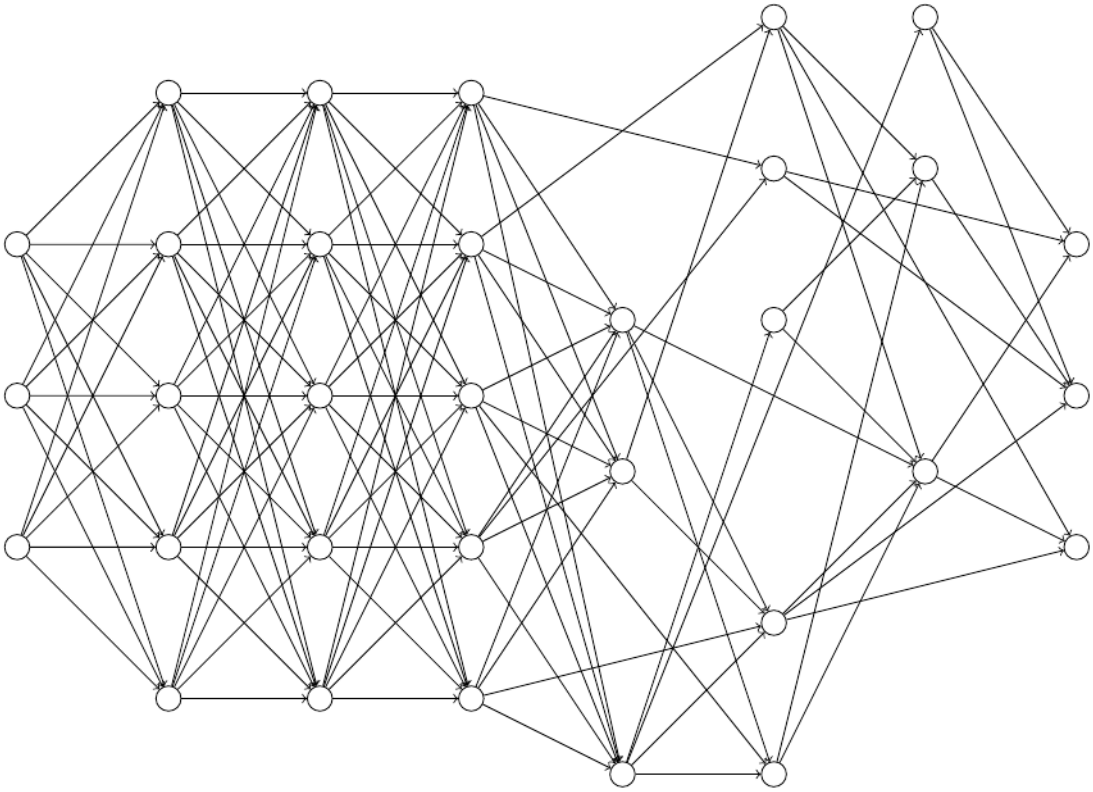
**Matura paper Alte Kantonsschule Aarau**

# Two ex-nihilo Implementations of Reinforcement Learning Algorithms

Luis Hartmann, G19D  Fabio Panduri, G19E

October 2022



Submitted to Lukas Wampfler

# Contents

# Abstract

In this paper, two Reinforcement Learning algorithms, namely the algorithm Deep Q-Learning (DQL) and the algorithm NeuroEvolution of Augmenting Topologies (NEAT), were successfully implemented. The implementations are publicly available on `https://github.com/fabiopanduri/matura`. In a further step, the two algorithms were compared. For the performance comparisons, the games Cartpole and Pong were used as benchmarks. For Cartpole, both DQL and NEAT ended up with a stable and good performance after their training. However, DQL took about four times longer than NEAT to arrive at this performance. Pong was only successfully solved by DQL and not by NEAT, though only by use of a special reward system which gives rewards based on the paddle's position relative to the ball.

# Preface

Admittedly, I would rather rewrite the entire backpropagation section than start this preface. Alas, Luis has tasked me with writing an introduction to the preface. A choice which I cannot understand (as we both know that it would have turned out way better if he wrote it), but here I am writing, reminiscing about how we decided to work on Reinforcement Learning and definitely regretting that we did not choose to program a preface-writing bot. (Or did we? Is this machine-generated text?)

Anyways, as you may guess, we both love writing code, at least one of us loves writing formulae and, depending on the topic (i.e. preface or no preface), we also both love writing in general. The central point, however, is writing code. We both started to code early on. And then we started to code again when we began at the Alte Kantonsschule Aarau and at SOI. Aside from loving to code, our curiosity concerning the field of Machine Learning is as big as our theory chapter. That is why we hope to also arouse the reader's interest in this field with our paper. In addition, we hope for this paper to be a good introduction to the topics discussed, which can be of use to anyone intrigued with Machine Learning.

"What exactly are the topics of the paper?", you might ask. Let us say we covered anything from Neural Networks and Stochastic Gradient Descent over Deep Q-Learning to NeuroEvolution of Augmenting Topologies. "How on earth did you come up with those topics?", you might proceed. I will leave this question to Luis.

When, sometimes, I am asked: "Where did the two of you get your idea to implement Machine Learning algorithms from?", I struggle to find a satisfying answer.

I feel as though the idea has always been around, for at least as long as the two of us go to school at the Alte Kantonsschule Aarau, fuelled not by the bling-bling Machine Learning is popularly associated with but by the unquenchable desire to prove to ourselves that it is, in fact, not magic. There are numerous occasions where the theme came up, some of them I remember vividly.

As early as winter of 2019/2020 I wanted to program a simple (so I thought) script to read sheet music from images and translate it into note names. While this was very much an overestimation of my abilities, it was nonetheless a learning experience and my first attempt at Machine Learning.

The first somewhat successful Machine Learning program I saw was shown to me by Fabio about a year later. We were sitting in German class, presumably discussing "Nathan der

Weise" by Lessing, when I saw his tinkering with a few dozens of lines of code and asked him what the program was good for. He told me how, the evening before, he had implemented a small neuroevolutionary algorithm playing tic-tac-toe and was now, bored with Mrs. Amsler's lesson (something I find incomprehensible), trying out some stuff with it. I was fascinated! Machine Learning, wasn't that supposed to be absurdly difficult? And Fabio had implemented it, just like that, on a Monday evening? I must admit to feeling somewhat envious of his skill.

The impression stuck with me. When it came to developing a plan suitable for the programming project, which we would have to complete before summer holidays of 2021, Machine Learning was of course the first thing to pop into our minds. I believe we should thank Mrs. Vàzquez for her disapproving of the idea (in favour of an encrypted messaging application). Had we been more stubborn about the Machine Learning project, we might never have written our matura paper about the topic, for although Fabio and I have done a fair few projects together over the last years, no two have ever been on the same topic.

But we would not let go. I recall clearly when, in a math lesson sometime during that year, Mr. Sax showed us ideas for the Projektarbeit we were going to write. The hypnotisingly elegant mathematical beauty of Mandelbrot's fractal was only to capture our interest for a few moments. Fabio had leaned over to me as we were considering the options for - what else could it have been - doing something in the field of Machine Learning. I presume we would have gone through with that, had it not been for Leana's joining our team for the Projektarbeit, consisting up until that point of only Fabio and myself. Thanks to her joining we had subsequently decided on a different subject, as we also shared an interest in Physics, and as such we brought the paper "Drei praktische Anwendungen der Schwingungslehre" [4] into being.

And so it should be that in winter of 2021/2022 we had still not got round to doing our Machine Learning project. The lesson with Mr. Senn which officially started the matura project, the grand finale of projects to be done at Alte Kantonsschule Aarau, would soon have passed and with it had arrived the time to select a topic to write the matura paper on. The choice wasn't a difficult one.

So it happens that, when one asks me the question of "How did you decide on the subject for your matura paper?", I ponder for a second, considering possible answers with nostalgic thoughts flowing through my mind, after which I have so far most often answered "Fabio and I thought it would be an interesting and relevant topic" but shall in the future be able to respond with "There's a few paragraphs about it in the paper. You best read it for yourself!"

# Acknowledgements

We like to thank, first and foremost, Mr. Wampfler for taking up the task of overseeing the project, for his support by way of recommendations and feedback throughout the project and for the work of assaying the paper and the presentation, which he does together with our second examiner Mr. Graf, to whom we, too, give our best thanks. In addition, we were again glad for Mr. Graf's LaTeX template. We are also much obliged to Rocco Panduri and Oliver Töngi for proofreading our paper and giving us last feedback and advice on content as well as English.

We could never mention every developer and computer scientist whose work has been essential for this paper, but some names are Ken Stanley, the DeepMind team, Guido Van Rossum, Bram Moolenaar et al., the pip developers, the gym community, Travis E. Oliphant et al., Donald Knuth and Leslie Lamport, the GNU Project, the Linux Foundation, the Debian Project and, of course, the diligent (and presumably well-paid) guys working at Bell Labs during the late sixties. Last but not least, we are most grateful for all the developers and maintainers of quality Free and Open-Source Software left unmentioned, without your gracious contributions to human kinship, projects like this one would not be possible.

# 1 Introduction

If not magic, then what is Machine Learning? At its core, the field is a clever application of mathematics, especially statistics, resulting in algorithms capable of solving some previously unsolved challenges.

The aim of this paper is to take a deep dive into Reinforcement Learning algorithms. For this we have chosen two concrete algorithms, namely Deep Q-Learning (DQL) and NeuroEvolution of Augmenting Topologies (NEAT).

We set out to implement both of the algorithms ourselves, *ex-nihilo*, instead of resorting to already existing implementations. This meant, on the one hand, that we had to learn and comprehend all the theories behind the algorithms and, on the other hand, that we could immediately apply this theory to our implementations.

DQL and NEAT are both algorithms capable of solving Reinforcement Learning problems. Interestingly, they work in a completely different fashion. DQL is a deep Reinforcement Learning algorithm employing the Stochastic Gradient Descent algorithm to optimise a Neural Network. NEAT, on the other hand, is an evolutionary algorithm inspired by the process of evolution as it was first described by C. Darwin in [5].

We have chosen those two algorithms in part for this reason. Their different approaches to the same type of problems make it simple yet interesting to compare their performances, which, next to the ex-nihilo implementations, has been our goal for this project. For comparison and benchmarking we have used two well-known classics. The first, Cartpole, is a classic in benchmarking for various Reinforcement Learning algorithms while the second, Pong, is the all-time classic arcade game which we have all certainly played at least once before.

Based on their performance in the original publications as well as the fact that NEAT is almost two decades older than DQL, it might be assumed that DQL will outperform NEAT in both tasks, given the same preconditions. However, a publication comparing the two algorithms came to a different conclusion: "Measurements indicated that NEAT reliably produces better AIs than RL in the studied problem classes." [1, p. iii]. (Note that "RL" refers to the DQL algorithm.)

We shall now start with a chapter explaining the theory of Machine Learning, especially the Reinforcement Learning algorithms covered in the paper.

# 2 Theory

Although we do not wish to cover the sociophilosophical aspects of this field, we hope to do our part in helping to clear up some of the haze which often comes with the terms Machine Learning and Artificial Intelligence.

## 2.1 Machine Stupidity

The topic of this paper is Machine Learning. Applications where Machine Learning shines typically involve handling large amounts of data. While other approaches to working with data tend to be more difficult and less accurate with larger datasets, Machine Learning ingeniously exploits the data to be more accurate the larger the set of data is. Typical applications include pattern recognition, classification of samples, generation of data similar to a provided dataset, among others.

The reader might associate Machine Learning with Artificial Intelligence. There is probably nothing wrong with this. In fact, even the Oxford Learner's Dictionary understands Machine Learning as "a type of artificial intelligence in which computers use huge amounts of data to learn how to do tasks rather than being programmed to do them" [12]. In spite of this association with intelligence, the algorithms we implemented are stupid. This means that many properties which would be expected from a non-stupid entity are not fulfilled by our algorithms. For one, their abilities are confined to an extremely specific application. Additionally, they are incapable of thought, of making decisions more complex than selecting an item out of a list of about three, of applying themselves to other areas, as well as of many other things indeed.

Stupid they may be, there is one thing which our algorithms are capable of: learning. By learning we do not mean studying for a physics exam. This would require our algorithms to not be stupid, a criterion which is not met. We instead mean finding a solution to a given problem within a very specific framework without being told in advance exactly how to do so. Our algorithms learn by finding the solutions by way of exploration, trial and error as well as educated guesses, through continual improvement of a strategy by help of feedback.[1]

Because machines are stupid, they cannot learn by themselves. They have to be given exact instructions before any learning can occur. This is where a solid learning approach (in our

[1] Interestingly, our algorithms have many similarities with small children. First, they are stupid. Second, they are able to learn. Third, they get better and better the more they are exposed to data. There are certainly more shared attributes to be found.

case Reinforcement Learning) is needed and where humans (in our case the authors of this paper) come in and implement said approach. Let us explain it.

## 2.2 Reinforcement Learning

The subject of the paper is Reinforcement Learning, one of the big three categories of Machine Learning, along with Supervised and Unsupervised Learning.

Reinforcement Learning is applicable to Machine Learning problems where "sequential decisions are to be made" [7, 1:13]. This includes of course the games Pong, Cartpole, as well as many (video)games and, for example, self-driving cars.

A typical Reinforcement Learning algorithm consists of an agent and an environment, the agent acting within the environment according to a policy.

**The Agent.** The agent interacts with the environment through repeatedly observing its state $s$, performing an action $a$ according to a policy $\pi(s, a)$ and receiving a reward $r$ from the environment. The process of observing a state, performing an action and gaining a reward is called a "step". The goal of the agent is to maximize rewards $r$. [3, p. 501]

**The Environment.** The environment is the structure in which the agent is acting and from which it is receiving rewards. This may be a game like Pong or Chess, but could also be a busy traffic intersection (in the case of a self-driving car algorithm). Different environments have different attributes. The environment we used for both Pong and Cartpole is

- fully observable: The agent is (in theory) able to observe the entire state of the environment.
- single agent or dual agent: One or two players.
- deterministic: Given state $s$ the next state $s'$ is uniquely defined after taking action $a$.
- continuous: There are no interruptions or breaks, as opposed to discrete. [6, slide 16]

What is important to realize is that more often than not in practical situations, it is not possible to fully know the environment's model, which is where model-free Reinforcement Learning techniques come in handy. Instead of working with a model, model-free techniques use, for example, Q-Function approximation, in the case of Q-Learning. Still, the model for the games we used is fully known - the game implementations themselves are the model.

We now discuss the three most important functions for Reinforcement Learning. Those are the policy function $\pi$, the value function $V$ and the quality function $Q$.

