Titel der Arbeit

Bachelorarbeit der Fakultät für Physik
der
Ludwig-Maximilians-Universität München

vorgelegt von
Jonas C. Märtens
geboren in Hamburg

München, den 01.02.2024

Contents

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Chapter 1

Introduction

1.1 Artificial Intelligence

Artificial Intelligence (AI) is a broad subject of study that can be defined in different ways [1, chapter 1]. John McCarthy, often called the "father of AI" [2, 3, 4], defines it as "the science and engineering of making intelligent machines" [5], where intelligence means "the computational part of the ability to achieve goals in the world" [5]. AI is sometimes mistakenly used interchangeably with Machine Learning. Machine learning is a subset of AI concerned with enabling AI systems to learn from experience [1, chapter 1]. Machine learning enables the development of large-scale AI systems as they are used today.
The study of artificial intelligence was first proposed by McCarthy et al. in late 1955 [6]. It went through two major hype cycles in the sixties and the eighties [7, 2, 8] followed by phases of "AI winter". The current (as of early 2024) AI boom, sometimes also called "AI spring" [9] was started by groundbreaking advances in speech recognition [10] and image classification [11] in 2012 [12, 13] and reached the public at the latest in late 2022, following the release of ChatGPT [14], a multipurpose AI-chatbot, open to everyone [15].
These breakthroughs are made possible mainly by advancements in the field of machine learning, enabling AI systems to learn from huge amounts of data. In addition, the exponential increase in computation and storage capabilities as predicted by Moore's Law [16], algorithms like backpropagation [17] allowed incorporating large amounts of data into machine learning models in realistic amounts of time.
Today, AI systems are indispensable in many areas such as web search engines [18], recommendation systems [19], human speech recognition and generation [20, 10], image recognition and generation [21, 11] and personal assistants [14] and surpasses humans in high level strategy games like go and chess [22, 23] as well as other video games [24].

Chapter 2

Neural Networks

2.1 Overview

At the heart of almost all the technologies mentioned in the last paragraph are deep artificial neural networks. The next section will outline the mathematical details of how these systems work and learn. Although the comparison of artificial neural networks, from now on just called "neural networks", to their biological counterpart can be criticized as oversimplifying the inner workings of biological brains [25, chapter 1.1], the architecture of neural networks is heavily inspired by how decision-making and learning work in the human brain [25, 26, chapter 1.1]. I will illustrate the basic principles of neural networks at the example of a network that detects the gender of a person by looking at pictures.

A neural network consists of a number of layers of artificial neurons, called *nodes*. In a *fully connected* network, each node is connected to every node in the next layer. The connection strengths are called *weights*. An image of a person can be fed into the network by setting the *activation* values of the first layer of the network, the *input layer*, to the individual pixel values of the image. This information is then fed forward through the layers of the network until the *output layer* is reached. If a network has at least one layer in between the input and output layer, it is called a *deep neural network*. These intermediate layers are called *hidden layers*. If the outputs of each layer are only connected to the inputs of the next layer, the network is called a *feedforward neural network*. If *feedback* connections are allowed, the network is called a *recurrent neural network*.
In our example, the output layer should consist of only two nodes. If the activation of the first node is larger than the activation of the second node, the network thinks that the person in the picture is a male. If on the other hand the second node has a larger activation, the network classifies this person as female [25, chapter 1.2].

In order to make accurate predictions, a reasonable set of network parameters (i.e.the weights) has to be found. This is done by training the network with pre-classified images. After an image has been processed by the network, the output is compared to the correct classification and the network parameters are updated in a way that would improve the networks output if the same image was to be processed again [25, 27, chapter 1.2].
This is similar to how humans learn from experience. If we were to misclassify a persons gender, the unpleasant social experience that may come with that mistake would cause us to update our internal model of what different genders look like to not make the same mistake again.

One of the main strengths of neural networks is their ability to *generalize* [28]. When a network was trained on a large enough set of examples, it gains the ability to generalize this

knowledge to examples that were previously unseen. The gender classification network from our example doesn't just memorize the genders of the people it has seen, but instead learns about the features that help to identify the gender of a random person.

The problem of image classification is a rather complex one. One wouldn't typically think of it as finding a function that maps the values of each input pixel to the classification output. But even very complex problems can be modeled by equally complex functions. The universal approximation theorem states, that a feedforward neural network with at least one hidden layer with appropriate activation functions (see 2.2.3 for details) can approximate any continuous function if given enough nodes [29, chapter 6.4.1]. That's why training a neural network can be thought of as fitting the network to the training data.

2.2 Mathematical Details

The following section will outline the mathematical details of how neural networks work. The definitions and derivations are based on [25, chapter 1.2-1.3], [29, chapter 5-6], [27] as well as [30, chapter 4.4].

The most complete treatment of the mathematical details of neural networks is arguably given by the framework of computational graphs [31]. In this framework, a neural network is treated as single that maps a set of input values to a set of output values. This function is composed of individual mathematical operations and can be represented as a directed graph [30, section 1.4]. I will not rigorously define the framework of computational graphs here, as this is beyond the scope of this thesis. Instead, I will explain the principles of neural networks starting from a single neuron and then build up to a fully connected neural network.

2.2.1 Notation

The following notation will be used throughout this section:

$\boldsymbol{x}$	:	input vector of the neural network
$\boldsymbol{a}$	:	activation vector of a layer
$\boldsymbol{z}$	:	pre-activation vector of a layer
$\boldsymbol{y}$	:	output vector of the neural network
$\boldsymbol{b}$	:	bias vector of a layer
x_i, a_i, z_i, y_i, b_i	:	individual elements of the respective vectors, for individual nodes
$\boldsymbol{w_i}$	:	weight vector of all weights connected to neuron i
w_{ij}	:	weight of the connection from neuron i of a layer to neuron j of the previous layer
W	:	weight matrix of a layer. Contains rows $\boldsymbol{w_i}$
$^{[J]}$	:	superscript denoting the layer of a variable

2.2.2 Single Node

Before we can build a neural network out of nodes, we have to define how a single node works. Each node receives the activations a_i from the nodes of the previous layer as inputs. Each connection is assigned a weight w_i, stored in the node's weight vector $\boldsymbol{w}$.
Additionally, each node has a so-called *bias* b. The bias shifts the net input of the node by a constant value. It is needed to model certain problems where part of the prediction is independent of the input [25, p. 6]. Examples include all problems where the output should not be zero even if all inputs are zero.

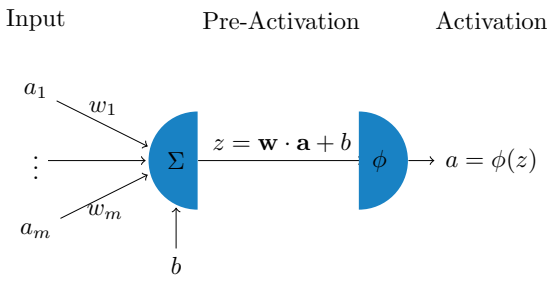


Figure 2.1: A single node of a neural network. To get the activation a of the node, the pre-activation z is calculated from the inputs a_i and the bias b and is passed through an activation function ϕ.

The net input, called *pre-activation value* z of a node is the weighted sum of all inputs plus the bias:

$$z = \sum_{i=1}^{m} w_i a_i + b = \boldsymbol{w} \cdot \boldsymbol{a} + b. \tag{2.1}$$

The pre-activation value is then passed through an *activation function* ϕ to get the activation a of the node, which is then passed on to the next layer, where the process is repeated:

$$a = \phi(z). \tag{2.2}$$

The whole process is illustrated in figure 2.1.

