E}}_{s \sim \rho_{\theta_{old}}} \left[\min(r_t(\theta) A_{\pi_{\theta_{old}}}, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) A_{\pi_{\theta_{old}}}) \right] \tag{5.43}$$

Here ϵ is a hyperparameter. Clipping the probability ratio $r_t(\theta)$ ensures that the policy does not change too much at once. Additionally, taking the minimum has the effect, that either $r_t(\theta) A_{\pi_{\theta_{old}}}$ or $(1 - \epsilon) A_{\pi_{\theta_{old}}}$ acts as the lower, pessimistic, bound.
The PPO algorithm is shown in Table 5.3.5.

Algorithm 8: PPO

Input: clipping parameter ϵ, number of iterations N_{iter}, number of actors N_{act}

for *iteration* $= 1$ **to** N_{iter} **do**
  for *actor* $= 1$ **to** N_{act} **do**
    Run policy π_{old} in environment for T steps
    Compute advantage estimates $A_1, ..., A_T$
  end
  Optimize surrogate L w.r.t. θ, with K epochs and minibatch size $M < NT$
  $\theta_{old} = \theta$
end

Table 8: PPO, from [Schulman et al., 2017]

Proximal Policy Optimization delivered impressive results in various locomotion tasks and outperformed TRPO.
In a later paper, OpenAI used PPO "to learn dexterous in-hand manipulation policies which can perform vision-based object reorientation on a physical Shadow Dexterous Hand." In short, they trained a human-like robotic arm to manipulate real objects in the physical world. [OpenAI et al., 2018] (A video is available at: `https://www.youtube.com/watch?v=jwSbzNHGflM&feature=youtu.be`)
Furthermore, a scaled-up version of PPO was used to power OpenAI's Dota playing AI which outperformed the world's top human players. [OpenAI, 2018] Here, PPO proved its scalability. Their system was running on 256 GPUs and 128.000 CPU cores. OpenAI Five, as they call it, learned entirely by playing against itself (and previous versions of itself), starting with completely random parameters. Every day it accumulated 180 years worth of games. [OpenAI, 2018]
On the one hand these examples show that Deep Reinforcement Learning methods can be massively scalable and that they are able to learn highly complex strategies. On the other hand, however, this demonstrates how inefficient the current methods are. Imagine how a human being would play if he/she had the ability to train for 180 years a day.

6 Conclusion

In Chapters 3 and 4 the fundamentals of Reinforcement Learning and Deep Learning were discussed. The topics covered in those chapters are essential prerequisites for the last chapter, Deep Reinforcement Learning. There some of the state-of-the-art algorithms were discussed. Of course, Deep Reinforcement Learning is a very comprehensive field and includes various further domains that were not part of this thesis. I could not discuss these, but look forward to exploring them in the future. However, I believe, that my thesis on the one hand explains the core components of Deep RL and on the other hand also covers the most important innovations of recent years. After all, I believe it serves as a good starting point to enter the field. During my personal journey on writing "Learning Deep Reinforcement Learning", I certainly did learn how these techniques work. Therefore, I am very grateful to my supervisor, Vadim Savenkov, who allowed and encouraged me to write this thesis.

Currently, Deep Reinforcement Learning and the field of Artificial Intelligence in general is ongoing research and moves at a fast pace. At its core, these methods have been inspired by humans, animals, their behaviour and most importantly their biological brains. Nowadays, research conducted in Artificial Intelligence, in turn, also sparks new ideas in other areas, such as psychology and neuroscience. [Botvinick et al., 2019] [Banino et al., 2018] Looking into the future, I believe that Artificial Intelligence will enable us to better understand our own minds. Scientists say that the human brain is the most complex structure known. Therefore, it is quite natural to believe, that the mystery of human intelligence might only be solved once we have understood the artificial one. For this reason, working on artificial intelligence implies working on understanding the most complex structure we know of. I think this is a quite motivating thought.

When I first learned about AI about two and a half years ago, I got into the area of Machine Learning and especially Deep Learning. While writing this thesis, I extensively studied Reinforcement Learning. Only recently I was also introduced to more structured approaches, such as knowledge graphs. I believe that a combination of those three approaches (and possibly others) will lead to powerful AI applications. After all, human intelligence does not seem to be based on a single algorithm. In general, however, I think that Deep Reinforcement Learning provides the right framework (interaction, sequential decision making) to most likely combine them all. Overall, Deep Reinforcement Learning is an exciting approach to Artificial Intelligence and I look forward to pursuing this path.

References

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Abdel-Hamid et al., 2014] Abdel-Hamid, O., Mohamed, A., Jiang, H., Deng, L., Penn, G., and Yu, D. (2014). Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(10):1533–1545.

[Achiam, 2018] Achiam, J. (2018). OpenAI SpinningUp. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`.