### 2.2.1 The Policy Function

The agent uses the policy function to calculate the probability Pr of taking an action $a$ in a given state $s$:

$$\pi(s, a) := \Pr(a' = a | s' = s). \tag{2.1}$$

In practice, it is unfeasible to calculate the exact policy $\pi$, which is therefore approximated by the function $\pi_\theta$. The function $\pi_\theta$ may be chosen arbitrarily, though it is typically of a lower dimension than $\pi$ is, making its evaluating less computationally demanding. Policy function approximation is seldom trivial, it is one of the central challenges in many Reinforcement Learning approaches. [3, p. 502]

### 2.2.2 The Value Function

The value function serves as a tool for the agent, giving it information about the value of being in a state $s$. The value function $V_\pi(s)$ is defined formally as

$$V_\pi(s) := \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_k \mid s_0 = s\right), \tag{2.2}$$

where $\pi$ is an arbitrary policy and $\gamma$ the discount factor, for which $0 < \gamma < 1$ must hold [3, p. 504]. The function $\mathbb{E}$ denotes the expected total reward. This total reward equals the sum over the individual, discounted rewards $\gamma^k r_k$ for every time step $k$, when starting with state $s_0 = s$.

Discounting the rewards is done because rewards are of less value the further they are in the future because, to give only one reason, their accuracy decreases with time.

From equation 2.2 we deduce the optimal value function $V(s)$ of state $s$ for the optimal policy:

$$V(s) = \max_\pi \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_k \mid s_0 = s\right), \tag{2.3}$$

where $\max_\pi$ indicates that the term to the right of it is evaluated with the policy $\pi$ which makes the expression maximal [3, p. 504].

Furthermore, we can write equation 2.3 recursively as

$$\begin{aligned} V(s) &= \max_\pi \mathbb{E}\left(r_0 + \sum_{k=1}^{\infty} \gamma^k r_k \mid s_1 = s'\right) \\ &= \max_\pi \mathbb{E}(r_0 + \gamma V(s')), \end{aligned} \tag{2.4}$$

where $s'$ is the state following $s$ when an action $a_0$ has been performed and $r_0$ is the reward gained during said step.

Note equation 2.4 assumes actions selected according to an optimal $\pi$.

### 2.2.3 The Quality Function

We have thus far seen the policy function $\pi(s, a)$ as well as the value function $V(s)$.

Those two functions are related through the quality function $Q(s, a)$, which describes the quality of taking an action $a$ in a state $s$. The relation is described by the equations

$$\pi(s, a) = \text{argmax}_a Q(s, a) \tag{2.5}$$

and

$$V(s) = \max_a Q(s, a), \tag{2.6}$$

where $\text{argmax}_a$ evaluates to the argument $a$ making the term to the right of it maximal [3, p. 512].

The Quality Function is formally defined as

$$\begin{aligned} Q(s, a) &:= \mathbb{E}\left(R\left(s', s, a\right) + \gamma V\left(s'\right)\right) \\ &= \sum_{s'} P\left(s'|s, a\right)\left(R\left(s', s, a\right) + \gamma V\left(s'\right)\right), \end{aligned} \tag{2.7}$$

where $R(s', s, a)$ is the reward received after taking action $a$ in state $s$ leading to state $s'$ and $P(s'|s, a)$ the probability of state $s'$ following state $s$ when taking action $a$ [3, p. 512]. Since, in our case, the environments are deterministic, this probability is always equal to 1.

We now give a brief introduction into deep Reinforcement Learning and Neural Networks, before we get to DQL and NEAT.

### 2.2.4 Deep Reinforcement Learning

"I think it [deep Reinforcement Learning] is one of the most exciting fields in Artificial Intelligence. It's marrying the power and the ability of deep Neural Networks to represent and comprehend the world with the ability to act on that understanding. (...) Taken as a whole that's really what the creation of intelligent beings is. Understand the world and act." [7, 0:08]

As we saw earlier, one of the challenges of Reinforcement Learning is the approximation of the policy $\pi$. With traditional Reinforcement Learning this is done by ordinary functions. However, deep Reinforcement Learning uses Neural Networks, which can be used as universal function approximators, for this. The advantages of using Neural Networks are diverse. For one, thanks to their universality, Neural Networks are applicable to virtually all problems and do not have to be manually adjusted. They can also be improved automatically by the backpropagation algorithm (see algorithm 1). Many problems cannot be solved with traditional Reinforcement Learning because of their high dimensionality. For Neural Networks, high dimensions are no problem, as they shine in multi-dimensional function approximation, solving problems where ordinary functions would long be too computationally inefficient. One example of this are convolutional Neural Networks commonly used for image recognition.

However, they also come with challenges of their own, as Neural Networks are typically non-linear, increasing the risk for the learning process to diverge and become chaotic. The structures of the Neural Networks also have to be chosen carefully, particularly their dimensions, connections and activation functions.

## 2.3 Neural Networks

Nowadays, Neural Networks are used in a lot of applications. For example, they are used to classify objects of a given dataset. This might include finding all the antelope pictures in a dataset consisting of pictures of antelopes and penguins.

The basic idea of Neural Networks is to mimic the behaviour of the neurons of us humans. In fact, the first Neural Networks were designed to function as similarly as possible to the human neurons. However, with time the design of Neural Networks has changed as they were developed to better solve the problems they are meant to solve. [11, Chapter 1]

### 2.3.1 Mathematical Description

A Neural Network is essentially a multidimensional function. It takes a vector as an input and returns a vector as an output. The network is made out of multiple layers. The first is called the input layer, followed by some number of hidden layers and then the network is completed with an output layer. A Neural Network is called a "deep" Neural Network if it has at least two hidden layers. (However, the "deep" prefix is often used casually, not strictly following this definition.) Each layer consists of some number of neurons. The input layer has $n$ neurons if the dimension of the input vector is $n \times 1$. Given an input vector $\vec{v}$, we can set the activation of a neuron in the input layer. For the $j$-th neuron of the input layer the activation will be the $j$-th entry of the input vector. In general, each neuron of the $l$-th layer is connected with each neuron of the $(l+1)$-th layer.

Finally, there is a so-called bias for each neuron. This is an additional constant that is not depending on any activations and is added to the neuron's activation. [11, Chapter 1]

#### Notation

We mostly used the same notation as used in [11] (introduced there in Chapter 2). However, we did slightly change it and also introduced some new details.

A layer of the Neural Network is indexed by the variable $l$. The input and output layers are the layers with $l = 0$ and $l = L$ respectively. Given the $j$-th neuron in the $l$-th layer, we denote the activation of that neuron by $a_j^{(l)}$. Note that $(l)$ is not an exponent in this case. If we want to refer to the activations of all the nodes of layer $l$, we simply write $a^{(l)}$, which is a vector containing $a_j^{(l)}$ as the $j$-th entry. The bias for the $j$-th neuron in the $l$-th layer is denoted as $b_j^{(l)}$. Just as before we write $b^{(l)}$ for the biases vector of layer $l$. Furthermore, we

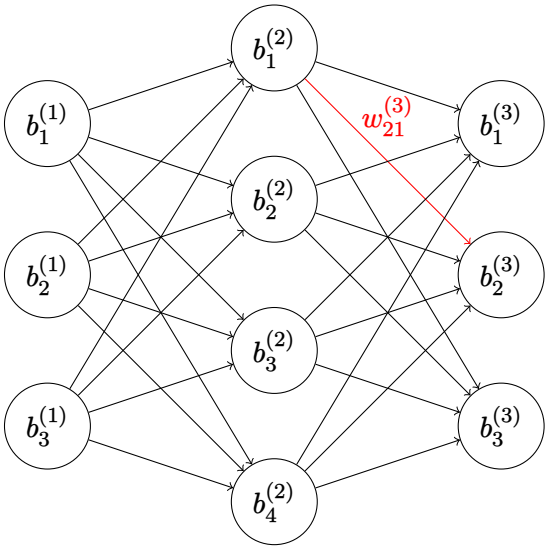



Figure 2.1: Example of a Neural Network

write $w_{jk}^{(l)}$ as the weight of the connection between the $k$-th neuron of layer $(l-1)$ to the $j$-th neuron of layer $l$. Additionally, we use a more compact way to refer to all the weights used to calculate the $l$-th layer from the $(l-1)$-th. For that we create a weight matrix $w^{(l)}$ which has $w_{jk}^{(l)}$ as entry in the $j$-th row and $k$-th column. The vector $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$ is called weighted input of layer $l$. We denote the activation function for layer $l$ as $\sigma_l$. If we write $\sigma^{(l)}$, we mean the vectorized function of $\sigma_l$. Thus, when applying $\sigma^{(l)}$ to a vector, the function $\sigma_l$ will be applied to each element of that vector. The activation function is the function that is applied to the weighted input which then results in the activation of the neuron. In addition, we will denote the derivative of $\sigma_l$ as $\varsigma_l$ and the corresponding vectorized form as $\varsigma^{(l)}$.

### Feed Forward

As mentioned previously, Neural Networks are essentially nothing more than functions. They take the input vector $\vec{v} = a^{(0)}$ as an input and then feed the activation to the next layer and so on until the last layer is reached. The activation of the last layer is then returned as the output vector $a^{(L)}$. For the $j$-th neuron in the $l$-th layer the activation is calculated from all the neurons in layer $(l-1)$ according to

$$a_j^{(l)} = \sigma_l\left(\sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}\right). \tag{2.8}$$

However, if we use the activation vectors, bias vectors and weight matrices, we can use the simplified notation:

$$a^{(l)} = \sigma^{(l)}(w^{(l)}a^{(l-1)} + b^{(l)}). \tag{2.9}$$

The right expression is equivalent to $\sigma^{(l)}(z^{(l)})$. In general, we therefore have

$$a_j^{(l)} = \sigma_l(z_j^{(l)}). \tag{2.10}$$

This shows the relation of the activation of a layer $l$ and the weighted input of the same layer. [11, Chapter 2]

### 2.3.2 Gradient Descent

Neural Networks need to learn to solve the task they are meant to solve, which is most often done by adjusting the network's weights and biases, but can also be achieved through manipulating its structure. There are multiple algorithms which achieve this goal. We use Stochastic Gradient Descent (SGD) in this paper. This learning algorithm needs a lot of data to work. For instance, if we want to distinguish pictures of antelopes and penguins, we need a big, labelled dataset of such pictures. Given such a database, we can use it to train the network using the SGD algorithm. The algorithm feeds a batch of pictures from the database through the Neural Network and calculates an error based on what the network's predictions are. This error is then propagated backwards through the Neural Network according to the four equations of backpropagation. Finally, the weights and biases of the Neural Network are adjusted using the calculated error.

#### The Loss and Cost Function

Before we can discuss the four equations of backpropagation, we need to talk about the loss and cost function. The difference between these two functions is that the loss function is applied to a single output while the cost function is applied to an entire batch of outputs. Given an input $\vec{x}$, the output vector $a^{(L)}(\vec{x})$ of the Neural Network and the corresponding label (a label denotes the desired 'model' output) $y(\vec{x})$, the loss function $\mathcal{L}$ is a measure of how much the output vector differs from the label. It is this function that we want to minimize in the end. After all, if $\mathcal{L}$ is 0, the output of the Neural Network would correspond perfectly to the label. [11, Chapter 2]

There are different loss functions that can be used. We will use the quadratic loss function. For a single training example $\vec{x}$ the quadratic loss function is defined as

$$\mathcal{L}(\vec{x}) = \frac{1}{2}||y(\vec{x}) - a^{(l)}(\vec{x})||^2, \tag{2.11}$$

where $||\cdot||$ denotes the length of the vector. For an entire training batch of $n$ examples we can simply take the mean of the single losses, we then get the cost function defined as

$$C = \frac{1}{n}\sum_{\vec{x}} \mathcal{L}(\vec{x}). \tag{2.12}$$

Note that this definition is not purposeful for all loss functions. However, it is meaningful for the quadratic loss function. [11, Chapter 2]

### The Error

Now that the loss and cost functions are defined, we continue by defining the error of a neuron. We will consider the $j$-th neuron in the $l$-th layer. Hence, the weighted input of that neuron is $z_j^{(l)}$. We can now consider the expression $\partial\mathcal{L}/\partial z_j^{(l)}$. This is the partial derivative of $\mathcal{L}$ with respect to $z_j^{(l)}$, which tells us how much $\mathcal{L}$ changes when $z_j^{(l)}$ changes. We now define the error $\delta_j^{(l)}$ of a neuron to be exactly this partial derivative:

$$\delta_j^{(l)} := \frac{\partial\mathcal{L}}{\partial z_j^{(l)}}. \tag{2.13}$$

If we say that the error of a neuron is big, then the activation as well as the weighted input of the neuron have a big negative impact on the loss $\mathcal{L}$. Because of that we want to minimize the error. Similarly, if the derivative of the loss with respect to the weighted input is big, then the weighted input has a big influence on $\mathcal{L}$. Hence, we see that this partial derivative has exactly the properties we want from a function describing the error. By tweaking the weights and biases in the right way, we can change $z_j^{(l)}$ to decrease the loss $\mathcal{L}$. On the other hand, if the error is small, then a change in the weighted input will not change $\mathcal{L}$ by much. [11, Chapter 2]

### The four Equations of Backpropagation

We can now start our discussion about the backpropagation equations. Those can be found in [11, Chapter 2] and are

$$\delta^{(L)} = \nabla_a\mathcal{L} \odot \varsigma^{(L)}(z^{(L)}), \tag{2.14}$$

$$\delta^{(l)} = ((w^{(l+1)})^T\delta^{(l+1)}) \odot \varsigma^{(l)}(z^{(l)}), \tag{2.15}$$

$$\frac{\partial\mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)} \text{ and} \tag{2.16}$$

$$\frac{\partial\mathcal{L}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \cdot \delta_j^{(l)}. \tag{2.17}$$

The $\odot$ denotes the Hadamard product for vectors. Unlike the dot or cross product, the Hadamard product multiplies vectors elementwise. That is, given two vectors $\vec{v}$ and $\vec{u}$, the product $\vec{v} \odot \vec{u}$ is given by

$$(\vec{v} \odot \vec{u})_j = v_j \cdot u_j. \tag{2.18}$$

While it is not necessary to use, it does simplify the notation a bit. [11, Chapter 2]

Furthermore, the expressions $\partial\mathcal{L}/\partial b_j^{(l)}$ and $\partial\mathcal{L}/\partial w_{jk}^{(l)}$ are the partial derivatives of $\mathcal{L}$ with

respect to $b_j^{(l)}$ and $w_{jk}^{(l)}$ respectively. So, for some given bias or weight in the Neural Network, these derivatives tell us how much the loss changes when changing that bias or weight. [11, Chapter 2]

Finally, the term $\nabla_a \mathcal{L}$ is a vector. The $j$-th element of the vector is defined to be $\partial \mathcal{L} / \partial a_j^{(L)}$, which is again a partial derivative of $\mathcal{L}$ with respect to some element of the output vector. [11, Chapter 2]

**The first Equation.** We will start by discussing and deriving equation 2.14. Rewriting the equation using components $j$, we get

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \cdot \varsigma^{(L)}(z_j^{(L)}). \tag{2.19}$$

To derive this equation, we rewrite $\varsigma^{(L)}$ and then use equation 2.10 to obtain

$$\varsigma^{(L)}(z_j^{(L)}) = \frac{\partial(\sigma^{(L)}(z_j^{(L)}))}{\partial z_j^{(L)}} = \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}. \tag{2.20}$$

And thus, we see that equation 2.19 becomes

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(L)}} \tag{2.21}$$

by the chain rule, which is our initial definition of the error.