2.2.3 Activation Functions

The activation function ϕ is used to introduce non-linearity into the network and thus increasing its modeling power [25, section 1.2.1.3]. Some activation functions are also referred to as *squashing function* [30, p. 10], as they map the unbounded pre-activation value z to a bounded activation value a. The choice of activation function has a large impact on the performance of the network in terms of both accuracy and speed [32]. The type of function heavily influences the way that information is processed by the network and the complexity of the function naturally has a large impact on the computational cost of the network. The best choice therefore depends on the problem that is being solved and the architecture of the network. Typically, the same activation function is used for all nodes in a layer and is applied to the pre-activation value of each node individually, but different layers can use different activation functions depending on their purpose [29, p. 174]. For a long time, the most popular activation functions were ([29, chapter 6.3], [25, section 1.2.1.3]) the sigmoid function:

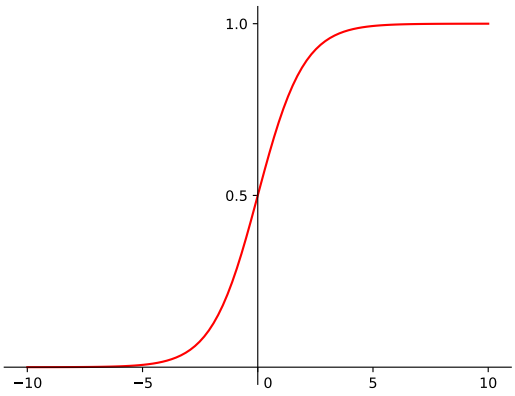
$$\phi(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

and the hyperbolic tangent function:

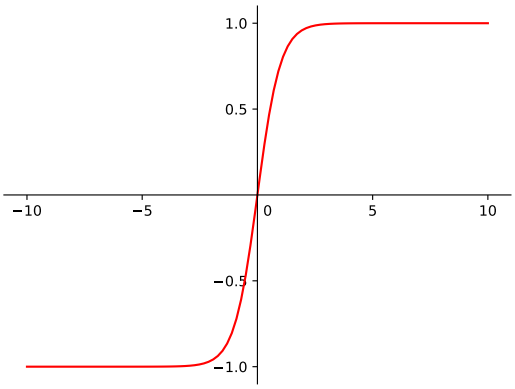
$$\phi(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1 \tag{2.4}$$

as well as the sign function:

$$\phi(z) = \operatorname{sign}(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z = 0 \\ -1 & \text{if } z < 0. \end{cases} \tag{2.5}$$



(a) The sigmoid activation function.



(b) The hyperbolic tangent activation function.

Figure 2.2: The most popular activation functions before the rise of ReLU.

The sign function can map neural network output to a binary classification, but it is not suitable for backpropagation (see section 2.2.6) due to its derivative being zero almost everywhere. The sigmoid function and the hyperbolic tangent function are both differentiable and limit the output to the range $(-1, 1)$ and $(0, 1)$ respectively. They are however more computationally expensive than other activation functions and suffer from the *vanishing gradient problem*. The vanishing gradient problem is caused by the fact that the derivative of the sigmoid function approaches zero for large absolute values of z. This leads to the weights of the nodes in the first layers of the network being updated very slowly, as the gradient of the loss function with respect to the weights of these nodes is very small (see section 2.2.6)[25, section 1.4.2][32]. The sigmoid function and the hyperbolic tangent function can be seen in figure 2.2a and 2.2b.

In recent years, the *rectified linear unit* (ReLU) and similar stepwise linear functions have become the go-to activation functions for deep neural networks [29, chapter 6.3.2][32]. ReLU is defined as:

$$\phi(z) = \max(0, z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0. \end{cases} \tag{2.6}$$

Its main advantage is its very low computational cost, as it consists of only a single comparison. Although it is not as prone to the vanishing gradient problem as the sigmoid and the hyperbolic tangent, the problem still exists for negative values of z. This has been addressed by variations like *Leaky ReLU*, introducing a small but non-zero slope for negative values [32]:

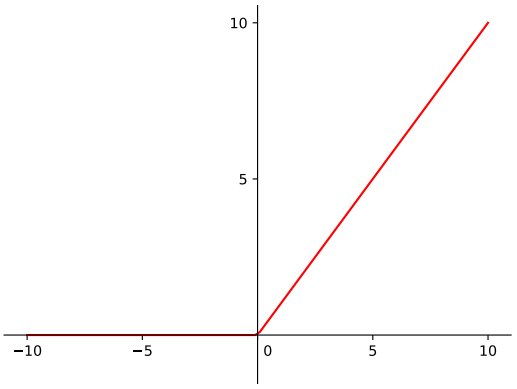
$$\phi(z) = \max(0.01z, z) = \begin{cases} 0.01z & \text{if } z \leq 0 \\ z & \text{if } z > 0. \end{cases} \tag{2.7}$$

The ReLU function and its leaky version can be seen in figures 2.3a and 2.3b.

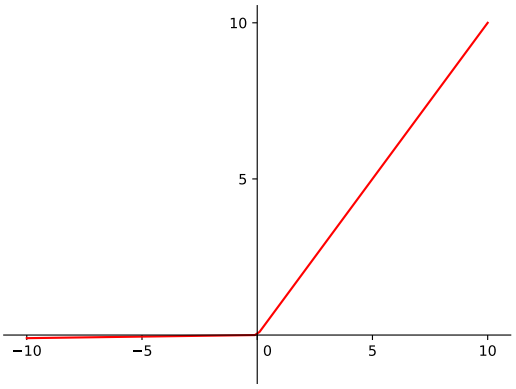
2.2.4 Forward-Propagation in Feedforward Neural Networks

Now that the workings of the nodes are defined, we can build a fully connected neural network out of these building blocks. I will illustrate the process at the example of a feedforward neural network as defined in section 2.1. An example of such a network can be seen in figure 2.4.

The process of feeding an input vector $\boldsymbol{x}$ through the network to get the output vector $\boldsymbol{y}$ is called *forward-propagation*, as the information propagates through the network layer. The



(a) The rectified linear unit (ReLU) activation function.



(b) The leaky rectified linear unit (Leaky ReLU) activation function.

Figure 2.3: Modern, stepwise linear activation functions.

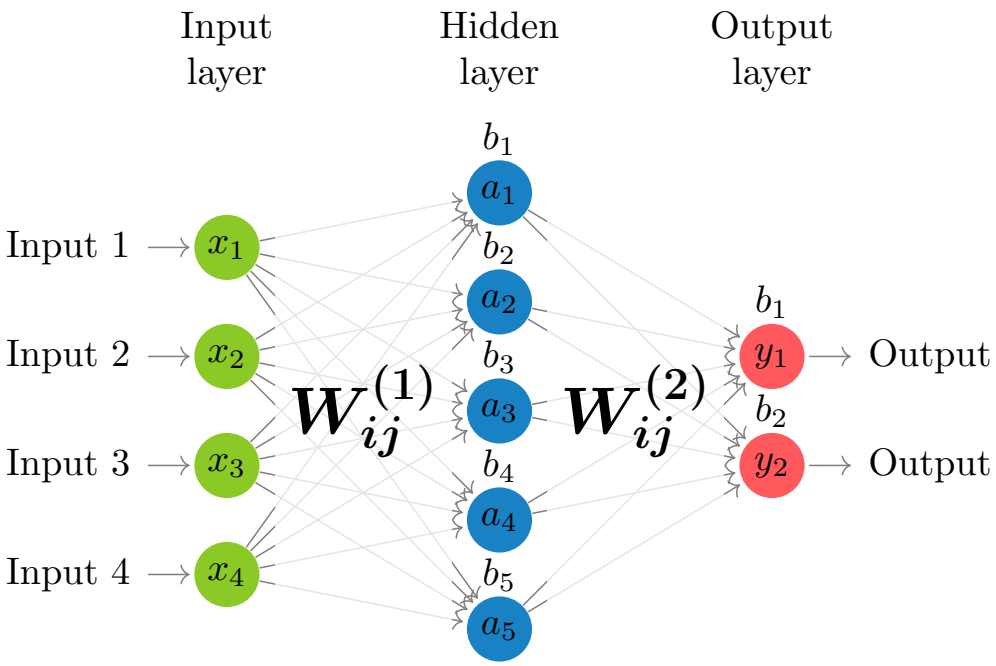


Figure 2.4: A feedforward neural network with one hidden layer.

activation of any layer $\boldsymbol{a}^{[J]}$ can be calculated from the activation of the previous layer $\boldsymbol{a}^{[J-1]}$ by generalizing equation 2.1 and 2.2 to the vector case:

$$z_k^{[J]} = \sum_{i=1}^{m} w_{ki}^{[J]} a_i^{[J-1]} + b_k^{[J]} \implies \boldsymbol{z}^{[J]} = W^{[J]}\boldsymbol{a}^{[J-1]} + \boldsymbol{b}^{[J]}, \tag{2.8}$$

$$a_k^{[J]} = \phi(z_k^{[J]}) \implies \boldsymbol{a}^{[J]} = \phi(\boldsymbol{z}^{[J]}), \tag{2.9}$$

where the activation function $\phi : \mathbb{R}^m \to \mathbb{R}^m$ is applied element-wise. The activation of the input layer $\boldsymbol{a}^{[0]}$ is simply the input vector $\boldsymbol{x}$ and the activation of the output layer $\boldsymbol{a}^{[L]}$ is the output vector $\boldsymbol{y}$.