[Banino et al., 2018] Banino, A., Barry, C., Uria, B., Blundell, C., Lillicrap, T., Mirowski, P., Pritzel, A., Chadwick, M., Degris, T., Modayil, J., Wayne, G., Soyer, H., Viola, F., Zhang, B., Goroshin, R., Rabinowitz, N., Pascanu, R., Beattie, C., Petersen, S., and Kumaran, D. (2018). Vector-based navigation using grid-like representations in artificial agents. *Nature*, 557.

[Baxter and Bartlett, 2001] Baxter, J. and Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319â€“350.

[Bellemare et al., 2017] Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning.

[Bellemare et al., 2015] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2015). The arcade learning environment: An evaluation platform for general agents. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 4148–4152. AAAI Press.

[Bellman, 1954] Bellman, R. (1954). The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515.

[Bellman, 1957] Bellman, R. (1957). A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684.

[Botvinick et al., 2019] Botvinick, M., Ritter, S., X. Wang, J., Kurth-Nelson, Z., Blundell, C., and Hassabis, D. (2019). Reinforcement learning, fast and slow. *Trends in Cognitive Sciences*, 23.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).*

[Dayan and Niv, 2008] Dayan, P. and Niv, Y. (2008). Reinforcement learning: The good, the bad and the ugly. *Current Opinion in Neurobiology*, 18(2):185 – 196. Cognitive neuroscience.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding.

[Dhariwal et al., 2017] Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. (2017). Openai baselines. https://github.com/openai/baselines.

[Espeholt et al., 2018] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. (2018). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures.

[Fortunato et al., 2017] Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. (2017). Noisy networks for exploration.

[Glorot et al., 2011] Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G., Dunson, D., and DudÃ-k, M., editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA. PMLR.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Haarnoja et al., 2018] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor.

[Hasselt, 2010] Hasselt, H. V. (2010). Double q-learning. In Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S., and Culotta, A., editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc.

[Hasselt et al., 2016] Hasselt, H. v., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press.

[Heess et al., 2017] Heess, N., TB, D., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., Erez, T., Wang, Z., Eslami, S. M. A., Riedmiller, M., and Silver, D. (2017). Emergence of locomotion behaviours in rich environments.

[Hessel et al., 2017] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Horgan et al., 2018] Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. (2018). Distributed prioritized experience replay.

[Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366.

[Howard, 1960] Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.

[Hunter and Lange, 2000] Hunter, D. R. and Lange, K. (2000). Quantile regression via an mm algorithm. *Journal of Computational and Graphical Statistics*, 9(1):60–77.

[Kakade and Langford, 2002] Kakade, S. and Langford, J. (2002). Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pages 267–274, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

[LeCun et al., 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Comput.*, 1(4):541–551.

[Levine, 2018] Levine, S. (2018). Cs 294 112: Deep reinforcement learning, lecture notes.

[Li et al., 2019] Li, Y., Chen, Y., Wang, N., and Zhang, Z. (2019). Scale-aware trident networks for object detection.

[Lin, 1992] Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321.

[Mcculloch and Pitts, 1943] Mcculloch, W. and Pitts, W. (1943). A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1312.5602.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–533.

[Nielsen, 2018] Nielsen, M. A. (2018). Neural networks and deep learning.

[OpenAI, 2018] OpenAI (2018). Openai five. `https://blog.openai.com/openai-five/`.

[OpenAI et al., 2018] OpenAI, Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., Schneider, J., Sidor, S., Tobin, J., Welinder, P., Weng, L., and Zaremba, W. (2018). Learning dexterous in-hand manipulation.

[Park et al., 2019] Park, D. S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E. D., and Le, Q. V. (2019). SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition. *arXiv e-prints*, page arXiv:1904.08779.

[Paszke et al., 2017] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

[Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

[Raval, 2017] Raval, S. (2017). Which activation function should i use? `https://youtu.be/-7scQpJT7uo`.

[Real et al., 2018] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–.

[Schaul et al., 2016] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. *CoRR*, abs/1511.05952.

[Schulman et al., 2015] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015). Trust region policy optimization.

[Schulman et al., 2016] Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv e-prints*, page arXiv:1712.01815.

[Sutton, 1988] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.

[Sutton et al., 1999] Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA. MIT Press.

[Todorov et al., 2012] Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *IROS*, pages 5026–5033. IEEE.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M., Jaderberg, M., Czarnecki, W. M., Dudzik, A., Huang, A., Georgiev, P., Powell, R., Ewalds, T., Horgan, D., Kroiss, M., Danihelka, I., Agapiou, J., Oh, J., Dalibard, V., Choi, D., Sifre, L., Sulsky, Y., Vezhnevets, S., Molloy, J., Cai, T., Budden, D., Paine, T., Gulcehre, C., Wang, Z., Pfaff, T., Pohlen, T., Wu, Y., Yogatama, D., Cohen, J., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Apps, C., Kavukcuoglu, K., Hassabis, D., and Silver, D. (2019). AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`.

[Wang et al., 2016a] Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. (2016a). Learning to reinforcement learn.

[Wang et al., 2016b] Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016b). Sample efficient actor-critic with experience replay.