For computational purposes we can rewrite $\nabla_a \mathcal{L}$ as $(a^{(L)} - y)$ as we use the quadratic loss function. Equation 2.14 then becomes

$$\delta^{(L)} = (a^{(L)} - y) \odot \varsigma^{(L)}(z^{(L)}). \tag{2.22}$$

This now enables us to calculate the error $\delta^{(L)}$ given the required information, which, except for $\varsigma^{(L)}(z^{(L)})$, is all calculated in a feed forward pass. So, once the Network has fed forward an input, the error of the Network's prediction in the last layer can be calculated using this formula.

**The second Equation.** As we have seen, the first equation defines the error of a given forward pass in the last layer $L$. However, we need the error of any arbitrary layer $l$. That is where the second equation is useful. This equation relates the error of a given layer $l$ to the next layer $(l+1)$. Thus, given the error $\delta^{(l+1)}$, we can calculate the error $\delta^{(l)}$ using the second equation. Since we have the error of the last layer, we can calculate the errors of all layers.

We now write equation 2.15 in component form:

$$\delta_j^{(l)} = \sum_h w_{hj}^{(l+1)} \cdot \delta_h^{(l+1)} \cdot \varsigma^{(l)}(z_j^{(l)}). \tag{2.23}$$

This equation can be derived from the definition of $\delta_j^{(l)}$ (equation 2.13), by applying the chain rule and using the same definition for $\delta_h^{(l+1)}$, which results in

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \sum_h \frac{\partial \mathcal{L}}{\partial z_h^{(l+1)}} \frac{\partial z_h^{(l+1)}}{\partial z_j^{(l)}} = \sum_h \delta_h^{(l+1)} \frac{\partial z_h^{(l+1)}}{\partial z_j^{(l)}}. \tag{2.24}$$

We then differentiate $z_h^{(l+1)}$ with respect to $z_j^{(l)}$:

$$\begin{aligned} z_h^{(l+1)} &= \sum_h w_{hj}^{(l+1)} \cdot a_j^{(l)} + b_h^{(l+1)}. \\ \frac{\partial z_h^{(l+1)}}{\partial z_j^{(l)}} &= w_{hj}^{(l+1)} \cdot \varsigma^{(l)}(z_j^{(l)}). \end{aligned} \tag{2.25}$$

We used the fact that $a_j^{(l)} = \sigma^{(l)}(z_j^{(l)})$ to calculate this derivative. If we now substitute this back into equation 2.24, we end up with

$$\delta_j^{(l)} = \sum_h w_{hj}^{(l+1)} \cdot \delta_h^{(l+1)} \cdot \varsigma^{(l)}(z_j^{(l)}), \tag{2.26}$$

which corresponds to equation 2.23.

**The third Equation.** For a Gradient Descent step, knowing the error of each layer is necessary, but not sufficient. This equation along with the next one is thus also needed, so as to know how to adjust the Neural Network given an error. We will now derive the third equation, which is the rate of change of the loss $\mathcal{L}$ with respect to the $j$-th bias in layer $l$. Equation 2.16 states that this partial derivative is equal to the error $\delta_j^{(l)}$ of the corresponding node. To derive this equation we use the chain rule to obtain

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}. \tag{2.27}$$

From equation 2.13 we know that the left factor of the right hand side of equation 2.27 is the error $\delta_j^{(l)}$. To simplify the right factor we will write the definition of $z^{(l)}$ in component form

and differentiate it with respect to $b_j^{(l)}$, leading to

$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_j^{(l-1)} + b_j^{(l)},$$
$$\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = 1. \tag{2.28}$$

As this derivative is 1, we can simplify equation 2.27 to be

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)}, \tag{2.29}$$

which is exactly equation 2.16 which we wanted to obtain.

**The fourth Equation.** The fourth equation is the rate of change of the loss with respect to a given weight $w_{jk}^{(l)}$. Just as the third equation helps us change the biases in the right direction, this equation defines how the weights must be adjusted.

For the derivation we start with the left-hand side of equation 2.17 and apply the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} \cdot \frac{\partial \mathcal{L}}{\partial z_j^{(l)}}. \tag{2.30}$$

As before, the rightmost factor is $\delta_j^{(l)}$. For the left factor we once again start with the definition of $z_j^{(l)}$, but this time we differentiate with respect to $w_{jk}^{(l)}$:

$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_j^{(l-1)} + b_j^{(l)},$$
$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = a_j^{(l-1)}. \tag{2.31}$$

We then insert these terms into equation 2.30 to obtain

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(l)}} = a_j^{(l-1)} \cdot \delta_j^{(l)}, \tag{2.32}$$

which is equation 2.17 that we wanted to obtain. [11, Chapter 2]

### The Stochastic Gradient Descent Algorithm (SGD)

Now that we have discussed and derived the four equations of backpropagation, we can finally have a look at how Neural Networks learn with the Gradient Descent algorithm (GD). Gradient Descent is an algorithm that uses the four equations of backpropagation to adjust the weights

and biases in the Neural Network based on the loss of a given training example. Oftentimes in practice, a modified version of GD called Stochastic Gradient Descent (SGD) is used. This is also the algorithm we used and implemented in this paper. What distinguishes SGD from GD is that SGD uses a whole batch of training examples while GD uses only one training example per epoch. An epoch is one iteration of backpropagation, that is, one iteration of adjusting all the weights and biases. SGD then uses the error of the individual training examples to calculate the cost. Due to the fact that the cost is defined as a sum over loss functions (see equation 2.12), the derivative of the cost function corresponds to the sum of the derivatives of the loss function, and thus, to the sum of errors (according to equation 2.13). [11, Chapter 2]

The SGD algorithm is central to DQL. However, it also works for itself, so we could use it to classify pictures of antelopes and penguins or to approximate a multivariable function.

The pseudocode for the SGD algorithm is shown in the listing for algorithm 1.

**Algorithm 1** Stochastic Gradient Descent

1: **procedure** $SGD(S, \eta)$:
2: $\Delta w = [Z \text{ with } \dim(Z) = \dim(w^{(l)}) \text{ and } (Z)_{ij} = 0 \ \forall i, j \text{ for each } w^{(l)}]$
3: $\Delta b = [z \text{ with } \dim(z) = \dim(b^{(l)}) \text{ and } z_i = 0 \ \forall i \text{ for each } b^{(l)}]$
4: **for all** $x \in S$ **do**
5: $a^{(1)} = x[0]$
6: **for all** $l \in \{2, 3, \dots, L\}$ **do**
7: $z^{(l)} = w^{(l)} \cdot a^{(l-1)} + b^{(l)}$
8: $a^{(l)} = \sigma^{(l)}(z^{(l)})$
9: **end for**
10: $\delta^{(L)} = \nabla_a \mathcal{L} \odot \varsigma^{(L)}(z^{(L)})$
11: **for all** $l \in \{L-1, L-2, \dots, 2\}$ **do**
12: $\delta^{(l)} = ((w^{(l+1)})^T \delta^{(l+1)}) \odot \varsigma^{(l)}(z^{(l)})$
13: **end for**
14: $\Delta w[l] = \Delta w[l] + \delta^{(l)} \cdot (a^{(l-1)})^T$
15: $\Delta b[l] = \Delta b[l] + \delta^{(l)}$
16: **end for**
17: **for all** $l \in \{L, L-1, \dots, 2\}$ **do**
18: $w^{(l)} = w^{(l)} - \frac{\eta}{m} \cdot \Delta w[l]$
19: $b^{(l)} = b^{(l)} - \frac{\eta}{m} \cdot \Delta b[l]$
20: **end for**
21: **end procedure**

The input $S$ is a set of $m$ training examples with their corresponding label. Furthermore, we have the learning rate $\eta$ which is a hyperparameter that can be changed. This learning rate controls the rate at which the weights and biases are adjusted in line 18 and 19. Algorithm 1 first creates two lists which will contain the summed partial derivatives of the loss with respect to the weights and biases respectively (lines 2 and 3). At first the $\Delta w$ list is initialized to contain an $n \times m$ matrix filled with zeros for each weight matrix $w^{(l)}$ with dimensions $n \times m$ (line 2). Similarly, the $\Delta b$ list is initialized with $n \times 1$ vectors filled with zeros for each bias

vector $b^{(l)}$ with the dimensions $n \times 1$ (line 3). Subsequently, in lines 4 to 16 the algorithm iterates over all the training examples $x$ in $S$. First, it sets the initial activation $a^{(1)}$ to the training example's activation $x[0]$ on line 5. Then, it feeds forward the activation to the last layer $L$ and saves all the $z^{(l)}$ and $a^{(l)}$ vectors (line 6 to 9). On line 10 it computes the error of the last layer $\delta^{(L)}$ according to equation 2.14 ($y = x[1]$, $\mathcal{L}$ is calculated as in equation 2.11). In the three following lines the error is then propagated backwards through the Neural Network according to equation 2.15. Afterwards, the $\Delta w$ and $\Delta b$ lists are updated on lines 14 and 15 respectively. To do this, equations 2.16 and 2.17 are used. Finally, after the loop over all training examples has terminated, the weights and biases are adjusted in lines 17 to 20. The average of the partial derivatives of $\mathcal{L}$ with respect to the two variables is taken by dividing the summed derivatives by the amount of training examples $m$. Then, that amount is multiplied by the learning rate $\eta$ and subtracted from all the weights and biases. [11, Chapter 2]

## 2.4 Deep Q-Learning (DQL)

Falling into the category of model-free, gradient-free and off-policy deep Reinforcement Learning, DQL might be called the "hippie" of Reinforcement Learning algorithms. We shall briefly explain those terms.

As a model-free algorithm, DQL does not rely on an environment in which it is possible to predict the state $s'$ when state $s$ and action $a$ are given. In other words, it works for non-deterministic environments. Being gradient-free means that the learning process of the algorithm does not depend on the gradient of the policy function. Being off-policy means that DQL does not learn by estimating the optimal policy $\pi$. It instead learns by estimating the quality function $Q(s, a)$ (hence the "Q" in DQL). Actions are chosen based on this quality function. In particular, at every step $s$ the action

$$a = \text{argmax}_a Q(s, a) \tag{2.33}$$

is taken. The term hippy is comically employed to describe the way in which Deep Q-Learning does not require a model, nor use a gradient, nor use a policy.

### 2.4.1 Q-Function Approximation

Fundamental to the idea of Q-Learning (with or without the "Deep") is the way in which the Q-Function approximation is optimized to get closer and closer to the real Q-function. This works by updating after every step $k$ (one step may equal, for example, one game tick) the old Q-function $Q^{\text{old}}(s_k, a_k)$ using the target $y$. The target $y$ describes the value which $Q^{\text{old}}$ should have had ideally, and is defined as

$$y := r_k + \gamma \text{max}_a \hat{Q}(s_{k+1}, a). \tag{2.34}$$

The term is composed of the reward $r_k$ received in step $k$ plus the discounted expected future reward [3, p. 517]. The function $\hat{Q}$ is a special target Q-Function which is only used to calculate the target $y$. The target Q-Function $\hat{Q}$ is fixed over multiple learning steps and updated only every $C$ steps to equal the Q-Function $Q$. This helps reduce the risk of divergence in the learning process. [10, p. 535]

Put together, this gives the equation

$$\begin{aligned} Q^{\text{new}}(s_k, a_k) &= Q^{\text{old}}(s_k, a_k) + \alpha \left( y - Q^{\text{old}}(s_k, a_k) \right) \\ &= Q^{\text{old}}(s_k, a_k) + \alpha \left( r_k + \gamma \text{max}_a \hat{Q}(s_{k+1}, a) - Q^{\text{old}}(s_k, a_k) \right), \end{aligned} \tag{2.35}$$

where $\alpha$ is the Q-learning rate, which dictates how quickly the Q-function is allowed to change. [3, p. 517]

**Adding on the "Deep".** To upgrade Q-Learning to DQL is as simple as using a Neural Network (called Deep Q-Network or DQN) for approximating the quality function. Updating the Q-function then works quite similarly, but instead of directly updating it we calculate the loss function using

$$\mathcal{L}_i(\theta_i) = \mathbb{E}\left[ \left( r_k + \gamma \text{max}_{a_{k+1}} \hat{Q}(s_{k+1}, a_{k+1}, \theta_i^-) - Q(s_k, a_k, \theta_i) \right)^2 \right], \tag{2.36}$$

where the index $i$ denotes the current iteration (iterations correspond to episodes in algorithm 2) and $\theta$ stands for the Neural Network's parameters (the weights and biases) [10, p. 529]. Notice the difference between $\theta$ and $\theta^-$. This version of DQL uses two networks. The first is the Q-Network $Q$ with parameters $\theta$ and the second is the target Q-Network $\hat{Q}$ with parameters $\theta^-$, which is used as the target Q-Function. Its parameters are updated every $C$ steps. [10, p. 535]

With this definition of the loss function we have everything we need to calculate the gradient according to equation 2.14 and perform SGD (described in algorithm 1) on the DQN, making for a fully-functional deep Reinforcement Learning algorithm. Algorithm 2 provides a pseudocode for DQL.

### 2.4.2 Experience Replay

Another speciality of this version of DQL is the use of experience replay. The experience replay mechanism uses a so-called replay memory of size $N$ to store the agent's last $N$ game experiences. A game experience (or "transition") $e_t$ at time $t$ takes the form

$$e_t = (s_t, a_t, r_t, s_{t+1}, \text{done}_t), \tag{2.37}$$

where $s_t$ is the state observed at time $t$, $a_t$ the action performed at said time step, which led to state $s_{t+1}$. The reward received at that time is $r_t$ and the Boolean $\text{done}_t$ denotes whether the current episode terminated at time $t$. As such, it corresponds to the variable done on line 8 of algorithm 2. [10, p. 529]

Deep Q-Learning with experience replay makes use of this replay memory by performing a Q-Learning update not only on the most recent few steps, but instead on random sample of steps (called "minibatch") of size $n$, which is taken from replay memory. With this procedure, the data is better distributed and randomized, thereby smoothening the learning process.

Interestingly, this type of learning is only possible for off-policy algorithms, because on-policy algorithms, which rely on actions taken based on the policy $\pi$, use the target

$$y' = r_k + \gamma Q(s_{k+1}, a_{k+1}) \tag{2.38}$$

with the action $a_{k+1} = \text{argmax}_a \pi(s_{k+1}, a)$. Due to the recursive nature of the calculation of $y'$, it also has to take the action $a_k = \text{argmax}_a \pi(s_k, a)$ to calculate the reward $r_k$ so as to have the best estimation of the future reward.

Q-Learning, being off-policy, does not use this target. It uses the target $y$ as defined by equation 2.34. As we can see, the optimal action is used to calculate the target $y$ regardless of action $a_k$. This infers that the target $y$ will be the best estimation for the expected future reward for an arbitrary action $a_k$, which can thus be taken and Q-Learning will be able to learn from it. [3, p. 517]

### 2.4.3 Exploration versus Exploitation

Finally, because Q-Learning is antifragile [14], it gains from the disorder introduced by sometimes not taking the action thought to be optimal. This can, for example, help avoid local minima during Q-Function approximation. The process is formally known as exploration, as opposed to exploitation, where the agent takes the action thought to be the best, thus "exploiting" its knowledge. Exploration is achieved by introducing a variable $\epsilon$ with $0 \leq \epsilon \leq 1$. At every step, the agent performs a random action with probability $\epsilon$ and the optimal action (according to the current Q-Function) with probability $1 - \epsilon$. The learning rate can be annealed throughout the training process according to

$$\epsilon = \max(\varepsilon^t, \epsilon_{min}), \tag{2.39}$$

where $\varepsilon$ is the annealing rate, $\epsilon_{min}$ a lower bound for $\epsilon$ and $t$ the current time step in the training process.