Equations 2.8 and 2.9 can be applied recursively to calculate the activation of each layer from the input layer to the output layer:

$$\begin{aligned}
\boldsymbol{y} = \boldsymbol{a}^{[L]} &= \phi(W^{[L]}\boldsymbol{a}^{[L-1]} + \boldsymbol{b}^{[L]}) && (2.10)\\
&= \phi(W^{[L]}\phi(W^{[L-1]}\boldsymbol{a}^{[L-2]} + \boldsymbol{b}^{[L-1]}) + \boldsymbol{b}^{[L]}) && (2.11)\\
&= \phi(W^{[L]}\phi(W^{[L-1]}\phi(\dots\phi(W^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]})\dots) + \boldsymbol{b}^{[L-1]}) + \boldsymbol{b}^{[L]}). && (2.12)
\end{aligned}$$

This equation also shows the significance of the activation function. Without it, the whole network would be equivalent to a single layer and could be replaced by a single matrix multiplication. It would therefore only be able to model linear functions. To show this, let's set $\phi(z)$ to the identity in equation 2.12. This yields:

$$\begin{aligned}
\boldsymbol{y} &= W^{[L]}W^{[L-1]}\dots W^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[L]} + W^{[L]}\boldsymbol{b}^{[L-1]} + \dots + W^{[L]}W^{[L-1]}\dots W^{[2]}\boldsymbol{b}^{[1]} && (2.13)\\
&= \widetilde{W}\boldsymbol{x} + \tilde{\boldsymbol{b}}. && (2.14)
\end{aligned}$$

2.2.5 Loss Functions and Gradient Descent

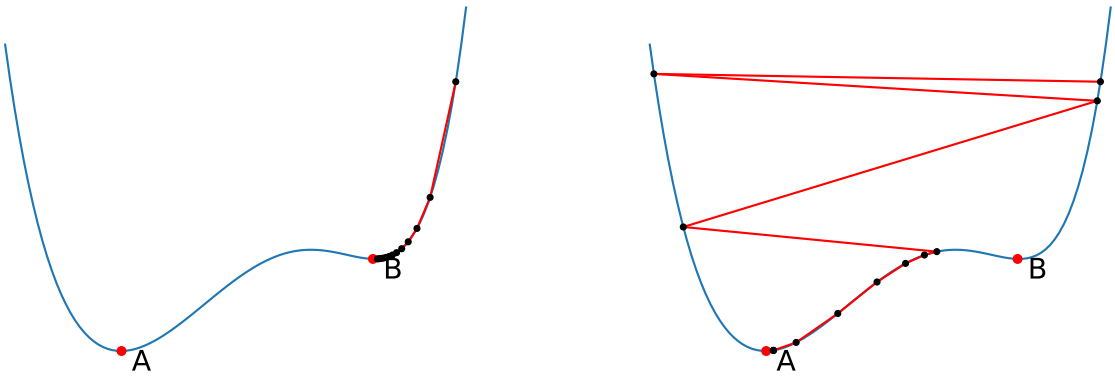
In order to produce meaningful output, the network's weights and biases have to be adjusted to minimize the error of the network's output. This process is called *training* the network. The error of the network is measured by a *loss function* $\lambda(\boldsymbol{y}, \hat{\boldsymbol{y}})$, where $\hat{\boldsymbol{y}}$ is some target output vector. The loss function is a measure of how far the network's output $\boldsymbol{y}$ is from the target output $\hat{\boldsymbol{y}}$. As we want to minimize the network's error, the training process is essentially an optimization problem [29, chapter 4.3]. Let's step back from neural networks for a moment and look at a method to minimize a function $f(\boldsymbol{x})$ with respect to its parameters $\boldsymbol{x}$. The most common method to do this in machine learning is *gradient descent* [29, chapter 4.3].

If we imagine the function $f(\boldsymbol{x})$ as a landscape, the goal of gradient descent is to find the lowest point of the landscape. To do this, the algorithm starts at some point $\boldsymbol{x}_0$ and in each iteration, it takes a step in the direction of the steepest descent. The size of the step is determined by the *learning rate* η. Choosing an appropriate learning rate is crucial for the algorithm to converge. Figure 2.5 shows gradient descent with different learning rates and the resulting paths through the landscape.

As is known from multivariable calculus, the direction of the steepest descent of a function $f(\boldsymbol{x})$ is given by the negative gradient $\nabla f(\boldsymbol{x})$. We can therefore update the parameters $\boldsymbol{x}$ in each iteration by:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \eta\nabla f(\boldsymbol{x}_n). \tag{2.15}$$

After a sufficient number of iterations, the algorithm will converge to a local minimum of the function. In the region around the minimum, the gradient is close to zero and the algorithm



(a) Gradient descent with a small learning rate.

(b) Gradient descent with a large learning rate.

Figure 2.5: Gradient descent with different learning rates. Choosing the learning rate too small can lead to slow convergence and to being stuck in local minima. Choosing the learning rate too large can lead to *overshooting* and to the algorithm diverging. Here, the algorithm still converges, but the large oscillations slow down the convergence.

Algorithm 1 Gradient Descent

Require:
$f(\boldsymbol{x})$: Function to minimize
η: Learning rate
ϵ: Threshold
$\boldsymbol{x}_0$: Initial parameters
Output: $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$: Parameters that minimize $f(\boldsymbol{x})$

1: $\boldsymbol{x} \leftarrow \boldsymbol{x}_0$ ▷ Initialize parameters
2: **while** $\|\nabla f(\boldsymbol{x}_n)\| > \epsilon$ **do** ▷ Until convergence
3: $\quad \boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f(\boldsymbol{x})$ ▷ Update parameters
4: **return** $\boldsymbol{x}$ ▷ Return parameters

will not change the parameters significantly anymore. We therefore define a threshold ϵ and stop the algorithm if the norm of the gradient falls below this threshold. The gradient descent algorithm is summarized in algorithm 1.

Going back to neural networks, the function that we want to minimize is the loss function $\lambda(\boldsymbol{y}, \hat{\boldsymbol{y}})$. As the loss function depends on the network's output $\boldsymbol{y}$, which in turn depends on the network's parameters $\boldsymbol{w}$ and $\boldsymbol{b}$, the loss function is a function of the network's parameters $\lambda(\boldsymbol{w}, \boldsymbol{b})$.

For single layer networks, the calculation of the gradients is straightforward. For multi-layer networks however, the loss function depends on the parameters of earlier layers in a non-trivial way. The next section will outline an algorithm, that allows the efficient calculation of the gradients of multi-layer networks.

2.2.6 The Backpropagation Algorithm

The calculation of the gradients is the most computationally expensive part of training a neural network. The invention of the *backpropagation* algorithm by Rumelhart et al. in 1986 [17] was a major breakthrough in the field of neural networks, as it allowed very efficient gradient calculation and thus enabled the training of large neural networks.
The idea behind backpropagation is to propagate the error back through the network after each forward-propagation step using *local gradients* δ. During the forward-propagation step, the activations of each layer are stored in memory and used to calculate the derivatives need for the backpropagation step. This is a form of *automatic differentiation* [33, 34] and allows the calculation of the gradients with the same time complexity as the forward-propagation step. This is sometimes called the *cheap gradient principle* [35]. The following mathematical derivation of the backpropagation algorithm is based on [30, chapter 4.4] as well as [29, chapter 6.5].