[Wang et al., 2016c] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. (2016c). Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1995–2003. JMLR.org.

[Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256.

[Wu et al., 2017a] Wu, Y., Mansimov, E., Liao, S., Grosse, R., and Ba, J. (2017a). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation.

[Wu et al., 2017b] Wu, Y., Mansimov, E., Liao, S., Radford, A., and Schulman, J. (2017b). Openai baselines acktr, a2c. `https://openai.com/blog/baselines-acktr-a2c/`.

[Zeghidour et al., 2018] Zeghidour, N., Xu, Q., Liptchinsky, V., Usunier, N., Synnaeve, G., and Collobert, R. (2018). Fully convolutional speech recognition.

A Algorithms

Algorithm 9: Policy Iteration

Input: threshold θ
Output: policy $\pi \approx \pi_*$
1. Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in (S)$
2. Policy Evaluation
repeat
    $\Delta \leftarrow 0$
    for $s \in \mathcal{S}$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end
until $\Delta < \theta$
3. Policy Improvement
$stable \leftarrow true$
for $s \in \mathcal{S}$ **do**
    $old - action \leftarrow \pi(s)$ **for** $a \in \mathcal{A}(s)$ **do**
        $Q(s, a) \leftarrow \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$
    end
    $\pi(s) \leftarrow arg\max_a Q(s, a)$
    if $old - action \neq \pi(s)$ **then**
        $stable \leftarrow false$
end
if *policystable* **then return** $V \approx V_*$ and $\pi \approx \pi_*$
else Go to 2

Table 9: Policy iteration

Algorithm 10: Value iteration

Input: policy π , threshold parameter θ
Output: $V \approx v_\pi$
Initialize $V(s)$ arbitrarily, $\forall s \in \mathcal{S}^+$, $V(s) = 0$
repeat
  $\Delta \leftarrow 0$
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end
until $\Delta < \theta$
return $\pi \approx \pi_*$, s.t. $\pi(s) = arg \max_a \sum_{s',r} p(s', r \mid s, a) [r + \gamma V(s')]$

Table 10: Value iteration

Algorithm 11: First Visit Monte Carlo Prediction for action values

Input: policy π , *num_ of_ episodes*
Output: $Q \approx q_\pi$
Initialize $Q(s, a)$ arbitrarily, $\forall s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
for $i \leftarrow 0$ **to** $num_of_episodes$ **do**
  Generate episode following π:
   $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for $t \leftarrow 0$ **to** $T - 1$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if S_t *in* $S_0, S_1, ..., S_{t-1} and A_t$ *in* $A_0, A_1, ..., A_{t-1}$ **then**
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$
  end
end
return Q

Table 11: First Visit Monte Carlo Prediction for action values

Algorithm 12: Monte Carlo Control with exploring starts

Input: $num_of_episodes$
Output: $Q \approx q_\pi$
Initialize policy $\pi \in \mathcal{A}$ arbitrarily $\forall s \in \mathcal{S}$
Initialize $Q(s, a)$ arbitrarily, $\forall s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
for $i \leftarrow 0$ **to** $num_of_episodes$ **do**
  Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(\mathcal{S})$ randomly s.t. all pairs have $probability > 0$
  Generate episode from S_0, A_0 following π: $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for $t \leftarrow 0$ **to** $T - 1$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if $S_t \ in \ S_0, S_1, ..., S_{t-1} and A_t \ in \ A_0, A_1, ..., A_{t-1}$ **then**
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$
      $\pi(S_t) \leftarrow arg \max_a Q(S_t, a)$
  end
end
return π

Table 12: Monte Carlo Control with exploring starts

Algorithm 14: Q-Learning

Input: step-size $\alpha \in (0, 1]$, *num_ of_ episodes*
Output: $Q \approx q_\pi$
Initialize $Q(s, a)$ arbitrarily, $\forall s \in \mathcal{S}_+, a \in \mathcal{A}(s)$
for $i \leftarrow 0$ **to** $num_of_episodes$ **do**
  Initialize S
  foreach *step in episode* **do**
    Choose A at S using Q
    Take action A, observe R and S'
    Choose A' at S' using Q
    $Q(S, A) \leftarrow Q(S, A) + \alpha\,[R_{t+1} + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
    $A \leftarrow A'$
  end
end
return Q

Table 14: Q-Learning

Algorithm 13: Sarsa

Input: step-size $\alpha \in (0, 1]$, *num_ of_ episodes*
Output: $Q \approx Q_\pi$
Initialize $Q(s, a)$ arbitrarily, $\forall s \in \mathcal{S}_+, a \in \mathcal{A}(s)$
for $i \leftarrow 0$ **to** $num_of_episodes$ **do**
  Initialize S
  Choose A at S using Q
  foreach *step in episode* **do**
    Take action A, observe R and S'
    Choose A' at S' using Q
    $Q(S, A) \leftarrow Q(S, A) + \alpha\,[R_{t+1} + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$
    $A \leftarrow A'$
  end
end
return Q

Table 13: Sarsa