### 2.4.4 The Deep Q-Learning Algorithm

All of this put together makes for the DQL algorithm as described in algorithm 2.

**Algorithm 2** DQL with experience replay [10, p. 535]

1: **procedure** $DQL(M)$
2: $D$ = Array with size $N$
3: $Q$ = Neural Network with parameters $\theta$ initialized randomly
4: $\hat{Q}$ = Neural Network with parameters $\theta^- = \theta$
5: **for** episode = $0..M$ **do**
6: done = false
7: observe state $s_0$ from environment
8: **while** !done **do**
9: With probability $\epsilon$ select random action $a_t$
10: otherwise select $a_t = \text{argmax}_a Q(s_t, a, \theta)$
11: Play one game step with action $a_t$
12: Observe reward $r_t$ and new state $s_{t+1}$ and set done = true if episode terminated
13: Store transition $(s_t, a_t, r_t, s_{t+1}, \text{done})$ in $D$
14: Sample random minibatch of transitions $(s_j, a_j, r_j, s_{j+1}, \text{done}_j)$ from $D$
15: Initialize empty training batch $b$
16: **for all** transitions $\in$ minibatch **do**
17: Initialize array of target rewards $y_j = Q(s)$
18: $y_j[a_j] = r_j$
19: **if** not $\text{done}_j$ **then**
20: $y_j[a_j] = y_j[a_j] + \gamma \text{max}_a \hat{Q}\left(s_{j+1}, a, \theta^-\right)$
21: **end if**
22: Append $s_j$ and $y_j$ to $b$
23: **end for**
24: Perform SGD (see algorithm 1) on $b$ with respect to $\theta$
25: Every $C$ steps perform update $\hat{Q} = Q$
26: **end while**
27: **end for**
28: **end procedure**

Lines 2 to 4 initialize the networks and the replay memory. The for-loop on line 5 performs Deep Q-Learning for $M$ episodes. An episode may have an arbitrary length, though for games it is often defined to terminate after one game move, e.g. after a point is gained. The while-loop on line 8 performs one episode, the termination of which is indicated by the variable "done". In lines 8 to 11 the action to take is then calculated and performed in the environment. The following line stores the current step in replay memory as a transition. Lines 13 to 24 perform experience replay learning on a random minibatch of transitions. This means generating a training batch $b$ according to the Q-Learning method, which can then be used to update the weights and biases $\theta$ of $Q$. To do this, every target value $y_j$ corresponding to state $s_j$ is calculated and stored. The target value $y_j$ equals the reward received in the

transition plus the discounted estimate of rewards to be expected in the future, according to the target Q-Function $\hat{Q}$. Of course, if the episode terminates at the current time step, the expected future reward equals zero. Last, SGD is performed on $\theta$ with the generated training data in line 22. The $\hat{Q}$ function is updated every $C$ steps in line 24.

## 2.5 NeuroEvolution of Augmenting Topologies (NEAT)

The biggest challenge with Neural Networks is often to find the optimal topology, that is the number of layers, number of nodes per layer and connection weights, as these are fixed, but generally not known in advance. Neuroevolution algorithms try to solve this problem. This means that they try to find the best topology for a Neural Network automatically. In order to do this, they use genetic algorithms. NeuroEvolution of Augmenting Topologies (or, for short, NEAT) is, as the name suggests, a Neuroevolution algorithm. It was invented in 2002 by Kenneth O. Stanley. The algorithm's main point of interest is that it solves particular problems which commonly arise in other Neuroevolution algorithms. [13, p. 99]

### 2.5.1 Genetic Algorithms

Genetic algorithms use the antifragile nature of evolution (see [14]) to find a solution to a problem when a large search space is given [9, p. 5]. Such an algorithm needs some preliminaries to work. First, we need a population of candidates (also known as individuals) representing potential solutions. Each potential solution needs to be encoded by genes and summarized in a genome. [9, p. 6]

Second, we need a way to evaluate a potential solution given the corresponding genome. How this is done depends on the problem, but one approach is to simulate the problem with the potential solution. Afterwards, a fitness value is assigned to the potential solution according to a fitness function, which tells the algorithm how "fit" a given potential solution is. [9, p. 7]

Furthermore, we need a way to combine existing potential solutions to form new, possibly better ones. This is called generating new offspring and is done in the so-called crossover step. [9, p. 8]

Finally, the possibility of discovering an entirely new gene must be given. We have mutations therefor which replace some genes of the population's candidates with entirely new ones, or which slightly alter existing genes. This ensures that every possible candidate can theoretically be tested once. [9, p. 8]

Once these preconditions are fulfilled, we can start with a randomly initialized population as our first generation. All the individuals of this population are then evaluated by the fitness function and ordered by their fitness value. The fittest of them are allowed to produce offspring. Finally, some or all of the offspring of the new generation are mutated. This process is then repeated. We thus have generations (also called iterations) of candidates that are constantly

recombined to new generations and, with time, the average fitness as well as the best fitness of a given generation should improve. [9, pp. 8, 9]

### 2.5.2 Neuroevolution and NEAT

As already mentioned, NEAT is designed to solve some of the cruces common among other Neuroevolution algorithms [13, p. 99]. An overview of how this is done is given in the following paragraphs.

**Crossover between different Topologies.** When dealing with different topologies of Neural Networks, the crossover step of genetic algorithms is not that straightforward. We need an easy way to mate two candidates even if they have a completely different topology. NEAT solves this problem with the unique genome system it uses and with crossover, which we will discuss later. [13, p. 101]

**Competing Conventions Problem.** The problem of competing conventions is that we can sometimes have two candidates solving the same problem in entirely different ways. This might result in bad offspring when the two mate and therefore makes evolution less efficient. NEAT employs a principle inspired by homology (which is nature's solution to a problem similar to the competing conventions problem[2]) to solve the competing conventions problem. Each gene has a special innovation number. This innovation number is a historical marker, so it is an indicator to where it came from. Hence, two genes from two different candidates can be compared[3]. [13, pp. 103, 104]

**Protection of new Topologies.** If a new topology is mutated, the weights of the resulting Neural Network will not be optimized yet. Thus, this new topology might not survive for long, because other topologies with their weights already optimized have a better fitness. A way to protect a new topology is therefore needed. NEAT uses speciation with a shared fitness function. Speciation is a way to group together all individuals with a similar structure. Then, the average fitness of a species is compared to the average fitness of the other species, and, based on that, the number of offspring each species may produce is calculated. This assures that new topologies have a better chance of survival and are able to optimize their weights first before being compared to other species. It also enables the algorithm to search for a better topology and optimize the weights of already existing ones at the same time. [13, pp. 104, 105]

[2]The problem in nature is that genes do not have a fixed length. Therefore, if crossovers would happen on some random intervals of the chromosomes, the genes would be destroyed over time and complex life would not be possible. The way that nature solves this problem is by homology. Two genes which are alleles of the same trait are called homologous. Thus, nature has a sort of marker that prevents chunks of an allele to be swapped by some other data of another allele.

[3]The two genes having the same innovation number can be compared to them being homologous.

**Minimal Solutions.** Another problem in Neuroevolution is that networks often tend to become more and more complex as time goes on. However, generally, having more complex networks is not conducive to solving the tasks at hand. Networks do not generally become less complex over time, so starting out with a complex one will most probably never yield a network simpler than the initial one. Common Neuroevolution algorithms solve this problem by considering the complexity in the fitness function, so that candidates with a higher complexity have a lower fitness. This may however cause the fitness function to have different effects than those intended.

NEAT solves this differently, by starting with minimal structures, i.e. only the input and output nodes are given and no connections exist yet. The optimal solution and topology is then derived from this. The minimal starting point ensures the least complex solutions. [13, pp. 105, 106]

### 2.5.3 Genetic Encoding

NEAT uses so-called direct encoding. This means each node and connection in the Neural Network of a given candidate has a corresponding gene in the candidate's genome. [13, p. 101]

The node genes consist of their innovation number and a type specifying if they are an input, hidden or output node. Note that NEAT nodes do not have biases. The connection genes are defined by their start and end point (out- and in-node), their weight, a Boolean marker to indicate whether the connection is active and their innovation number. [13, p. 107]

### 2.5.4 Mutation

There are two types of mutations in NEAT, the structural mutation and the weight mutation. Structural mutations change the topology of the Neural Network, by adding either a node or a connection. Weight mutations either double the weight of a connection or randomly assign a new weight to a connection. [13, pp. 107, 108]

**Structural Mutations.** Adding a node (procedure *add_node* in algorithm 3) means that a randomly chosen connection $C$ in the Neural Network is split into two and a new node is inserted in between. To do this, the old connection is deactivated, then the new node $N$ is created, to which a new connection with weight 1, going from node $C.out$ to $N$, is added. Then, a second connection going from $N$ to $C.in$ is created. This connection will inherit the weight of $C$. [13, p. 107]

The mutation to add a connection (procedure *add_connection* in algorithm 3) randomly chooses two compatible nodes *out* and *in* and creates a connection with a random weight $w \in \mathbb{R}$ with $0 \leq w \leq 1$ going from *out* to *in*. For two nodes *out* and *in* to be compatible it is necessary that there is not already a connection between *out* and *in*, that *out* is not an output

and *in* is not an input layer node and that the new connection $out \rightarrow in$ does not make the network cyclic. [13, pp. 107, 108]

**Algorithm 3** Structural Mutations

**procedure** *add_node*($G$): ▷ The parameter $G$ represents the genome that is mutated.
    Choose random connection $C$ from $G$ with $C.active = true$
    $C.active = false$
    Create new node $N$
    Create new connection $C'$ with $C'.out = C.out$, $C'.in = N$, $C'.weight = 1$
    Create new connection $C''$ with $C''.out = N$, $C''.in = C.in$, $C''.weight = C.weight$
    Add $N$, $C'$ and $C''$ to $G$
**end procedure**

**procedure** *add_connection*($G$):
    Choose two random nodes $N_1$ and $N_2$ from $G$ that are compatible
    Create new connection $C$ with $C.out = N_1$, $C.in = N_2$, $C.weight = random(0, 1)$
    Add connection to $G$
**end procedure**

**Weight Mutations.** The weight mutations change the weights of connections in a genome. How this is done is determined by two probabilities. The first ($p_m$) represents the chance that the weight of a connection is mutated. The second ($p_d$) is the chance of that weight being doubled. If it is not doubled, then a new random weight is assigned to the connection. Weight mutations are shown in algorithm 4 [13, pp. 107, 111]

**Algorithm 4** Weight Mutations

**procedure** *weight_mutation*($G, p_m, p_d$):
    **for** Connection $C$ in $G$ **do**
        **if** $random(0, 1) < p_m$ **then**
            **if** $random(0, 1) < p_d$ **then**
                $C.weight = 2 \cdot C.weight$
            **else**
                $C.weight = random(0, 1)$
            **end if**
        **end if**
    **end for**
**end procedure**

### 2.5.5 Crossover

Crossover is the process of making a new child genome from two parent genomes. It is inspired by the process of sexual mating. Children should inherit some parts of both parent's genomes.

Assuming both parents had one allele of a gene, the ideal child would inherit one allele from each parent. For this to work we need a way to tell if two given alleles are of the same gene. This is solved elegantly by NEAT's innovation number system. In the crossover process, all the connection genes are aligned by their innovation number. Hence, alleles of the same genes can easily be identified. [13, pp. 108, 109]

In this alignment there are three types of genes. The matching genes are genes which are present in both genomes. Genes with an innovation number bigger than every innovation number of the other genome's genes are called excess genes. Genes which are neither matching nor excess are called disjoint genes. [13, pp. 108, 109]

The genome of the offspring is created by first randomly choosing one of the two alleles of each matching gene and then adding all excess and disjoint genes, too. The offspring is mutated immediately after creation. [13, pp. 108, 109]

This process is shown in figure 2.2.

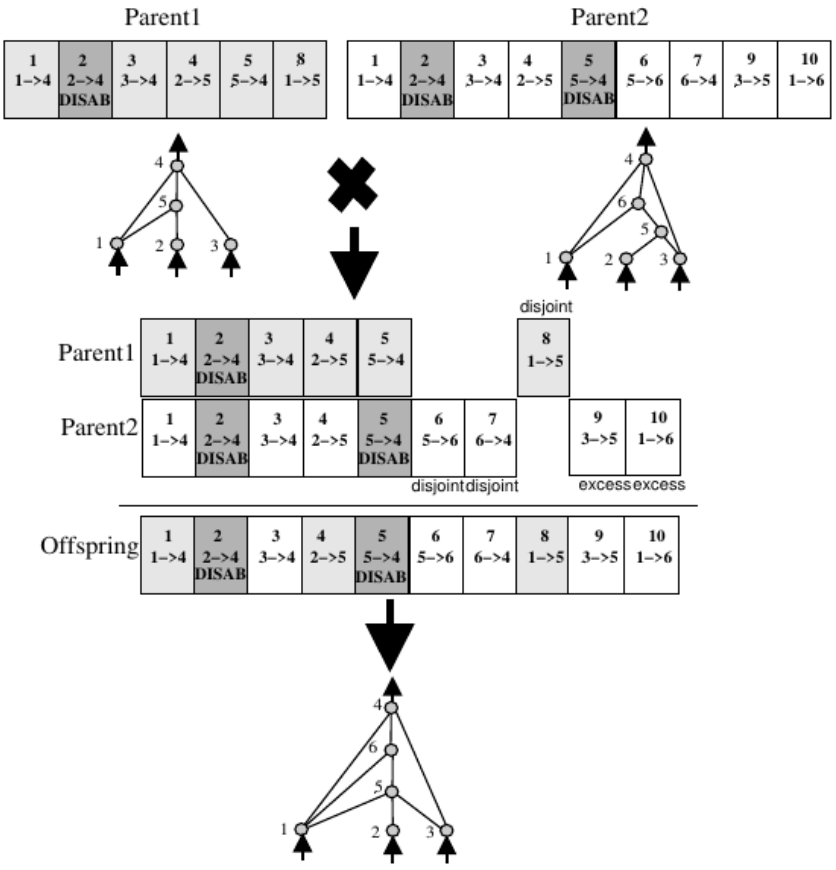



Figure 2.2: Alignment of Parent genes Parent1 and Parent2, showing matching, disjoint and excess genes. Taken from [13, p. 109]

.

### 2.5.6 Speciation

As explained in subsection 2.5.2, speciation is NEAT's way of protecting newly mutated topologies.