To calculate the update of a weight $w_{ij}^{[K]}$, we have to calculate the derivative of the loss function λ with respect to the weight $w_{ij}^{[K]}$:

$$\Delta w_{ij}^{[K]} = \eta \cdot \frac{\partial \lambda}{\partial w_{ij}^{[K]}}. \tag{2.16}$$

This derivative can be evaluated using the chain rule:

$$\frac{\partial \lambda}{\partial w_{ij}^{[K]}} = \frac{\partial \lambda}{\partial a_i^{[K]}} \frac{\partial a_i^{[K]}}{\partial z_i^{[K]}} \frac{\partial z_i^{[K]}}{\partial w_{ij}^{[K]}} = \delta_i^{[K]} a_j^{[K-1]} \tag{2.17}$$

where we used equation 2.8 to simplify the last derivative and defined the local gradient $\delta_j^{[K]}$ as:

$$\delta_i^{[K]} = \frac{\partial \lambda}{\partial a_i^{[K]}} \frac{\partial a_i^{[K]}}{\partial z_i^{[K]}}. \tag{2.18}$$

Let's take a closer look at the local gradient $\delta_i^{[K]}$. The second factor in equation 2.18 is the derivative of the activation function ϕ with respect to the pre-activation value $z_i^{[K]}$. This term is straightforward to calculate. Remembering that the activation function is applied element-wise, using equation 2.9 we get:

$$\frac{\partial a_i^{[K]}}{\partial z_i^{[K]}} = \frac{\partial \phi(z_i^{[K]})}{\partial z_i^{[K]}} = \phi'(z_i^{[K]}). \tag{2.19}$$

The first factor in equation 2.18 is the derivative of the loss function λ with respect to the activation $a_i^{[K]}$. If layer K is the output layer, this derivative is simply the derivative of the loss function with respect to the output value y_i:

$$\frac{\partial \lambda}{\partial a_i^{[K]}} = \frac{\partial \lambda}{\partial y_i}. \tag{2.20}$$

If layer K is not the output layer, the derivative is slightly more complicated. We can use the chain rule trick again, re-introducing the activation values $a_n^{[K+1]}$ of the next layer, which all depend on $a_i^{[K]}$:

$$\frac{\partial \lambda}{\partial a_i^{[K]}} = \sum_{n=1}^{m_{K+1}} \frac{\partial \lambda}{\partial a_n^{[K+1]}} \frac{\partial a_n^{[K+1]}}{\partial a_i^{[K]}} = \sum_{n=1}^{m_{K+1}} \frac{\partial \lambda}{\partial a_n^{[K+1]}} \frac{\partial a_n^{[K+1]}}{\partial z_n^{[K+1]}} \frac{\partial z_n^{[K+1]}}{\partial a_i^{[K]}}. \tag{2.21}$$

Similar to equation 2.17, we can use equation 2.8 to simplify the last derivative and identify the local gradients $\delta_n^{[K+1]}$:

$$\frac{\partial \lambda}{\partial a_i^{[K]}} = \sum_{n=1}^{m_{K+1}} \delta_n^{[K+1]} w_{ni}^{[K+1]}. \tag{2.22}$$

Plugging equations 2.19 and 2.20 or 2.22 back into equation 2.18 yields the final form of the local gradient:

$$\delta_i^{[K]} = \phi'(z_i^{[K]}) \cdot \begin{cases} \frac{\partial \lambda}{\partial y_i} & \text{if layer } K \text{ is the output layer} \\ \sum_{n=1}^{m_{K+1}} \delta_n^{[K+1]} w_{ni}^{[K+1]} & \text{otherwise.} \end{cases} \tag{2.23}$$

This equation can be applied recursively to calculate the local gradients of all layers from the output layer to the input layer. The weight update $\Delta w_{ij}^{[K]}$ can then be calculated using equation 2.16 and 2.23:

$$\Delta w_{ij}^{[K]} = \eta \cdot \delta_i^{[K]} a_j^{[K-1]}. \tag{2.24}$$

When introducing the backpropagation algorithm, I summarized it as a method of *propagating the error back through the network*. The recursive usage of equation 2.23 is exactly that: We start at the output layer and calculate the local gradients of all nodes in the output layer using equation 2.23. Then we use these local gradients to calculate the local gradients of the previous layer using equation 2.23 again and so on until we reach the input layer. Note that we need to store the activations of each layer in memory during the *forward-pass* to calculate the updates during the *backward-pass* as mentioned in the beginning of this section.

The derivations for the bias updates are analogous to the weight updates and are therefore omitted here. The only difference is that the last term in equation 2.17 is replaced by 1 as the bias is not connected to any previous layer. The algorithm for a whole training step (i.e. one forward-pass and one backward-pass) is summarized in algorithm 2.

2.2.7 Summary

In this chapter, the foundations of neural networks were outlined. Section 2.2.2 started by defining the building blocks of neural networks: The *nodes*. Section 2.2.4 combined *Layers* of these nodes into a *fully connected neural network*. The specific *architecture* of the network, that is the number of layers, the number of nodes per layer and the *activation functions*, depends on the problem that is being solved. Later, we will see that in our case of Deep-Q-Learning, the size of the *input vector* is determined by the number of cells that are visible to the agent and the size of the *output vector* is determined by the number of possible actions that the agent can take.

Algorithm 2 One training step of a neural network: Forward- and Backpropagation

Require:
$\boldsymbol{x}$: Input vector
$\hat{\boldsymbol{y}}$: Target output vector
$\boldsymbol{w}$: Weight array containing all weight matrices $\boldsymbol{w}^{[K]}$
$\boldsymbol{b}$: Bias array containing all bias vectors $\boldsymbol{b}^{[K]}$
λ: Loss function
ϕ: Activation function
η: Learning rate

Output: Updates the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ of the network in place.

$\boldsymbol{a}^{[0]} \leftarrow \boldsymbol{x}$ ▷ Initialize activations
for $K = 1, \ldots, L$ **do** ▷ Forward-pass
  $\boldsymbol{z}^{[K]} \leftarrow \boldsymbol{w}^{[K]}\boldsymbol{a}^{[K-1]} + \boldsymbol{b}^{[K]}$ ▷ Calculate pre-activations
  $\boldsymbol{a}^{[K]} \leftarrow \phi(\boldsymbol{z}^{[K]})$ ▷ Calculate activations
$\delta^{[L]} \leftarrow \phi'(\boldsymbol{z}^{[L]}) \cdot \frac{\partial \lambda(\boldsymbol{a}^{[L]}, \hat{\boldsymbol{y}})}{\partial \boldsymbol{a}^{[L]}}$ ▷ Initialize local gradients
for $K = L, \ldots, 1$ **do** ▷ Backward-pass
  $\Delta \boldsymbol{w}^{[K]} \leftarrow \eta \cdot \delta^{[K]}\boldsymbol{a}^{[K-1]}$ ▷ Calculate weight updates
  $\Delta \boldsymbol{b}^{[K]} \leftarrow \eta \cdot \delta^{[K]}$ ▷ Calculate bias updates
  $\delta^{[K-1]} \leftarrow \phi'(\boldsymbol{z}^{[K-1]}) \cdot \boldsymbol{w}^{[K]T}\delta^{[K]}$ ▷ Calculate local gradients
$\boldsymbol{w} \leftarrow \boldsymbol{w} - \Delta \boldsymbol{w}$ ▷ Update weights
$\boldsymbol{b} \leftarrow \boldsymbol{b} - \Delta \boldsymbol{b}$ ▷ Update biases

On this basis, the *forward-propagation* step could be analyzed in detail and the significance of the activation function was explained. Next, section 2.2.5 introduced the *loss function* as a measure of the network's error and the *gradient descent* algorithm was outlined as a method to minimize this error. Finally, the *backpropagation* algorithm treated in section 2.2.6 provides a way to efficiently compute the gradients needed for gradient descent and enables the *training* of large neural networks. The whole process of forward-propagation, followed by backpropagation and network parameter updates is summarized in pseudocode in algorithm 2.

Chapter 3

Deep Q-Learning

3.1 Reinforcement Learning

Out of the three main branches of machine learning, *supervised learning*, *unsupervised learning* and *reinforcement learning*, reinforcement learning is the most similar to the way humans learn.
While supervised learning is based on the idea of learning from examples and unsupervised learning is concerned with finding patterns in data [36], reinforcement learning is based on the idea of learning from experience [37, chapter 1.1]. In reinforcement learning, the learning agent is not told what action to take, but instead has to learn which actions lead to a desired outcome by trial and error.
This framework has led to many successes over the last decades, including computer programs that beat the world's best players in board games like chess, Go, backgammon [22, 23, 38] or even in video games like Atari games [39] and Dota 2 [40], as well as teaching robots how to walk and handle objects [41, 42]. TODO: Add OpenAI Q^* here?

3.1.1 Overview

In a reinforcement learning scenario, the *agent* observes some information about the *environment*, called the *state* of the environment. It then uses its *policy* to map the state to an *action* that it takes in the environment. As a reaction, the environment returns a *reward* to the agent and transitions to a new state [43].
During the learning process, the agent tries to find a policy that maximizes the total reward that it receives from the environment. Depending on the type of problem, the agent can either try to maximize the reward in the short term or in the long term.

The following sections will refine these concepts and introduce the formalism of reinforcement learning.

3.1.2 Notation

Before diving into the details of reinforcement learning, I will introduce the notation that will be used throughout the following sections. It is mostly identical to the notation used in [37]. Some of these definitions might seem a bit abstract at first, but they will become clearer in the following sections, where they are introduced and explained one by one.

- s_t: The state of the environment at time step t.
- a_t: The action taken by the agent at time step t, based on the state s_t.