**Partitioning.** To partition a population into species we first need a way of measuring the similarity of two genomes. In order to do that we use the function $\delta : P \times P \to \mathbb{R}$ , which is defined as

$$\delta = \frac{c_1 \cdot E}{N} + \frac{c_2 \cdot D}{N} + c_3 \cdot \bar{W}. \tag{2.40}$$

The variables $E$ and $D$ denote the number of excess and disjoint genes respectively. $N$ is the number of genes in the bigger genome and $\bar{W}$ is the average weight difference of matching genes. In addition, we find three constants $c_1, c_2, c_3 \in \mathbb{R}$ that allow us to adjust the importance of each term. The set $P$ denotes the set of all candidates, i.e. the population. Thus, the function maps pairs of candidates to a real number. The bigger this number is, the more different are the candidates which are compared. [13, p. 110]

When creating a species we need a threshold to asses if a candidate has enough similarity to the species' representative (which can be any one of its candidates) to belong to said species. This threshold is defined by $\delta_t$. [13, p. 110]

Partitioning is done by iterating over all candidates $c$ and checking for every species $S$ if $c$ has a distance to the representative of $S$ which is lower than $\delta_t$. The candidate is added to the first species $S$ meeting this criterion. If the candidate does not fit into any existing species, a new species containing only this candidate is created. Algorithm 5 shows the speciation procedure. [13, p. 110]

**Fitness Sharing.** Once the species are created, NEAT uses explicit fitness sharing. This means that the candidates' fitness values are adjusted inversely proportionally to the size of the species they belong to. Thanks to this, no species can become too big and take over the population. [13, p. 110]

In particular, the fitness $f_c$ of each candidate $c$ is adjusted to $f'_c$ according to

$$f'_c = \frac{f_c}{|S(c)|}, \tag{2.41}$$

where $|S(c)|$ denotes the cardinality (number of candidates) of species $S(c)$ to which candidate $c$ belongs. [13, p. 110]

**Algorithm 5** Speciation

```
1: procedure speciation(P, δ_t):
2:     Let S be the list of species
3:     for candidate c in P do
4:         for species s in S do
5:             r_s = s[0]
6:             if δ(r_s, c) < δ_t then
7:                 Add c to s
8:                 Break
9:             end if
10:        end for
11:        if c does not fit in any species then
12:            create a new species s' = [c]
13:            add s' to S
14:        end if
15:    end for
16:    return S
17: end procedure
```

**Offspring.** Now, the base offspring $o(S)$ which each species $S$ is allowed to have is calculated according to

$$o(S) = \left\lfloor \frac{f(S)}{f(P)} \cdot |P| \right\rfloor . \tag{2.42}$$

In this equation, $f(S)$ is the total adjusted fitness of species $S$, i.e. the sum over adjusted fitness values of each candidate in $S$. Accordingly, we have $f(P)$ as the total adjusted fitness of the whole population $P$. Last, $|P|$ is defined as the size of the population. [13, p. 110]

After calculating the base offspring, additional offspring is assigned to the species which have the biggest $\frac{f(S)}{f(P)}$ modulo 1 value. This is done until $|P'| = |P|$ holds for the new population $P'$.

Now equipped with all the theory needed to understand the entirety of our paper, we shall go on to explain how we put all of this into practice, in particular how we went about programming our two ex-nihilo implementations.

# 3 Material and Methods

The last chapter explained all theory necessary to understand the algorithms we implemented and examined in this paper. In this chapter we explain specifically how we went about putting this theory into practical implementations as well as how we gathered and evaluated the data presented in chapter 4.

The product of the implementation is a Python program consisting of several files and folders, with the capability of running, testing and evaluating the DQL and NEAT algorithms on the games Cartpole and Pong, with the possibility of adding other games in the future. The source code will be made accessible on `https://github.com/fabiopanduri/matura` under the free GPLv3-only licence.

**Programming Workflow.** Our editor of choice is vim 8 (`https://www.vim.org/`).

We used the version control system git to keep track of the versions of our code, the changes we made, and as a means of protection from human error. We used a git server to host the code repository.

Pre-commit hooks provided by the program pre-commit (available via pip) helped us keep the code clean by enforcing the pep-8 standard, managing imports and more.

For running the code we used the program python3 from the Debian Bullseye repository.

## 3.1 Program Structure

We implemented the algorithms in Python 3. The modules we used are available with pip and installable through the requirements.txt file in the repo. We settled for the object-oriented programming paradigm as it is well suited for implementing Machine Learning algorithms, especially in Python, where it is the industry-standard for the field.

### 3.1.1 Folder Structure

Let us briefly list the relevant folders in the main branch[1] and explain their contents. Note the use of path notation relative to the git repository's top-level folder main.

[1] As visible in the logs, we also used other git branches during development. Those have all been merged into the main branch, which is the only branch relevant for the paper.

**main/dql** : DQL source code

**main/neat** : NEAT source code

**main/pong** : source code for the game Pong

**main/etc** : code used by various classes, e.g. activation functions

**main/eval** : performance evaluation code

The individual folders make for importable python modules, ideal for reusability. Additionally, the file main/main.py serves as a control file for the whole project. It uses flags to choose the algorithm, game and other necessary data for the algorithms. For example, the `-v` (verbose) flag prints additional information. The flags differ depending on the algorithm used. For this reason, there is also a `-h` flag which prints information on all available flags. In addition, there is a more detailed explanation in the file main/README.md on how to use the main.py file.

### 3.1.2 Source Documentation

We have written our source code to be self-documenting, assuming the reader knows the theory behind the respective algorithm as it is presented in chapter 2. The important parts of the algorithms line up neatly with their listings provided in the previous chapter, so we will often refer the reader there for further explanation. Where it is instrumental, we also provide an extract of the source code as a listing, to go into further detail on the specific part of the code. When we talk about specific snippets of the code, we will henceforth cite the file and the approximate line number of the snippet in question.

### 3.1.3 Working with Data

Since, of course, we would have to run our programs many times to collect enough training data for statistically significant results, we decided to write short bash scripts (which reside in main/eval) which automatically execute the code. Their structure is very simple, consisting of a loop executing a command $n$ times. The exact commands used are explained in subsections 3.2.3, 3.4.3 and 3.5.3. Saving the data is implemented within the Python programs, where the data is stored in the json format, ready for evaluation. All the data collection was done on a Raspberry Pi 4 (Model B, 4GB RAM), which we often ran over night to have a new set of data samples in the morning.

We also used Python 3 for the evaluation of the data. The evaluation code resides in main/eval/eval.py. It reads the data collected in the json files, processes the data (for instance calculating averages, standard deviations, and so on), and finally saves the calculated data along with a plot of it. This is how we generated the plots in chapters 4 and 5. More in-depth explanations of the individual evaluation procedures shall follow in the following sections as well as in chapter 5.

## 3.2 Neural Networks

One of the most mathematically difficult part of this paper was the implementation of Neural Networks and the SGD algorithm. It was especially challenging because we did not use a ready to-use-framework for this. We wanted to do everything ourselves, from scratch - ex-nihilo, as we call it.[2]

The theory presented in section 2.3, for which one of the central references was Michael Nielson's website `neuralnetworksanddeeplearning.com` [11], was the basis for our Neural Network implementation.

### 3.2.1 Implementation

We implemented Neural Networks as a class `NeuralNetwork` which resides in main/dql/neural_network/neural_network.py.

**Parameters.** The class `NeuralNetwork` has two main parameters. The first one is a list of integers `dimensions` whose $i$-th element corresponds to the number of nodes in the $i$-th layer of the network. The other one is a float value `eta` which is the learning rate $\eta$ used in SGD. Furthermore, the `NeuralNetwork` class has three parameters with an empty list as their default value. These are `activation_functions`, `weights` and `biases`. They can be used to initialize the Neural Network with predefined activation functions, weights or biases respectively. The `activation_functions` parameter is a list of strings which correspond to each layer's activation function. Activation functions are defined in the file main/etc/activation_functions.py. The `weights` parameter is a list of `numpy` arrays representing the weight matrices. The dimensions of those arrays must correspond to the Neural Network's dimensions. Lastly, the `biases` list of `numpy` arrays is the list of bias vectors for each layer.

**Class Methods.** The `initialize_network` method (l. 72ff.) initializes the weights and biases of the Neural Network with random numbers given by the `value_range` parameter. The most important methods are `feed_forward` (l. 153ff.) and `stochastic_gradient_descent` (l. 165ff.). The `feed_forward` method feeds forward an input activation and is thus the implementation of equation 2.9. The `stochastic_gradient_descent` is the method that performs the SGD algorithm as it is described in algorithm 1.

We want to highlight a specific part of the code that might be confusing at first.

```
1 weight_delta_sum[l] = weight_delta_sum[l] + \
2                       np.dot(delta[l][..., None],
3                              activation_list[l - 1][..., None].T)
```

[2] Admittedly, nothing ("nihilo") does not *literally* mean nothing, as we *did* have the hardware, the operating system, the programming language, its modules (especially `numpy` for mathematical calculations) and the literature at our disposal.

Specifically, the part `[..., None]` is probably confusing. This is a `numpy` specific notation which ensures that the dimensions of the two vectors are compatible. In the end, the only thing this code snippet does is implement equation 2.17 (line 14 of algorithm 1).

We left out the discussion of some functions here as their functionality does not need any explanation or because they are irrelevant to the final algorithm. For example, we have methods to save or load a Neural Network.

### 3.2.2 Testing and Debugging

To test the Neural Network and the SGD algorithm, we had it approximate the function

$$f(x) = \frac{\cos(x) + 1}{2}. \tag{3.1}$$

We did this approximation in the file main/dql/neural_network/sgd_test.py which first trains the network and then evaluates it.

We resorted to simple `print` statement debugging when bugs occurred, using the statements to get insights into the code and find sources of errors thereby.

### 3.2.3 Gathering and Evaluating Data

We let the network approximate the function shown in equation 3.1. To do so, we let the gathering script collect data about the network. The testing file generated a lot of function values and then applied SGD to these training samples. The accuracy of the network can then be seen by looking at the loss $\mathcal{L}$ of the network over time. Therefore, the loss with respect to time was measured by the script. We had the gathering script run the command
`python3 main.py sgd -e 2000 -b 100 -s`. The `-e` flag specifies the number of epochs the SGD algorithm will run, the `-b` flag sets the batch size and the `-s` flag saves the data to a json file.

After having collected multiple data samples with the gathering script, we used `numpy` to calculate the average and the standard deviation of the loss per epoch over all the training samples. The results of those calculations were then plotted with `matplotlib` and are shown in section 4.1.

## 3.3 Games

We used two games to test and evaluate the algorithms DQL and NEAT.

### 3.3.1 Cartpole

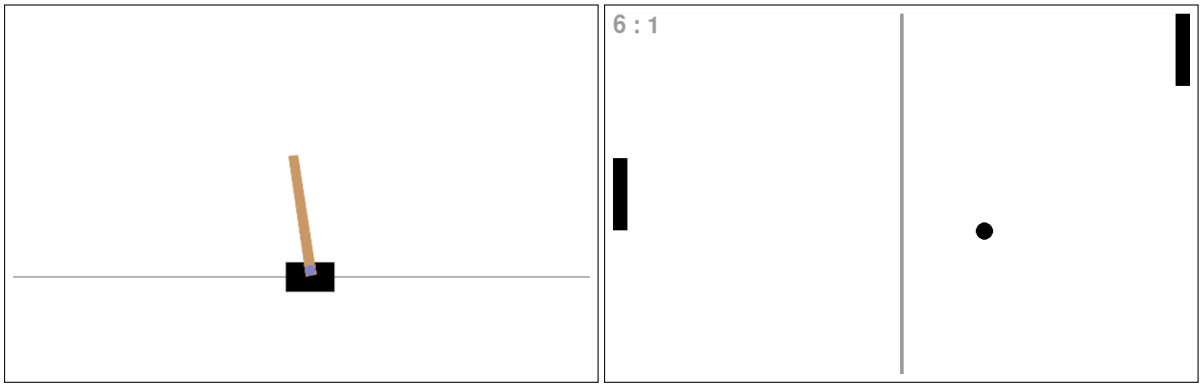
The first game, Cartpole, is a well-known benchmark for Machine Learning algorithms, with the trivial goal of balancing a stick (“pole”) atop a wagon (“cart”). The pole starts in the

vertical position and then falls according to the known rules of gravity. The agent has to counteract this by moving the cart to the right or to the left in every frame. Giving rewards is simple: +1 for every game step, unless the stick falls, in that case giving +0.

We did not implement this game ourselves and instead used OpenAI Gym's implementation "CartPole-v1" [2].

### 3.3.2 Pong

Subsequently, we used our own implementation of the all-time classic game Pong. We used `pygame` to render the game graphically, the rest of the implementation is done ex-nihilo, it resides in main/pong. Pong is a two-player game with the objective of not letting a ball fly off the screen, a bit like ping-pong. For that, each player has a paddle which diverts the ball when it is hit, according to the known rules of rigid body dynamics. In every frame, this paddle may be moved up, down, or not at all. Although Pong is a two player game, we decided that the agents should play against a stationary opponent during training, as this simplifies the implementation and will not affect performance negatively. Giving rewards is more complicated as we used multiple reward systems. We explain the different reward systems in subsection 3.4.1. Figure 3.1b shows our Pong playing-field[3], the agent on the right side and the stationary opponent on the left side of the screen. In the top left we see the score, which is 6:1 in favour of the opponent.




(a) Cartpole (b) Pong

Figure 3.1: Two Reinforcement Learning Problems

[3]The colours are inverted in the picture, so as to save printer ink.

## 3.4 Deep Q-Learning

The implementation of the DQL algorithm in this paper is based on the work of the DeepMind team as published in Nature [10, p. 529ff.]. The relevant theory is described in section 2.4. Reference implementation [8] provided some additional help for our implementation. Apart from that, we implemented DQL ex-nihilo, same as SGD.

### 3.4.1 Implementation

We did not attempt to solve the games by feeding the agent screen pixels (which was the DeepMind team's method of choice), as we deemed it to be too large of a task. We instead used variables directly describing the game's state, for example the cart's position and the pole's angle for Cartpole or the positions of the ball and the paddles for Pong. The DQL files lie in main/dql, which in turn contains code for the agent, the environment as well as the Neural Network described in the previous section.

#### Agent

The DQL agent is implemented as class `DQLAgent` which starts on line 54 in dql/agent/agent.py. The Deep Q-Learning algorithm as described in algorithm 2 is implemented in said class in the methods `learn` (l. 196ff.) and `replay` (l. 167ff.). The `learn` method performs `n_of_episodes` many DQL learning episodes. An episode usually lasts until a point is gained or, in the case of Cartpole, until the pole fell down. This is indicated by the `terminated` variable being set to `True` in the environment. Additionally, an episode lasts at most `self.max_simulation_time` many steps.

Below, we present the innermost for-loop of the the `replay` method, lines 177ff. in dql/agent/agent.py. In the excerpt, which corresponds directly to lines 16 to 23 of algorithm 2, the list of target rewards is prepared to be added to the current training batch, along with the list of states. The format of `transition` corresponds to the one used in lines 13 and 14 of algorithm 2, except for the variable `phi`, which is the preprocessed state $s_j$. Preprocessing is a technique used by DeepMind to reduce the size of the pixel data fed to their agent [10, p. 534]. Because we do not use pixel data, it is currently implemented as a placeholder without functionality.

```
1 for transition in minibatch:
2     phi, action, reward, next_phi, done = transition
3     target_rewards = self.q_network.feed_forward(phi)
4     taken = self.possible_actions.index(action[0])
5     target_rewards[taken] = reward
6     if not done:
7         target_rewards[taken] += self.discount_factor * np.max(
8             self.target_q_network.feed_forward(next_phi))
9     training_batch.append((np.array(phi), target_rewards))
```

The `ReplayMemory` class (l. 26ff.) is a helper class for interacting with the replay memory tuple.