- r_t: The reward returned by the environment at time step t, based on the state s_t, action a_t and the new state s_{t+1}.
- π: The policy of the agent, mapping states to actions.
- $\pi(a|s)$: The probability of the agent taking action a in state s.
- $p(s_{t+1}|s_t, a_t)$: The probability of the environment transitioning to state s_{t+1} after the agent takes action a_t in state s_t.
- τ: A trajectory, i.e. a sequence of states, actions and rewards $(s_0, a_0, r_0, s_1, a_1, r_1, \dots)$.
- G_t: The return at time step t, i.e. the total discounted reward from time step t onwards.
- γ: The discount factor, determining how much future rewards are worth compared to immediate rewards.
- $v_\pi(s)$: The value of state s under policy π, i.e. the expected return when starting in state s and following policy π.
- $q_\pi(s, a)$: The value of taking action a in state s under policy π, i.e. the expected return when starting in state s, taking action a and then following policy π.
- π_*, v_*, q_*: The optimal policy, value function and action-value function respectively.

3.1.3 Important Concepts

Markov Decision Processes

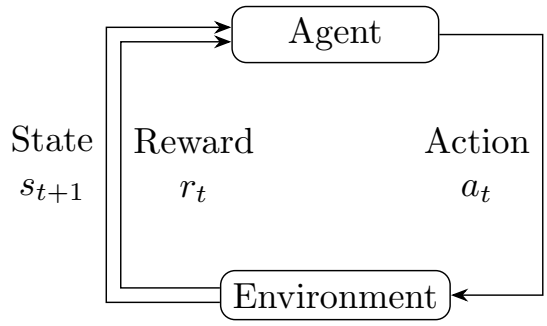


Figure 3.1: The agent-environment interface in a Markov decision process as defined in [37, chapter 3.1]. The agent observes the state s_t of the environment and takes an action a_t. The environment transitions to a new state s_{t+1} and returns a reward r_t to the agent.

Markov decision processes (MDPs) are a mathematical framework for modeling decision-making in situations where outcomes are partly random and only partly under the control of the agent [37, chapter 3]. In an MDP, the agent's actions influence not only the immediate reward, but also the next state of the environment and therefore all future rewards. They are a powerful abstraction that can be used to model a wide range of problems, from simple board games to complex real-world scenarios.
MDPs can be seen as a generalization of discrete-time *Markov chains*. Markov chains are a special case of *stochastic processes*, where the next state of the system is determined only by the current state and not by any previous states. This property is called the *Markov property* and can be expressed mathematically as [44]:

$$p(s_{t+1}|s_1, \dots, s_t) = p(s_{t+1}|s_t) = p_{ss'}. \tag{3.1}$$

The probability P of transitioning to state s_{t+1} from state s_t is independent of the previous states $s_1, \dots, s_{t-1}$ and only depends on the current state s_t. It is intuitive that the state

trajectories in a reinforcement learning scenario should always satisfy this property, as it allows the agent to make decisions based on the current state without having to consider the whole history of states that led to the current state.

MDPs slightly modify the definition of Markov chains by introducing the notion of actions and rewards. As outlined in the previous section, the agent interacts with the environment by taking actions a_i based on the current state of the environment s_i and receives rewards r_i from the environment in return. This *agent-environment interface* is illustrated in figure 3.1. A *trajectory* τ in an MDP is a sequence of states, actions and rewards [37, p. 48]:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, \dots). \tag{3.2}$$

The transition probability now depends on the action a_t that the agent takes in state s_t:

$$\begin{aligned} p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} &\to [0, 1] \\ s_{t+1}, r_t, s_t, a_t &\mapsto p(s_{t+1}, r_t | s_t, a_t), \end{aligned} \tag{3.3}$$

where $\mathcal{S}$ denotes the set of all possible states and $\mathcal{A}$ denotes the set of all possible actions. This probability function completely determines the dynamics of the reinforcement learning environment. Environments can be entirely deterministic or entirely stochastic, or anything in between. The four-argument transition probability, if known, can of course be used to calculate other properties of the environment, such as the three-argument state-transition probability [37, p. 49]:

$$\begin{aligned} p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} &\to [0, 1] \\ s_{t+1}, s_t, a_t &\mapsto p(s_{t+1} | s_t, a_t) = \sum_{r_t \in \mathcal{R}} p(s_{t+1}, r_t | s_t, a_t). \end{aligned} \tag{3.4}$$

where $\mathcal{R}$ denotes the set of all possible rewards.
Furthermore, knowing the four-argument transition probability of an environment allows us to create a simulation of the environment, which can be used to test reinforcement learning algorithms. This is a very useful property, as it allows us to test reinforcement learning algorithms in a controlled environment before deploying them in the real world.

Reward

After each time step t, the agent receives a reward r_t from the environment. The reward is a scalar value that indicates how good or bad the action a_t that the agent took in state s_t was [37, p. 53]. Rewards are the "trainer's" way of telling the agent what it should achieve. In order to use the full potential of reinforcement learning, rewards should be chosen carefully. The reward signal should not tell the agent *how* to achieve the goal, but only *what* the goal is. The agent should then be able to figure out the best way to achieve the goal by itself [37, ch. 3.2]. For example, if the goal is to teach a robot to walk, the agent should receive a reward for moving forward while maintaining a certain balance, but not for moving its legs in a certain way.
In order to be able to learn from these rewards, not just the immediate reward r_t should be taken into account, but also the rewards that the agent will receive in the future. This is done by introducing the notion of *return* The return G_t is a measure of the cumulative reward that the agent will receive from time step t onwards [37, ch. 3.3]. The simplest form of return is just the sum of all future rewards:

$$G_t = r_t + r_{t+1} + r_{t+2} + \dots + r_T = \sum_{i=t}^{T} r_i, \tag{3.5}$$

where T is the final time step in the current *episode*. Episodic tasks are task where a *terminal state* s_T can be reached, after which the episode ends. Examples are board games like chess or Go, where the game ends after one player wins. In contrast, for *continuing tasks*, as is the case for the environment that we will consider in this thesis, there is no terminal state and $T = \infty$. In this case, it's useful to introduce a *discounted* return [37, ch. 3.3]:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=t}^{\infty} \gamma^{i-t} r_i. \tag{3.6}$$

The *discount factor* γ determines how much immediate rewards should be valued compared to future rewards. A discount factor of $\gamma = 0$ means that only the immediate reward is taken into account, while a discount factor of $\gamma = 1$ would value all future rewards equally.
The discounted reward formula can be rewritten recursively as [37, p. 55]:

$$\begin{aligned} G_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\ &= r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \dots) \\ &= r_t + \gamma G_{t+1}. \end{aligned} \tag{3.7}$$

This recursive formulation is very important, as it allows us to express the return in terms of the return at the next time step. This will be useful in the next section, where we want to calculate the expected return of a state-action pair.

Policies

Reinforcement learning agents are characterized by their *policy* π. The policy maps states s to actions $a = \pi(s)$ and therefore determines the behavior of the agent [37, ch. 3.5]. The policy can be either *deterministic* or *stochastic*. A deterministic policy maps each state to exactly one action, while a stochastic policy maps each state to a probability distribution over actions. If a policy deterministically maps states to what it believes to be the best action, it is called a *greedy* policy [37, p. 64]. If an optimal policy has been found, it can be greedy, as it will always choose the best action. However, during the learning process, stochastic policies should be preferred, as they allow the agent to explore the environment and find better policies. In order to learn, an "inexperienced" agent needs to try actions that it assumes to be suboptimal to find out whether they are really suboptimal. This is called the *exploration-exploitation dilemma* [37, p. 3].
A popular solution which will also be used in this thesis is the *ϵ-greedy* policy [37, p. 100]. ϵ-greedy policies are greedy with probability $1 - \epsilon$ and choose a random action with probability ϵ. During training, ϵ is slowly decreased over time, so that the agent will explore the environment more at the beginning and exploit its knowledge later on. During evaluation, ϵ is set to zero, so that the agent will always choose the best action.