**Hyperparameters.** Hyperparameters are fixed via the dql/config.py file. We tested many different combinations of hyperparameters until we achieved satisfactory learning results. The hyperparameters which we used to collect data about the algorithm (shown in section 4.2) are listed below.

- Neural Network (nodes per layer): [(game-dependent, equals observation space size), 24, 24, (game-dependent, equals action space size)]
- Activation functions per layer: [(input layer), ReLU, ReLU, linear]
- Replay memory size (see subsection 2.4.2): $N = 2000$
- Minibatch size (see subsection 2.4.2): $n = 32$
- Discount factor (see equation 2.34): $\gamma = 0.95$
- Learning rate for SGD (see subsection 2.3.2): $\eta = 0.001$
- Annealing rate for $\epsilon$: $\varepsilon = 0.99$
- Lower bound for $\epsilon$: $\epsilon_{min} = 0.1$
- Target network update frequency (see subsection 2.4.4): $C = 100$

### Environments

The environments are implemented as classes `CartpoleEnvDQL` and `PongEnvDQL` in the folder dql/envrionment. Those classes are wrappers to the respective games for the agent to interact with. In the following listing we see a general, bare-bones environment class with its essential methods `fitness`, `make_observation` and `step`. The `Env` class uses the `game` object which must itself be a class which implements a game of choice. It must have methods to perform a game step (`game.step`), return the current game state (`game.observe`) and optionally to render the state (`game.render`).

```
1  class Env:
2      def __init__(self, render=True):
3          self.env = game
4          self.possible_actions = ["action0", "action1"]
5          self.state_size = len(self.make_observation())
6          self.render = render
7      def fitness(self, t, reward):
8          return t
9      def make_observation(self):
10         return self.env.observe()
11     def step(self, action):
12         observation, reward, done = self.env.step(action)
```

```
13          # if reward is not returned by step, it is calculated here
14          if self.render:
15              self.env.render()
16          return observation, reward, done
```

The `step` method is called by the agent to perform a game step. It takes as input the action and returns the next game state `observation`, calculated by `make_observation`, the `reward` and the Boolean `terminated` which indicates whether the game terminated at the current time step. The `fitness` method is used to evaluate the agent's performance. It is explained in detail in subsection 3.5.1.

**Observation Space.** As mentioned before, we did not feed the agent the raw pixel data as input. What we used instead were direct numeric parameters. The parameters used are defined in `make_observation` in the environment. For Cartpole, those are the pole's angle and angular velocity as well as the cart's position and velocity. These parameters are given by Gym's Cartpole implementation. For Pong, what we ended up using was the paddle's as well as the ball's $y$-coordinate relative to the screen window size.

**Rewards.** For Cartpole, we used Gym's reward system of $+1.0$ for every step in which the pole did not fall.

For Pong, we tested four different reward systems. The reward system is selected with the `--reward-system` flag. The first system, called v0 in code, gives a reward of $+1.0$ if the agent scored a point, $-1.0$ if the opponent scored a point and $+0$ if no one scored a point in the step. This corresponds to the reward system used by DeepMind as explained in [10, p. 534]: "In addition it [the agent] receives a reward $r_t$ representing the change in game score".

The second system (v1) gives a reward of $+1.0$ if the agent hits the paddle and $+0$ if not. Next, we tried a system (v2) similar to the one used in Cartpole. That is, $+1.0$ for every step, except for $-b$ for losing and $+b$ for winning frames, with $b > 1.0$.

The last reward system (v3) gives a reward of $+1.0$ for every step in which the ball's vertical coordinate is in between the paddles lower and upper vertical border and $+0$ if is outside. In other words, it rewards the agent for having the paddle on the same height as the ball. Admittedly, this last reward system could be called "cheating", as, in this way, we already tell the agent how it should play Pong. We discuss this in detail in subsection 5.2.2.

The performances of the different reward systems are discussed in section 5.2.

### 3.4.2 Testing and Debugging

Because the initial writing and debugging phase was rather short and consisted of only a few intense programming sessions, and thanks to Python's generous error messages, the algorithm

started to work without crashing quite fast. That the program did not crash, however, did not mean that the algorithm was actually learning.

Getting this to work was the greater challenge than writing the code. It mostly consisted of many hours of trial-and-error, running the code, seeing how well it performed, changing one or two hyperparameters, running it again, and so on, until desirable learning results were achieved.

An indispensable help in tracking the performance during training were the live plots and the `--verbose` flag of the main.py file. The former is enabled with the `-l` flag and provides a `matplotlib` plot resembling the ones used in chapter 4 which is updated after every episode, thus giving us immediate feedback on the algorithm's performance. We used the latter in conjunction with `print` debugging, for example to print rewards or network parameters at regular intervals, so as to see if they developed as expected or if there was unwanted behaviour.

### 3.4.3 Gathering and Evaluating Data

With the data collection method described in subsection 3.1.3, we collected data about the training performance of DQL. We wanted to be able to compare the collected data with data collected while training the NEAT algorithm. For this we used the `fitness` method, which we defined to be the same for DQL and NEAT. DQL does not need a `fitness` function except for its evaluating, as such we could use NEAT's fitness function which is described in subsection 3.5.1. In addition to the fitness, we measured the time each episode took in seconds.

The command which the bash script ran to start the program was
`python3 main.py dql -g cartpole -e 1000 -s` for Cartpole and
`python3 main.py dql -g pong -e 200 -s --reward-system vx` for Pong. The flag `-g` tells the program which game should be tested, `-e` specifies how many episodes should be played ($M$ in algorithm 2) and `-s` enables saving the data as json files. The flag `--reward-system` is Pong-specific and is used to specify which reward system should be used.

With those commands, we collected multiple data samples, each consisting of the same number of episodes. We subsequently used `numpy` to calculate the mean fitness per episode, its standard deviation as well as average time per episode and its antiderivative over all the samples and plotted these values with `matplotlib`. The plots are shown in section 4.2.

## 3.5 NeuroEvolution of Augmenting Topologies

We based our NEAT implementation on the original NEAT paper by K. Stanley and R. Miikkulainen [13]. Their generous theoretical explanation of the algorithm was mostly sufficient for the ex-nihilo implementation of NEAT. It was the implementation details which we had to figure out for ourselves, as the explanation is only theoretical and does not include deeper insight into practical nuances.

### 3.5.1 Implementation

Because we did not use pixel data as input for DQL, we could use the exact same input values for DQL and NEAT. This is necessary to be able to compare the algorithms. Code for NEAT resides in the folder main/neat. It contains the files which implement the NEAT algorithm as well as the environments for Cartpole and Pong.

#### Agent

The most important agent files are neat/genetics.py, which implements the genome system, and neat/neat.py, which implements the genetic algorithm NEAT and trains the agent therewith.

**Genetics.** The file neat/genetics.py, which we shall now explain, implements the genetics underlying NEAT. Nodes and connections are implemented as classes `NodeGene` (l. 20ff.) and `ConnectionGene` (l. 48ff.) with all the attributes they need as described in subsection 2.5.3.

Furthermore, we have the `Genome` class (l. 66ff.) that implements a genome. We also decided to implement the delta function (equation 2.40) and the crossover functionality as methods of the `Genome` class. The mutations described in subsection 2.5.4 are methods of the `Genome` class as well. For the connection mutation to work, we needed a way to detect and eliminate cycles. We used a slightly adapted version of an old-fashioned depth-first-search algorithm, implemented on lines 352ff., to do so. Moreover, we used a dynamic programming approach for the `feed_forward` method (l. 484ff.), which is necessary since NEAT's Neural Networks do not have layers.

In particular, the `feed_forward` method sets the activations of the input nodes in a table (dictionary on l. 490) and calls the `calculate_node` method on each output node. Once the `calculate_node` method (l. 456ff.) is called on a node, it checks by aid of said table if the node has already been calculated. If it has not, it recursively calls itself on all the node's predecessors to have all the values it needs for the calculation of the node's activation. Finally, the `Genome` class has some further methods to create, save or draw the network.

**Genetic Algorithm.** We now have the underlying genetics covered and can thus move on to explain the genetic algorithm which trains the agent. The genetic algorithm NEAT resides as class `NEAT` on lines 18ff. in neat/neat.py. It is separated into five main methods.

These are:

- `simulate_population` (l. 177ff.)
- `speciation` (l. 220ff.)
- `adjust_population_fitness` (l. 266ff.)
- `get_species_sizes` (l. 276ff.)
- `mate` (l. 342ff.)

Additionally, the `iterate` method (l. 76ff.) combines all the mechanisms of the algorithm, saves the collected data and prints information about each generation. It can be thought of as the `main` function of the NEAT algorithm.

The method `simulate_population` implements the simulation process described in subsection 2.5.1. The following code is taken from said method.

```
1  for t in range(max_t):
2      state, reward, terminated = env.step(action, t)
3      if terminated:
4          individual.fitness = env.fitness(
5              t, reward, alpha=self.alpha)
6          break
7      prediction = individual.feed_forward(state)
8      action_i = np.argmax(np.array(prediction))
9      action = env.possible_actions[action_i]
10 else:
11     individual.fitness = env.done_fitness
```

The method simulates a game step (l. 2), decides on an action to take in the next step (l. 7-9) and terminates if the `terminated` Boolean is `True` (l. 3) or if `max_t` many steps have been played. Note that we used a for-else clause (l. 1 & 10). This means that line 11 is executed if and only if the for-loop did not break, which means the game's termination was forced after `max_t` steps, in which case the current candidate has to be assigned the `self.done_fitness` value.

The `speciation` method implements speciation as described in algorithm 5, using the `adjust_population_fitness` method for fitness sharing (equation 2.41) and the `get_species_sizes` method to calculate the number of offspring each species is allowed to have according to equation 2.42. The `mate` method, of which some lines are shown in the following listing (l. 342-378 of neat.py) implements the mating step. For each species it randomly chooses two parents which then create a new child.

```
sorted_s = sorted(s, key=lambda x: - x.fitness)
l = max(math.ceil(len(sorted_s) * self.r), 1)
mating_s = sorted_s[0:l]
N = new_N[s_index]
i = 0
while i < N:
    if i == 0:
         best.append(mating_s[0])
         i += 1
         continue
     elif len(mating_s) == 1:
         c = Genome.load_network_from_raw_data(
             mating_s[0].save_network_raw_data())
```

```
14              new_generation.append(c)
15              i += 1
16              continue
17          p1, p2 = random.sample(mating_s, k=2)
18          child = Genome.crossover(
19              p1, p2, self.connection_disable_constant)
```

Lines 1 to 3 create the list `mating_s` containing the fittest `self.r` percent of candidates. Only these are then allowed to have offspring. The `max` on line 2 ensures that at least one candidate is in the `mating_s` list. The variable `N` on line 4 is the number of offspring the species is allowed to have, which has been calculated by the `get_species_sizes` method. The offspring is then created in the while-loop. First, the best individual is copied to the new species (l. 7-10). This is done by copying it to the `best` list, which prevents it from being mutated (we want to preserve the best candidate as-is in the new species). Next, lines 11 to 16 implement the special case where the `mating_s` list has only one candidate. In this case, the sole candidate is directly copied to the `new_generation`, being mutated afterwards. The saving and loading of the network on lines 12 and 13 is needed as copying the candidate with the `=` operator would copy it by reference, breaking the whole algorithm. Last, the mating is performed, crossing over two different candidates of the species on lines 17 to 19. What follows the lines shown in the listing in the source code (l. 380ff.) is a depth-first-search algorithm which prevents cycles from occurring in the network after the mating step.

**Hyperparameters** The hyperparameters of NEAT are defined via neat/config.py. We used different hyperparameters until we found a configuration that worked fine. For Cartpole, we used the following:

- Population size: $|P| = 150$
- Speciation constants (see subsection 2.5.6): $c_0 = 1$, $c_1 = 1$, $c_2 = 0.4$
- Delta threshold (see equation 2.40): $\delta_t = 2$
- Weight mutation probabilities (see subsection 2.5.4): $p_m = 0.8$, $p_d = 0.9$
- Node mutation probability (see subsection 2.5.4): 0.001
- Connection mutation probability (see subsection 2.5.4): 0.001
- Chance that a matching gene is disabled in offspring if one parent has it disabled: 0.75
- Percentage that is allowed to have offspring per species: $r = 0.5$
- Time steps after which the simulation is stopped: $max\_t = 10000$
- Alpha value for fitness calculation: $\alpha = 2000$

### Environments

The environments for NEAT are implemented as classes `PongEnvNEAT` and `CartpoleEnvNEAT` in the files neat/cartpole_gym_env.py and neat/pong_env.py. Their structure corresponds

to the structure of the DQL environments as described in subsection 3.4.1. The `step` method is called by the `simulate_population` method, which lies in neat/neat.py, through which the candidates interact with the environment.

**Rewards and fitness.** NEAT uses the same reward systems as DQL for both Cartpole and Pong.

The fitness to judge the individual candidates (as explained in subsection 2.5.1) was defined in the respective environment's `fitness` method. The same fitness method was used to gather data about NEAT's as well as DQL's performance.

For Cartpole, fitness was defined as

$$f = t, \tag{3.2}$$

that is the amount of time steps $t$ the current episode took.

For Pong, fitness was dependent on the reward system used. For systems v0 to v2, it was defined as

$$f = \begin{cases} 1 + e^{-\frac{t}{\alpha}} & r > 0, \\ 1 - e^{-\frac{t}{\alpha}} & r < 0, \\ 1 & r = 0, \end{cases} \tag{3.3}$$

where $r$ is the reward observed at the current time step. Fitness parameter $\alpha$ dictates the decay of the exponential function. We used this exponential function to ensure that the fitness is always positive, also when $r < 0$, because NEAT's fitness must never be negative, as equation 2.42 relies on positive fitness values.

For reward system v3, fitness was defined as

$$f = \frac{1}{t_{tot}} \sum_{t=0}^{t_{tot}} r_t, \tag{3.4}$$

where $t_{tot}$ is the amount of time steps the current game iteration (respectively episode for DQL) took and $r_t$ the reward received at time $t$.

### 3.5.2 Testing and Debugging

We tested NEAT using methods similar to the ones we employed with DQL. That is, we let it play Cartpole or Pong and observed its behaviour. This was again done with live plots and `print` statements.

We tried a lot of hyperparameter configurations to find one that worked acceptably well. We performed this task manually by changing the hyperparameters and observing the result.

When python errors occurred, we again used mostly `print` debugging and wise interpretation of error messages.

When debugging non-python errors (most often caused by `numpy`) like segmentation faults, we had a couple of more advanced tools at our disposal. These include the python3 flags `-X option` and `-d`, the python debugger pdb and python's `faulthandler` module. Additionally, gdb's backtrace was useful for tracing segmentation faults. PyCharm's debugging features like breakpoints and expression evaluation also came in handy to trace the sources of infinite loops. Finally, the `sys` module's `settrace` function can give a more detailed traceback (consisting of multiple gigabytes of data).

### 3.5.3 Gathering and Evaluating Data

The general data collection mechanism is the same as with DQL, except for the fact that NEAT uses generations where DQL uses episodes. This difference arises because NEAT always trains multiple candidates in one generation, whereas one DQL episode consists of only one agent. For this reason, we also collected two fitness values for every generation: One is the fitness of the best candidate (which does not have to be the same one for every generation) and the other is the average fitness of the entire population. Other than that, we also collected the time each generation took.

To collect the data, the bash script ran the command
`python3 main.py neat -g cartpole -i 1000 -s` for Cartpole and
`python3 main.py neat -g pong -i 200 -s --reward-system vx` for Pong. The `-i` flag denotes how many generations should be played.

Subsequently, we calculated and plotted the same values as we did with DQL.

In the next chapter, we present all the data which we collected with the techniques described above.

# 4 Results

In this chapter we present the data collected while training the algorithms. The evaluation of the data and the discussion of the results follows in the next chapter.

Data is presented in the form of `matplotlib` plots. The raw data used to generate those plots is accessible in the code repository (`https://github.com/fabiopanduri/matura`).

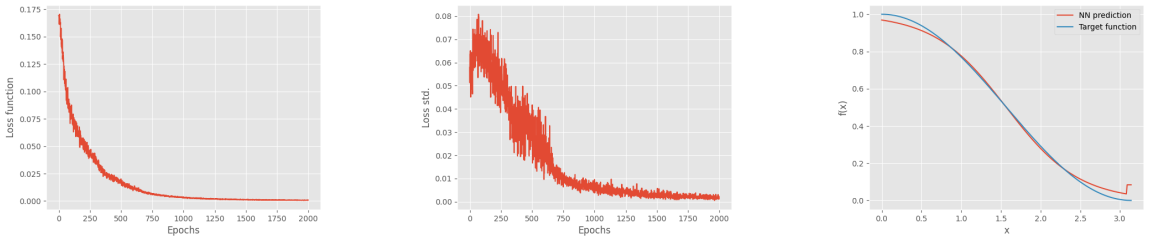
## 4.1 Stochastic Gradient Descent

Although the SGD algorithm as such is not a central part of this paper, we had to evaluate the algorithm outside its use in DQL, to ensure that it is working properly. In this way we knew that it was not the source of error when debugging DQL.

Its performance in approximating the function

$$f(x) = \frac{\cos(x) + 1}{2} \tag{4.1}$$

as defined in equation 3.1 is shown in figure 4.1a. The figure shows the trend of the square of the deviation from the desired value (which is equal to the loss) during SDG training. This trend is the average of 58 learning samples of 2000 epochs each. We see that it decreases rapidly, being below 0.010 at 1000 epochs and almost at 0.000 at 2000 epochs. The corresponding standard deviation is shown in figure 4.1b. It also decreases with each epoch and reaches a value of below 0.005 at 2000 epochs. Additionally, figure 4.1c shows a sample prediction of the approximated function after having learned for 2000 epochs. We see that the Neural Network's prediction lines up closely with the target. Notice the spike at around 3.1 on the $x$-axis, which results from an inaccuracy in the network's prediction.



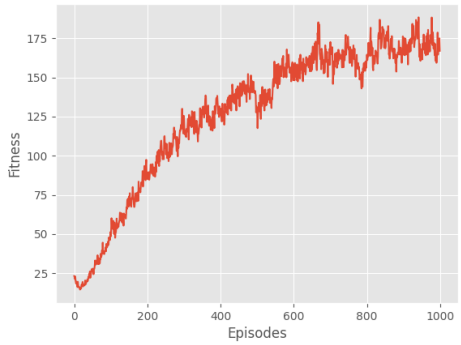

(a) Loss (b) Standard deviation (c) Sample

Figure 4.1: SGD on cosine data

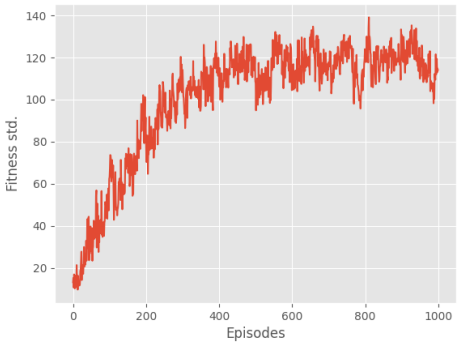
## 4.2 Deep Q-Learning

We start with the data gathered during the training of DQL on Cartpole. The agent was always trained for 1000 episodes. We gathered 159 learning samples, the trends shown in figure 4.2 depict the average of all samples. Figure 4.2a shows the mean fitness throughout the learning episodes. We see that the mean fitness is steadily increasing, reaching 100 after 200 episodes, but not exceeding 190 after the 1000 episodes. The standard deviation of said mean is shown in figure 4.2b. It approaches 140 at episode 200 and never exceeds it. Furthermore, we show the average time per episode in figure 4.2c. We see that the time increases at a rate similar to the fitness, not exceeding 7 seconds per episode. Last, we take a look at the average accumulated time for each episode $e$ ($x$-axis), that is the average total time in seconds it took to simulate all episodes up to episode $e$, in figure 4.2d. The accumulated time is therefore the antiderivative of the time per episode. It increases faster than linearly, 500 episodes taking about 1500 seconds and the whole 1000 episodes taking over 4250 seconds on average.

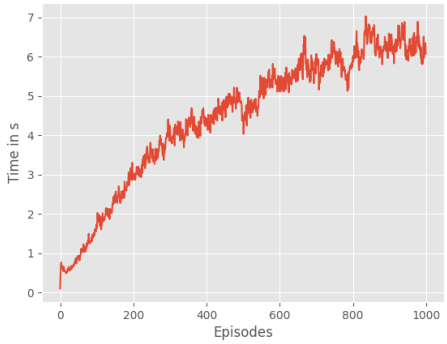
DQL only managed to learn Pong by use of reward system v3. We have thus omitted the results for reward systems v1 and v2, as they resemble the results for v0 and do not provide additional insight. In subsection 5.2.2 we discuss the performance of the different reward systems. We collected 48 samples of 200 episodes each for reward system v0 and 50 samples of 200 episodes each for reward system v3. Figures 4.3a and 4.3c show the fitness throughout the learning process as a mean of those samples, for reward systems v0 and v3, respectively. It is evident that the fitness does not increase with reward system v0. The standard deviation also does not change. Using system v3, however, we see an increase of the fitness from around 0.20 to around 0.35. The standard deviation increases at almost the same rate.



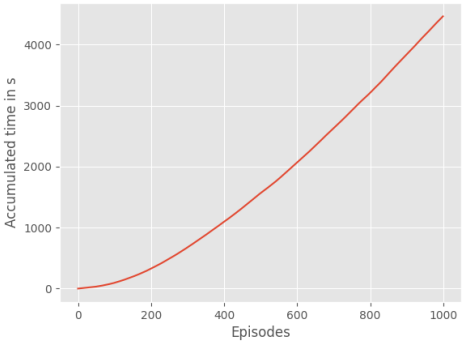

(a) Mean




(b) Standard deviation

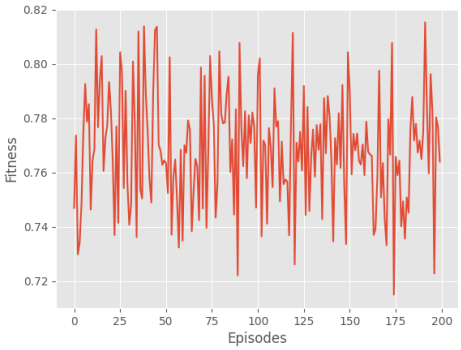



(c) Time



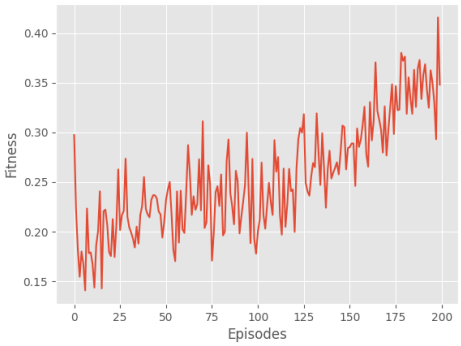

(d) Accumulated time

Figure 4.2: DQL on Cartpole data

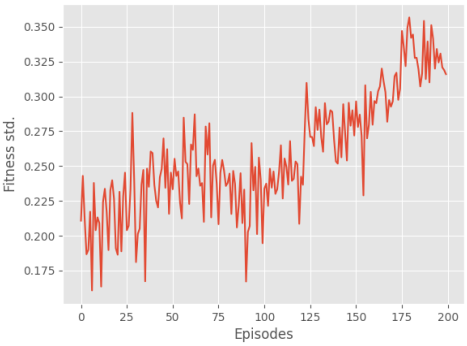



(a) Mean (v0)


(b) Standard deviation (v0)




(c) Mean (v3)




(d) Standard deviation (v3)

Figure 4.3: DQL on Pong data

## 4.3 NeuroEvolution of Augmenting Topologies

As NEAT has a `fitness` value for both the best and the average of candidates, we will henceforth show those values as two graphs within the same plot.

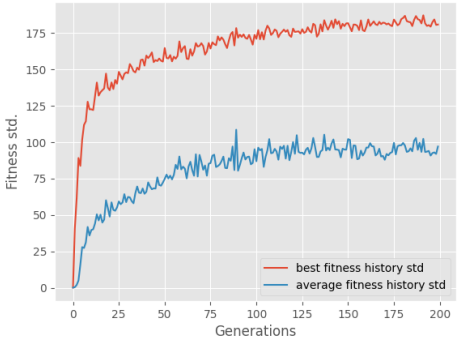
We start with the results of NEAT on Cartpole. The agent learned for 200 generations. As with DQL, we simulated multiple samples (250). We then took the mean as described in subsection 3.5.3. We start with the mean and standard deviation of the fitness values during training of NEAT on Cartpole, shown in figure 4.4a and 4.4b, respectively. The best candidate's fitness increases to almost 200, reaching a value of 50 at 50 episodes already. The corresponding time per generation can be seen in figure 4.4c, and the accumulated time is shown in figure 4.4d. We see that the time per generation increases almost linearly. The accumulated time reaches 500 at the end of the 200 episodes.

As with DQL, we shall not show results for Pong which use intermediary reward systems v1 or v2, presenting only results gathered with systems v0 and v3. We will, however, learn that none of the reward systems had the effect of enabling NEAT to learn Pong.

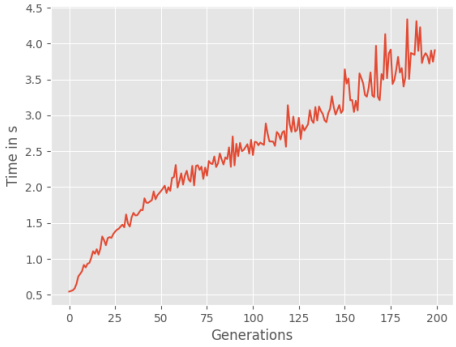
We let the gathering script run for 200 generations of NEAT on Pong. We gathered 47 samples for reward system v0 and 50 samples for reward system v3. All the plots of figure 4.5 depict the average of the mean or standard deviation over all the gathered samples of the respective reward system. As we can see in figures 4.5a and 4.5c, the mean fitness remained almost constant throughout the 200 observed episodes. The average fitness is very low for both reward systems while the best fitness constantly reaches a high value of around 1.8 for system v0 and around 0.9 for system v3. We do, however, see a slight increase in fitness in both graphs during the first 25 episodes.

What follows is the last chapter of this paper, in which we shall discuss the results we have just presented, present the conclusions we have drawn based on those and end with a few words of reflection on our work.

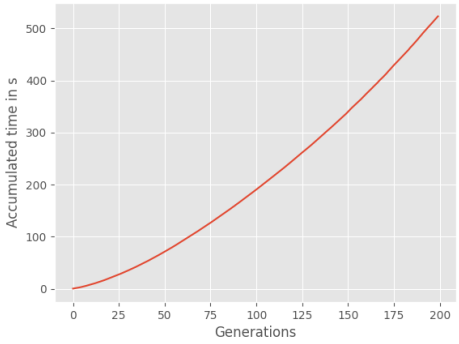

(a) Mean




(b) Standard deviation

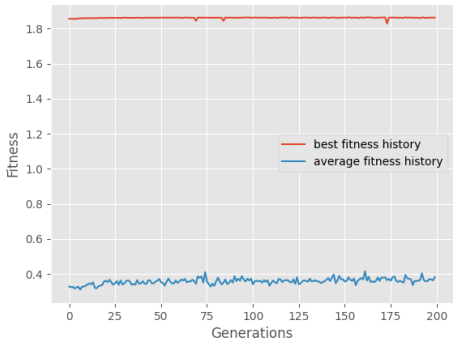



(c) Time

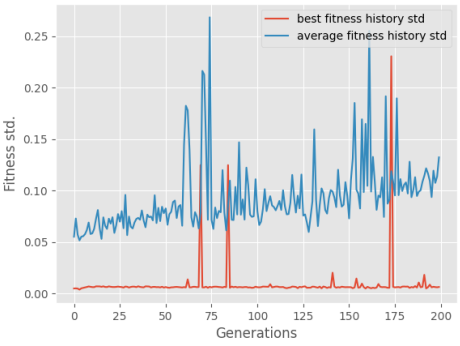



(d) Accumulated time

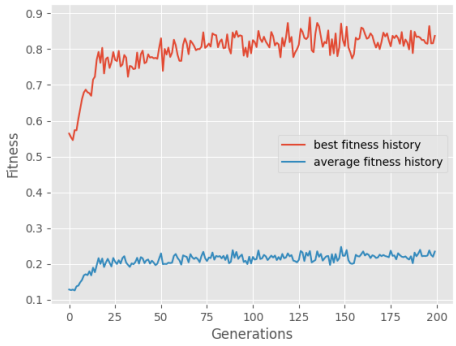
Figure 4.4: NEAT on Cartpole data



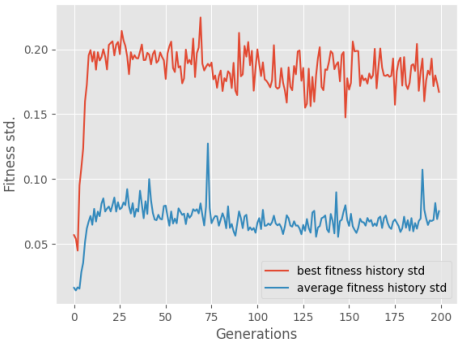

(a) Mean (v0)




(b) Standard deviation (v0)




(c) Mean (v3)




(d) Standard deviation (v3)

Figure 4.5: NEAT on Pong data

# 5 Discussion

In chapter 1 we hypothesised that DQL might outperform NEAT due to its being younger as well as its astounding performance in the original DQL publication, where it solved games given only the raw pixel data. However, as M. Andersson's recent paper [1] concluded that NEAT outperforms DQL on certain tasks, we could not be sure of our hypothesis.