Value Functions

To actually build a policy that maximizes the expected return, the agent needs to know how good each state or state-action pair is. In order to do this, most reinforcement learning algorithms use *(state) value functions* [37, ch. 3.5]. Value functions estimate the expected return of a state s when following policy π [37, p. 58]:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] = \mathbb{E}_\pi \left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i \middle| s_t = s \right]. \tag{3.8}$$

In the same manner, we can define the *action-value function* [37, p. 58]:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{i=t}^{\infty} \gamma^{i-t} r_i \middle| s_t = s, a_t = a\right]. \tag{3.9}$$

Optimality and Bellman Equations

Now the recursion formula from equation 3.7 comes in handy to express the value functions in terms of the value functions of the next state:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] \\ &= \mathbb{E}_\pi[r_t + \gamma G_{t+1}|s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_{r_t} p(s', r_t|s, a) \left[r_t + \gamma \mathbb{E}_\pi[G_{t+1}|s_{t+1} = s']\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_{r_t} p(s', r_t|s, a) \left[r_t + \gamma v_\pi(s')\right], \end{aligned} \tag{3.10}$$

where we reintroduced the four-argument transition probability. The summations are over all possible actions a and all possible next states s' and rewards r_t. Equation 3.10 is called the *Bellman equation* for v_π [37, p. 59].
The Bellman equation for q_π is analogous [43]:

$$q_\pi(s,a) = \sum_{s'} \sum_{r_t} p(s', r_t|s_t, a_t) \left[r_t + \gamma \sum_{a_{t+1}} \pi(a_{t+1}|s_{t+1} = s') q_\pi(s', a_{t+1})\right]. \tag{3.11}$$

The Bellman equations heavily simplify the calculation of the value functions, as they allow us to express the value of a state or state-action pair in terms of the values of the next state or state-action pairs. Imagine a game of chess: Even for very fast computers, evaluating every possible trajectory of moves until one player wins is not feasible. However, the Bellman equations allow us to calculate the value of a state or state-action pair without having to consider all possible trajectories. We just have to keep track of the values of individual states or state-action pairs and update them according to the Bellman equations.

We can now use these concepts to define *optimal* policies π_*. A policy is considered "better" than another policy, if it achieves a higher expected return in every state [37, p. 62]:

$$\pi \geq \pi' \Leftrightarrow v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}. \tag{3.12}$$

An optimal policy π_* is a policy that is better than or equal to all other policies [37, p. 62]. Associating the value functions v_* and q_* with the optimal policy π_*, we can define the optimal (action-)value functions as [37, ch. 3.6]:

$$v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S}. \tag{3.13}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}. \tag{3.14}$$

The optimal value functions satisfy the *Bellman optimality equations* [37, ch. 3.6]. When an agent follows an optimal policy, the sum over all possible actions can be replaced by using the action that maximizes the expected return:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}_\pi\left[r_t + \gamma \max_{a'} q_*(s_{t+1}, a') \middle| s_t = s, a_t = a\right] \\ &= \sum_{s'} \sum_{r_t} p(s', r_t|s, a) \left[r_t + \gamma \max_{a'} q_*(s', a')\right]. \end{aligned} \tag{3.15}$$

Solving equation 3.15 yields the optimal action-value function q_*, which can then be used to derive the optimal policy π_*. For most reinforcement learning problems however, it is not feasible to solve the Bellman optimality equations analytically, even if the transition probabilities are known. Therefore, most reinforcement learning algorithms use iterative methods to approximate the optimal value functions [37, ch. 4].

3.1.4 Summary

Before we dive into the details of the Deep Q-Learning algorithm, let's summarize the most important concepts of reinforcement learning that were introduced in this section.

Reinforcement learning is a framework for *learning from experience*. An *agent* interacts with an *environment* by taking *actions* based on the current *state* of the environment. The environment returns a *reward* which the agent uses to keep track of the *return* that it will receive in the future. This in turn allows the agent to learn a *policy* that maximizes the expected the return by learning the *value* of each state or state-action pair. *Optimal* policies can be found by solving the *Bellman optimality equations*.

3.1.5 Algorithms

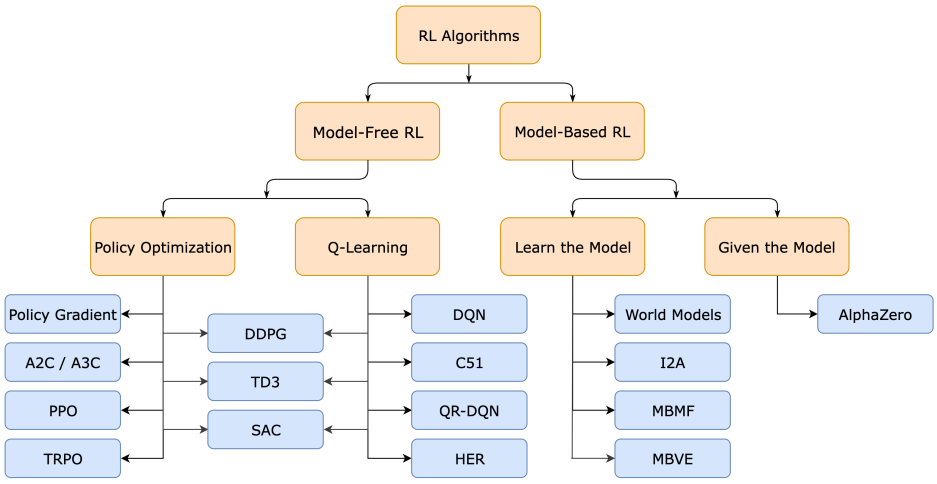


Figure 3.2: Overview of the different classes of reinforcement learning algorithms [45]. The papers that introduced the algorithms are also available at [45].

Figure 3.2 provides a basic overview of the different classes of reinforcement learning algorithms. We will not explain the details of each algorithm here, but instead focus on the general distinctions between the different classes of algorithms.

The first distinction is between *model-based* and *model-free* algorithms. Model-based algorithms have an internal model of the environment, which is either learned or provided by the user [45]. This allows them to plan ahead and simulate the environments dynamics to find the best action to take. Model-free algorithms on the other hand only implicitly learn the dynamics of the environment by interacting with it and learning value functions, as explained in the previous sections. Model-free algorithms are more flexible, as they can be applied to any environment, but they are also less sample-efficient, as they have to learn the dynamics of the environment by trial and error. Model-based algorithms on the other hand can be more sample-efficient, but they are harder to implement and fine-tune to specific problems [45]. Probably the most famous model-based reinforcement learning algorithm is AlphaZero [23], a program that taught itself to play chess, Go and Shogi at superhuman levels.

Model-free algorithms can be distinguished further by *what* is learned. In the last section, we introduced the concept of value functions. Algorithms that learn value functions are called *value-based* algorithms. If they learn the action-value function q_π, they are called *Q-learning* algorithms. Most of these algorithms use *off-policy* learning, which means that they learn the value of the optimal policy π_* while following a different policy π [45]. This allows them to learn from data that was collected at a previous stage of the training process, making them very sample-efficient. Probably the most famous Q-learning algorithm is Deep Q-Learning, which we will discuss in detail in the next section.
The other class of model-free algorithms are *policy-based* algorithms. These algorithms directly learn the optimal policy π_*, instead of learning value functions. Most of these algorithms use *on-policy* learning, restricting them to only learn from data that was collected while following the current policy π [45]. Because of that, they are less sample-efficient than Q-learning algorithms, but they also tend be more stable, as they directly learn the policy, instead of indirectly learning the policy by learning value functions [45].

As we can see in figure 3.2, there are also algorithms that use ideas from different classes of algorithms. In general, it is hard to draw clear distinctions between the different classes of algorithms, as their modular nature allows them to be combined in many different ways [45].

Choosing the best algorithm for a specific problem is one of the most important steps in applying reinforcement learning to a real-world problem. In this thesis, we will use Deep-Q-Learning, as its use of deep neural networks allows it to scale and generalize well. It is also a model free, off-policy algorithm, which makes it relatively easy to implement and modify. The sampling efficiency of off-policy learning is also a big advantage, as it allows training the network quickly on normal hardware.

3.2 Q-Learning

Before diving into the details of the Deep Q-Learning algorithm, we will introduce the general concepts of Q-Learning, building on the concepts of reinforcement learning that were introduced in the previous section.

Q-Learning was introduced by Watkins in 1989 [46]. It is a model-free, off-policy algorithm that learns an action-value function $Q(s, a)$ which is guaranteed to converge to the optimal action-value function q_* [47] [37, ch. 6.5]
The algorithm works by storing a table of action-values $Q(s, a)$ for each state-action pair (s, a) that the agent has encountered. Such a *Q-table* can be seen in figure 3.3. In the beginning, the Q-table is initialized randomly. Then, the agent interacts with the environment by taking actions and receiving rewards. The policy used for this interaction is derived from the Q-table for example by choosing the action with the highest action-value in an ϵ-greedy manner. For each sample (state s, action a, reward r, next state s') that the agent encounters, the Q-table is updated according to the following formula [37, ch. 6.5]:

$$Q(s, a) \leftarrow Q(s, a) + \eta \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right], \tag{3.16}$$

where η is the learning rate and γ is the discount factor. Comparing with equation 3.15, we can see that this update rule is an iterative approximation of the Bellman optimality equation for q_*. The new approximation of the optimal action-value is a weighted sum of the old approximation, and the new information that was gained from the sample, i.e. the immediate reward r and the discounted value of the next action. The learning rate η determines how

much the new information is weighted compared to the old approximation. The discount factor again γ determines how much future rewards are valued compared to immediate rewards.

We see that while time complexity stays constant when using lookup tables, the space

	a_1	a_2	...	a_n
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	...	$Q(s_1, a_n)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	...	$Q(s_2, a_n)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
s_m	$Q(s_m, a_1)$	$Q(s_m, a_2)$	...	$Q(s_m, a_n)$

Figure 3.3: A Q-table for an environment with m states and n actions.

complexity grows linearly with the number of states and actions. This makes Q-learning impractical for environments with large state spaces. Also, environments that have continuous state or action spaces have to be discretized, which can lead to a loss of information and therefore suboptimal policies. Even for seemingly simple environments such as a grid, where each grid cell can either be occupied or unoccupied, the number of possible states grows exponentially with the size of the grid. Therefore, we need a more efficient way to represent the action-values. This is where Deep Q-Learning comes in.

3.3 Deep Q-Learning

To solve the problem of large state spaces, we look back at chapter 2, where we learned, that deep neural networks are universal function approximators. This makes them a good candidate for approximating the action-values $Q(s, a)$. When we replace the Q-table with a neural network, we supply the current state s as input and get the action-values $Q(s, a)$ as output. For our grid environment example from the last section, this would mean that we supply the current grid as input. This eliminates the exponential space complexity of the Q-table, as the input vector of the network now grows linearly with the number of cells. Furthermore, the discretization problem is also solved, as the network can now take continuous inputs. Ideally, the network would store the same information as the table, but in a more compact way by learning the patterns in the data.

This approach is called *Deep Q-Learning* (DQN) and was introduced by Mnih et al. in 2013 [39]. DQN also introduces a few other concepts apart from the Q-network, which we will discuss in the following sections.

3.3.1 Q-Network

The Q-network, sometimes also called *policy network*, was already discussed in the previous section. It is a neural network that replaces the Q-table and approximates the action-values $Q(s, a)$. During evaluation, the environments state is supplied as input to the network. After a forward pass through the network, as explained in section 2.2.4, the output vector of the network corresponds to the approximate action-values $Q(s, a)$ for each action a. When the network has converged to a good approximation of the optimal action-values q_*, the action with the highest action-value can be chosen as the action that the agent takes in the current state. During training, this action is chosen in an ϵ-greedy manner, as explained in section 3.1.3 to maintain exploration [48].

3.3.2 Target Network

The Q-network also changes the way that the parameters are updated. We can no longer update individual Q-values, as this information is now encoded in the weights and biases of the network. Instead, we have to update the parameters of the network. This is done by using *gradient descent* and the *backpropagation* algorithm, as explained in sections 2.2.5 and 2.2.6.

The loss function that is used for the gradient descent requires a target output vector $\hat{y}$ for each sample. In the case of Q-learning, this would be the optimal action-values $q_*(a)$. As this is what we're optimizing for, we can't use it as the target output vector. Instead, we could use an earlier approximation of the action-values $Q(s, a)$. However, this can lead to oscillations or divergence of the network parameters, as the target network would use the same parameters as the Q-network [48]. To solve this problem, DQN introduces a second network, called the *target network*. The target network is a clone of the Q-network with frozen parameters. Using this separate network to calculate the action-value targets makes the learning process more stable [48].
In the original algorithm, the target network parameters are updated periodically by copying the parameters from the Q-network. This is called *hard target network update* [48]. In 2016, Lillicrap et al. [49] introduced *soft target network updates*, where the target network parameters are updated by slowly blending the parameters of the Q-network into the target network parameters:

$$\theta_{\text{target}} \leftarrow \tau\theta_{\text{Q}} + (1 - \tau)\theta_{\text{target}}, \tag{3.17}$$

where $\tau \ll 1$ is a small number between 0 and 1, θ_{target} are the target network parameters and θ_{Q} are the Q-network parameters.
This makes the learning process even more stable, as the target network parameters are constrained to slow updates [49]. In this thesis, we will also use soft target network updates.

3.3.3 Experience Replay

In the beginning of this section, DQN was introduced as an off-policy algorithm. Mnih et al. introduced a technique called *experience replay* [48], which massively improves the algorithms' quality. The technique is inspired by the way that humans learn [48] and works by storing tuples of states, actions, rewards and next states (s_t, a_t, r_t, s_{t+1}), called *transitions* in a *replay buffer*. During training, the network is updated by sampling a batch of transitions from the replay buffer and performing a gradient descent step on the loss function. This not only allows the network to learn from the same experience multiple times, increasing sample efficiency, but also breaks up the correlation between consecutive samples, which would otherwise decrease learning efficiency and stability [48]. In order to prevent the replay buffer from growing infinitely, the oldest samples are discarded when the buffer is full. This total buffer size and the sampling batch size are hyperparameters that have to be tuned for each problem.

One problem that exists in both on-policy learning and off-policy learning with experience replay is that rare transitions are sampled very infrequently, which can slow down learning. To solve this problem, Schaul et al. introduced *prioritized experience replay* in 2015 [50]. In prioritized experience replay, each transition is assigned a priority based on the magnitude of the loss function [50]. That way, transitions that are "surprising" to the agent are sampled more frequently, which speeds up learning [50].

3.3.4 Summary

In this section, we learned that Deep Q-Learning uses a neural network to map states to action-values. This allows it to scale to large and continuous state spaces. Learning is

performed *offline*, by saving encountered transitions to a replay buffer and sampling batches of transitions from the buffer to update the network parameters. A separate target network is used to calculate the action-value targets, which makes the learning process more stable. The target network parameters are updated by slowly blending the parameters of the Q-network into the target network parameters. The complete algorithm is shown in algorithm 3.

Algorithm 3 Deep Q-Learning

Initialize replay buffer $\mathcal{D}$ to capacity N
Initialize Q-network Q with parameters θ
Initialize target network $\hat{Q}$ with parameters $\theta^- = \theta$
(Initialize environment)
Initialize state s_1
Initialize ϵ to ϵ_0
for steps $t = 1, \ldots, T$ **do**
  ▷ Generate new transition
  Perform forward pass through Q to get action-values $Q(s_t, a')$
  With probability ϵ select a random action a_t, otherwise select $a_t = \text{argmax}_a\, Q(s_t, a)$
  Execute action a_t in environment and observe reward r_t and next state s_{t+1}
  Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
  ▷ Update Q-network parameters
  Sample random minibatch of transitions $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s'})$ from $\mathcal{D}$
  Set $\boldsymbol{y} = \boldsymbol{r} + \gamma \max_{a'} \hat{Q}(\boldsymbol{s'}, a')$
  Calculate mean loss $\mathcal{L} = \text{mean}(\mathcal{L}(\boldsymbol{y}, Q(\boldsymbol{s}, \boldsymbol{a})))$
  Perform gradient descent step on $\mathcal{L}$ using backpropagation and update θ
  ▷ Update target network parameters
  Update target network parameters $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$
  ▷ Update exploration rate
  Decrease ϵ

Chapter 4

Physical Model

Bibliography

[1] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Fourth edition. Pearson series in artificial intelligence. Hoboken: Pearson, 2021. ISBN: 978-0-13-461099-3.

[2] *Artificial intelligence*. In: *Wikipedia*. Page Version ID: 1185402635. Nov. 16, 2023. URL: https://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=1185402635 (visited on 11/16/2023).

[3] Elaine Woo. *John McCarthy dies at 84; the father of artificial intelligence*. Los Angeles Times. Section: Obituaries. Mar. 20, 2014. URL: https://www.latimes.com/local/obituaries/la-me-john-mccarthy-20111027-story.html (visited on 11/16/2023).

[4] Scott L. Andresen. "John McCarthy: Father of AI". In: *IEEE Intelligent Systems* 17.5 (Sept. 1, 2002). Publisher: IEEE Computer Society, pp. 84–85. ISSN: 1541-1672. DOI: 10.1109/MIS.2002.1039837. URL: https://www.computer.org/csdl/magazine/ex/2002/05/x5084/13rRUxE04ph (visited on 11/16/2023).

[5] *What is AI? / Basic Questions*. URL: http://jmc.stanford.edu/artificial-intelligence/what-is-ai/index.html (visited on 11/16/2023).

[6] John McCarthy et al. *A PROPOSAL FOR THE DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE*. URL: http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html (visited on 11/16/2023).