In this chapter, we will clear up the uncertainty by evaluating and comparing the performances of our implementations of NEAT and DQL at the tasks of solving Cartpole and Pong.

In addition to that we will reflect on the implementation process, our programming workflow and the general development of the project.

It is important to keep in mind that our implementations are by no means state-of-the-art. Therefore, we will not draw conclusions about the algorithms themselves, but only about our specific ex-nihilo implementations of the same.

In a further step, our implementations might be compared to model implementations. We will touch on suggestions for further research and possible extensions of the work done in this paper in subsection 5.5.1.

## 5.1 Stochastic Gradient Descent

As we have seen in section 4.1, the average loss decreases to almost 0.0 after 2000 learning epochs. The decrease is very stable, not showing any spikes of significant size. The standard deviation also decreases to almost 0.0. The sample curve shown in figure 4.1c also resembles the target function closely. However, a spike at the end of the graph is visible, which shows that the approximation is not perfect. Nonetheless, we expect the error to decrease further if the network is trained for a longer time. Together, this shows the SGD algorithm to perform extremely well and the implementation to be successful.

## 5.2 Deep Q-Learning

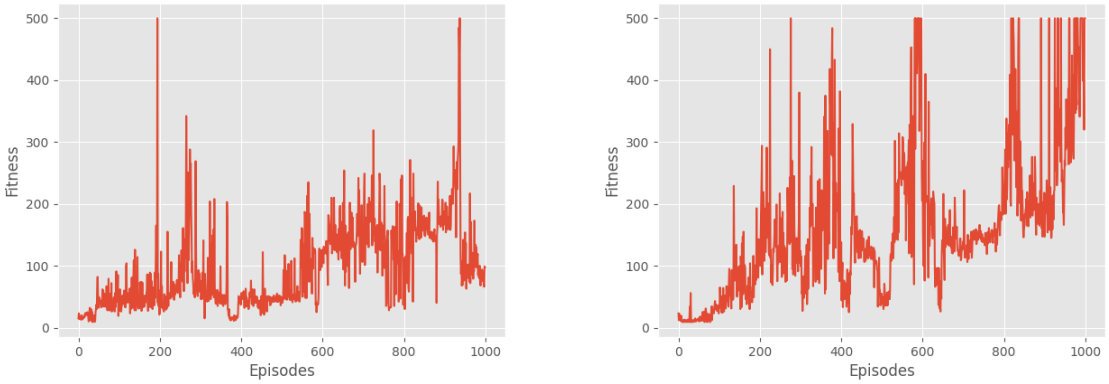
The discussion in this section is based on the results presented in section 4.2. In said section, we showed results gathered while training the DQL algorithm as graphs relating fitness (as an average over multiple learning samples) on the $y$-axis to the current episode on the $x$-axis. As fitness is implemented to increases if and only if the algorithm's performance improves, we

can deduce the learning performance from said graphs. Additionally, we have a graph showing the average time in seconds ($y$-axis) each episode ($x$-axis) took to complete, along with it's antiderivative. These graphs are useful to know how much time DQL took to reach a certain episode.

### 5.2.1 Cartpole

We have seen in section 4.2 that the mean fitness for Cartpole increases with time. The rise is not linear, slowing down over time and not increasing above 190. Remember that, for Cartpole, fitness equals the number of steps before the pole fell (see equation 3.2) and the episode is terminated. As such, a higher fitness directly translates to a better performance.

The standard deviation increases almost as steeply as the mean. This may find its explanation in the fact that the performance of the algorithm is very fragile. This means that, while the performance trend increases, there are always episodes in which a very low performance is observed, i.e. where the pole falls down very quickly. To illustrate this we present two randomly chosen samples in figures 5.1a and 5.1b. Those samples are taken from the data used to calculate figure 4.2a. We see that, while the overall fitness goes up, there are always episodes with very low fitness and the performance is not constant. This explains the high standard deviation.




(a) Sample 1

(b) Sample 2

Figure 5.1: DQL on Cartpole: two samples

All the same, the DQL agent did, on average, improve over time and we can therefore say that the DQL implementation was successful.

### 5.2.2 Pong

As mentioned before, only one reward system yielded desirable results for DQL on Pong. That system is v3.

From figure 4.3a it is evident that DQL was not able to learn Pong with reward system v0. The fitness values seem very random and look like noise around some mean value which is flat over time and below 1. This means that, on average, the agent loses more point than it gains.

With system v3, on the other hand, we have seen an increase in the mean fitness. This increase is rather small, yet it is significant, as it shows that DQL started to learn Pong. Given that the fitness has not started to stabilize until episode 200, it might even increase further if it is given more time. Again, we see that the standard deviation is rather high, which we put down to the same reason as with Cartpole.

We shall give possible explanations as to why the reward systems v0 to v2 were unsuccessful. First, the original system might distribute rewards too sparsely. We believe that it takes too much time for the agent to receive a reward after having performed an action. After all, the time difference is at least the time it takes for the ball to travel across the screen. With this large time difference, the agent cannot tell which action exactly was the cause for the reward he received. In contrast, system v3 immediately tells the agent whether or not his action was desirable. Although system v1 tries to solve this problem by giving more frequent rewards, it fundamentally suffers from the same defect, with its rewards still being rather sparse. Although system v2 distributes regular rewards, it does not differentiate adequately between "good" and "bad" actions because the agent is rewarded continually. Awarding negative rewards suffers from the same defects as systems v0 and v1.

As mentioned in subsection 3.4.1, the fourth and working reward system (v3) might be considered a bit of a "cheat". At the very beginning of chapter 2, we took learning to mean "(...) finding a solution to a given problem within a very specific framework without being told in advance exactly how to do so." By not telling the agent to solve the problem Pong (which would mean telling him to score as many points as possible, as it is done in system v0), but to always keep the paddle on the same height as the ball, this definition is not met. However, this is not a serious problem. The agent still managed to learn how to move the paddle correctly given only the observation parameters. Therefore, the agent definitely did learn something, even though it was not purely the problem Pong, but a subproblem derived from it as defined by reward system v3.

## 5.3 NeuroEvolution of Augmenting Topologies

In section 4.3, we presented results gathered about NEAT's performance as graphs showing the average fitness per generation as well as the average time in seconds each generation took. Notice that the only difference to the graphs shown in section 4.2 is that for NEAT, the $x$-axis is generations, which is in some ways NEAT's equivalent to DQL episodes. We now proceed to discuss these results before comparing the two algorithms' performances in the next section.

### 5.3.1 Cartpole

As can be seen in figure 4.4a, the fitness trend of the best candidate as well as the average fitness trend ascend over time. From equation 3.2 we know that the fitness is equivalent to the amount of time steps the agent was able to balance the pole. An ascending fitness over times thus implies an improvement of the agent's performance over time and we can conclude from this that we implemented a working NEAT algorithm. However, the ascending decelerates throughout the generations and settles at a fitness of around 180 and 90 for the best and average fitness respectively.

The standard deviation is rather high (see figure 4.4b), though it does not exceed 200 for the best candidate and stabilizes at around 100 for the candidates' average. It does not seem to grow much more afterwards, which implies a more stable performance in the long run. An explanation for this is that the fitness is bounded by 0 as well as by the maximal time steps the simulation is run, which is 500 in the gym Cartpole version we used (CartPole-v1). These boundaries eliminate the possibility of larger deviations, which means that the standard deviation cannot grow past a certain limit. Nonetheless, the NEAT algorithm is rather inconsistent in its performance.

### 5.3.2 Pong

Alas, NEAT did not manage to learn Pong, and none of the reward systems showed satisfactory results.

We start by looking at reward system v0. We can see in figure 4.5a that the best fitness trend is always above 1.8 while the average fitness trend does not exceed the 0.4 mark and is flat the whole time. As we have a rather big population of about 150 individuals, it is nearly impossible that there is a case in which not one candidate hits the ball by chance and then ends up scoring a point against the stationary opponent. This explains the comparatively high mean for the best candidate throughout the learning process. The value of 1.8 is to be expected, as the upper bound for the fitness is $2 = 1 + e^0$ (see equation 3.3). This upper bound along with there always being a candidate which scored a point by chance also explains the low standard deviation seen in figure 4.5b. Evidently, the high average fitness of the best candidate does not mean that the NEAT algorithm managed to learn Pong, as this value is obtained by chance, while the average fitness over all candidates remains low. Therefore, most of the candidates have a fitness value below 1 which means that they let the ball fly off the screen.

We can see in figure 4.5c that there is no increase in the performance of NEAT on Pong even when using v3 instead of v0. Same as with v0 we have a rather high performance of the best and a low performance of the worst candidates which can be explained by a similar reasoning as above.

There is one small sliver of hope, namely the slight increase in the mean fitness between

iterations 0 to 25 in figure 4.5c. Although this may only be statistical noise, it is possible that, given hyperparameters better than the ones we found, the fitness would increase further.

As it is, however, it is safe to say that NEAT on Pong was unsuccessful. The reasons therefor are in the end unclear to us.

## 5.4 Showdown - DQL versus NEAT

Having seen that both algorithms do, in principle, work, be it not on all the problems that we tested them on, we can now proceed to compare the two algorithms' performances.

As NEAT's generations are not related to DQL's episodes in any way, we cannot compare the graphs depicting the mean fitness of the two algorithms directly. However, the performance comparison is possible nonetheless. First, because we used the same definitions for the fitness functions of NEAT and DQL, we have the same $y$-axis for both algorithms' fitness graphs. Second, we can relate the $x$-axes by help of the graphs relating time in seconds to both DQL episodes and NEAT generations.

From figures 4.2a and 4.4a we can conclude that DQL did not outperform NEAT in Cartpole, as the fitness value at the end of training is about 180 to 190 for both algorithms. Even so, it took DQL more than 600 episodes to stabilize at a fitness of 180. On the other hand, NEAT's best candidate reached the same fitness level and started to stabilize by the 75th generation. While we cannot directly compare generations to episodes, we can compare the time it took the algorithms to reach their final fitness level. For that we take a look at the sum over the single time values it took to reach the 600th episode for DQL and the 75th generation for NEAT (see figures 4.2d and 4.4d). DQL needed about 2100 seconds while NEAT only took about 500 seconds to reach a fitness of 180. Thus, we can say that NEAT learns Cartpole about four times faster than DQL. Therefore, in this aspect, NEAT outperformed DQL, confirming the finding in [1, p. iii].

However, as NEAT was not able to learn Pong, DQL did outperform NEAT on Pong. This shows that DQL has a central advantage over NEAT too, as we have shown it to be able to learn a more complex problem than Cartpole. We are, however, convinced that NEAT can in theory also learn the game Pong. Nonetheless, it seems that it is easier to have DQL learn Pong than to have NEAT do the same.

## 5.5 Conclusions

Let us recollect the original goals we set ourselves. We set out to implement DQL and NEAT. As we have seen, this has been done successfully. We then wanted the implemented algorithms to learn Pong. This might be called a fifty-percent success, as NEAT did not learn Pong. However, we also had both algorithms learn the game Cartpole, which proves that they are both able to learn something.

Last, we wanted to compare the algorithms' performances in their learning speed and their effectiveness at the end of training. We have done this in the previous sections. We concluded that, for Cartpole, NEAT learns faster than DQL, but that their final effectiveness is the same, and that NEAT did not learn Pong, DQL thus outperforming it in the game.

The project can hereby be called completed. Of course, this leaves open the possibility of doing further work on the basis of the paper. The product of our project in the form of a code base may be improved and extended with additional functionality. We want to give some ideas therefor.

### 5.5.1 Further Research

Perhaps the most obvious next step is to let the two algorithms compete, say, in a Pong duel. Of course, this requires first to achieve desirable learning results with NEAT on Pong.

Additionally, our implementations for both DQL and NEAT might be compared to various model implementations of the algorithms. For DQL the most prevalent one is the original implementation used in publication [10] and for NEAT it is K. Stanley's original implementation in C++.

Next, we expect the two algorithms to be able to learn a plethora of other games, e.g. Snake, Pool, Pong 3D, perhaps even Chess. All of those games could be tested and evaluated.

Last, one might implement a system for automatically finding the optimal hyperparameters for every algorithm, something which might be done with yet another machine learning algorithm.

Of course, the above list is not exhaustive, and many more possibilities are to be explored.

### 5.5.2 Personal Conclusion

Except for bugs in our programs during development, which are to be expected, we ran into a low number of problems while writing this paper. This includes technical problems as well as personal disagreements, of which we did have the usual few, but which we could resolve sensibly. It hence seems that after having tried out different methods in the past, we found tools and a workflow that suit us rather well and allow us to work together efficiently.

We are herewith at the end of our matura paper, leaving the reader only with a few closing words, along with the bibliography, list of figures and the dullness of the paper's back cover, should the reader even have a physical copy of the paper. This is, however, no reason to be disappointed, for although, having written these fifty pages over the last months, we hope there are but a few things left unsaid, there will always remain things left undone. Perhaps the reader can implement one of the suggested improvements on the algorithms or improve parts of the code base. It is, after all, not as an annoyance and much rather as encouragement for further improvements of our knowledge and skills that we view what remains imperfect.

# Bibliography


[1] Marcus Andersson. "How does the performance of NEAT compare to Reinforcement Learning?" MA thesis. KTH Royal Institute of Technology, 2022.

[2] Greg Brockman et al. *OpenAI Gym (cartpole.py)*. 2016. URL: `https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py`. (accessed: 03.10.2022).

[3] Steven L. Brunton and J. Nathan Kutz. *Data Driven Science & Engineering*. Cambridge University Press, 2021, pp. 500–700.

[4] Leana Cadinu, Luis Hartmann, and Fabio Panduri. *Drei praktische Anwendungen der Schwingungslehre*. 2021. Projektarbeit Alte Kantosschule Aarau.

[5] Charles Darwin. *On the Origin of Species*. fifth edition. 1869.

[6] Lex Friedman. *Introduction to Deep Reinforcement Learning*. 2019. URL: `https://www.dropbox.com/s/wekmlv45omd266o/deep_rl_intro.pdf`. (Lecture slides).

[7] Lex Friedman. *MIT 6.S091: Introduction to Deep Reinforcement Learning (Deep RL)*. 2019. URL: `https://youtu.be/zR11FLZ-O9M`. (accessed: 08.05.2022).

[8] Keon (@keon) Kim. *deep-q-learning*. 2017. URL: `https://github.com/keon/deep-q-learning`. (accessed: 29.08.2022).

[9] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Third Printing. Complex Adaptive Systems. The MIT Press, 1998. ISBN: 978-0-262-63185-3.

[10] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533. DOI: `doi:10.1038/nature14236`.

[11] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[12] Oxford University Press. *Oxford Learner's Dictionaries*. 2022. URL: `https://www.oxfordlearnersdictionaries.com/definition/english/machine-learning`. (accessed: 18.09.2022).

[13] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary Computation* 10.2 (2002), pp. 99–127.

[14] Nassim Nicholas Taleb. *Antifragile*. Random House, 2012.


Cover picture: "A neat Deep Neural Network", by Fabio Panduri

# List of Figures