[7] *Google Books Ngram Viewer*. URL: https://books.google.com/ngrams/graph?content=AI&year_start=1800&year_end=2019&corpus=en-2019&smoothing=3 (visited on 11/16/2023).

[8] SITNFlash. *The History of Artificial Intelligence*. Science in the News. Aug. 28, 2017. URL: https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/ (visited on 11/16/2023).

[9] *AI Spring? Four Takeaways from Major Releases in Foundation Models*. Stanford HAI. URL: https://hai.stanford.edu/news/ai-spring-four-takeaways-major-releases-foundation-models (visited on 11/16/2023).

[10] Geoffrey Hinton et al. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012). Conference Name: IEEE Signal Processing Magazine, pp. 82–97. ISSN: 1558-0792. DOI: 10.1109/MSP.2012.2205597. URL: https://ieeexplore.ieee.org/document/6296526 (visited on 11/16/2023).

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 25. Curran Associates, Inc., 2012. URL: https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (visited on 11/16/2023).

[12] *A decade in deep learning, and what's next*. Google. Nov. 18, 2021. URL: https://blog.google/technology/ai/decade-deep-learning-and-whats-next/ (visited on 11/16/2023).

[13] Bryan House. *2012: A Breakthrough Year for Deep Learning*. Deep Sparse. July 17, 2019. URL: https://medium.com/neuralmagic/2012-a-breakthrough-year-for-deep-learning-2a31a6796e73 (visited on 11/16/2023).

[14] *Introducing ChatGPT*. URL: https://openai.com/blog/chatgpt (visited on 11/16/2023).

[15] *ChatGPT*. URL: https://chat.openai.com (visited on 11/16/2023).

[16] *What Is Moore's Law and Is It Still True?* Investopedia. URL: https://www.investopedia.com/terms/m/mooreslaw.asp (visited on 11/16/2023).

[17] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986). Number: 6088 Publisher: Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 11/16/2023).

[18] *9 ways we use AI in our products*. Google. Jan. 19, 2023. URL: https://blog.google/technology/ai/9-ways-we-use-ai-in-our-products/ (visited on 11/16/2023).

[19] Robin Burke, Alexander Felfernig, and Mehmet H. Göker. "Recommender Systems: An Overview". In: *AI Magazine* 32.3 (June 5, 2011). Number: 3, pp. 13–18. ISSN: 2371-9621. DOI: 10.1609/aimag.v32i3.2361. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2361 (visited on 11/16/2023).

[20] *ElevenLabs: AI Voice Generator & Text to Speech*. URL: https://elevenlabs.io/ (visited on 11/16/2023).

[21] *Midjourney*. Midjourney. URL: https://www.midjourney.com/home?callbackUrl=%2Fexplore (visited on 11/16/2023).

[22] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (Jan. 2016). Number: 7587 Publisher: Nature Publishing Group, pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://www.nature.com/articles/nature16961 (visited on 11/16/2023).

[23] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. Dec. 5, 2017. DOI: 10.48550/arXiv.1712.01815. arXiv: 1712.01815[cs]. URL: http://arxiv.org/abs/1712.01815 (visited on 11/16/2023).

[24] Kelsey Piper. *AI triumphs against the world's top pro team in strategy game Dota 2*. Vox. Apr. 13, 2019. URL: https://www.vox.com/2019/4/13/18309418/open-ai-dota-triumph-og (visited on 11/16/2023).

[25] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-94462-3 978-3-319-94463-0. DOI: 10.1007/978-3-319-94463-0. URL: http://link.springer.com/10.1007/978-3-319-94463-0 (visited on 11/18/2023).

[26] *Explained: Neural networks*. MIT News — Massachusetts Institute of Technology. Apr. 14, 2017. URL: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414 (visited on 11/16/2023).

[27] *What are Neural Networks? — IBM*. URL: https://www.ibm.com/topics/neural-networks (visited on 11/16/2023).

[28] Alexandre Gonfalonieri. *Understand Neural Networks & Model Generalization*. Medium. Jan. 29, 2020. URL: https://towardsdatascience.com/understand-neural-networks-model-generalization-7baddf1c48ca (visited on 11/18/2023).

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Cambridge, Massachusetts: The MIT Press, Nov. 18, 2016. 800 pp. ISBN: 978-0-262-03561-3.

[30] Simon Haykin. *Neural Networks: A Comprehensive Foundation: A Comprehensive Foundation: United States Edition*. Subsequent Edition. Upper Saddle River, NJ: Pearson, July 1, 1998. 842 pp. ISBN: 978-0-13-273350-2.

[31] Tyler Elliot Bettilyon. *Deep Neural Networks As Computational Graphs*. Teb's Lab. May 5, 2020. URL: https://medium.com/tebs-lab/deep-neural-networks-as-computational-graphs-867fcaa56c9 (visited on 11/23/2023).

[32] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. June 28, 2022. DOI: 10.48550/arXiv.2109.14545. arXiv: 2109.14545[cs]. URL: http://arxiv.org/abs/2109.14545 (visited on 11/23/2023).

[33] Ryan P Adams. "Computing Gradients with Backpropagation". In: ().

[34] Louis B. Rall, ed. *Automatic Differentiation: Techniques and Applications*. Red. by G. Goos et al. Vol. 120. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1981. ISBN: 978-3-540-10861-0 978-3-540-38776-3. DOI: 10.1007/3-540-10861-0. URL: http://link.springer.com/10.1007/3-540-10861-0 (visited on 11/30/2023).

[35] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. Google-Books-ID: qMLUIsgCwvUC. SIAM, Nov. 6, 2008. 448 pp. ISBN: 978-0-89871-659-7.

[36] *What is Machine Learning? — IBM*. URL: https://www.ibm.com/topics/machine-learning (visited on 12/03/2023).

[37] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*.

[38] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: https://dl.acm.org/doi/10.1145/203330.203343 (visited on 12/08/2023).

[39] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 19, 2013. DOI: 10.48550/arXiv.1312.5602. arXiv: 1312.5602[cs]. URL: http://arxiv.org/abs/1312.5602 (visited on 12/08/2023).

[40] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. Dec. 13, 2019. DOI: 10.48550/arXiv.1912.06680. arXiv: 1912.06680[cs,stat]. URL: http://arxiv.org/abs/1912.06680 (visited on 12/08/2023).

[41] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11 (Sept. 1, 2013). Publisher: SAGE Publications Ltd STM, pp. 1238–1274. ISSN: 0278-3649. DOI: 10.1177/0278364913495721. URL: https://doi.org/10.1177/0278364913495721 (visited on 12/08/2023).

[42] OpenAI et al. *Learning Dexterous In-Hand Manipulation*. Jan. 18, 2019. DOI: 10.48550/arXiv.1808.00177. arXiv: 1808.00177[cs,stat]. URL: http://arxiv.org/abs/1808.00177 (visited on 12/08/2023).

[43] *Part 1: Key Concepts in RL — Spinning Up documentation*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html (visited on 12/08/2023).

[44] Richard Serfozo. "Markov Chains". In: *Basics of Applied Stochastic Processes*. Ed. by Richard Serfozo. Probability and Its Applications. Berlin, Heidelberg: Springer, 2009, pp. 1–98. ISBN: 978-3-540-89332-5. DOI: 10.1007/978-3-540-89332-5_1. URL: https://doi.org/10.1007/978-3-540-89332-5_1 (visited on 12/12/2023).

[45] *Part 2: Kinds of RL Algorithms — Spinning Up documentation*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below (visited on 12/17/2023).

[46] Christopher Watkins. "Learning From Delayed Rewards". In: (Jan. 1, 1989).

[47] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (May 1, 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698 (visited on 12/17/2023).

[48] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015). Number: 7540 Publisher: Nature Publishing Group, pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236 (visited on 12/17/2023).

[49] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. July 5, 2019. DOI: 10.48550/arXiv.1509.02971. arXiv: 1509.02971[cs,stat]. URL: http://arxiv.org/abs/1509.02971 (visited on 12/17/2023).

[50] Tom Schaul et al. *Prioritized Experience Replay*. Feb. 25, 2016. DOI: 10.48550/arXiv.1511.05952. arXiv: 1511.05952[cs]. URL: http://arxiv.org/abs/1511.05952 (visited on 12/19/2023).

Appendix A

Appendix

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Selbständigkeitserklärung

Ich versichere hiermit, die vorliegende Arbeit mit dem Titel

Titel der Arbeit

selbständig verfasst zu haben und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Jonas Märtens

München, den 15. Januar 2024