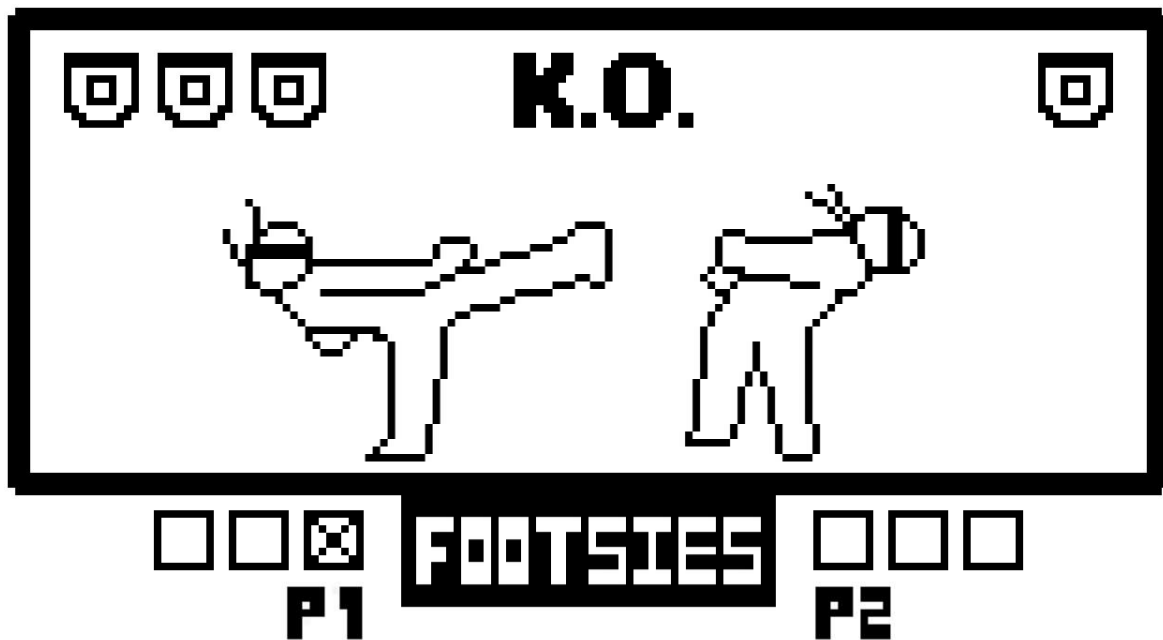


# READY PLAYER 21

Integration of Artificial Neural Networks in Multiplayer Games.



Leon Rossi W21d

17.12.2024

Matura Thesis

Kantonsschule Enge, Zürich

Patrik Marxer



## Table of Contents

1	Introduction .....	5
1.1	Preface.....	5
1.2	Personal motivation .....	5
1.3	Objective and Mission Statement .....	6
1.4	Executive Summary .....	7
2	Technical Background: Functionality of Artificial Neural Networks .....	8
2.1	.....	9
2.2	Architecture of an Artificial Neural Network .....	10
2.2.1	Perceptrons.....	10
2.2.2	Activation functions:.....	10
2.2.3	Layers .....	12
2.2.4	My Implementation of a basic neural Network.....	14
2.3	Training of Artificial Neural Networks .....	16
2.3.1	Genetic Algorithms .....	16
2.3.2	Gradient Descent .....	17
2.3.3	How I derive the partial Derivatives .....	23
2.3.4	Reinforcement Learning and Actor-Critic Method.....	24
3	Implementation of ANNs into Pong and Footsies .....	31
3.1	Background on Programming .....	31
3.2	Game Logic and ANN Implementation Objectives .....	32
3.2.1	Pong.....	32
3.2.2	Footsies .....	33
3.3	Approach 1: Genetic Algorithm.....	34
3.3.1	Pong.....	34
3.3.2	Footsies .....	35
3.4	Approach 2: Reinforcement learning.....	37
3.4.1	Pong.....	37
3.4.2	Footsies .....	38
3.5	Comparison of the Approaches .....	39
4	Exploring parameters.....	43
4.1	Method.....	43

4.2	Parameters.....	44
4.3	Hypotheses .....	44
4.3.1	Size of the ANNs .....	44
4.3.2	Learning Rate in Reinforcement Learning.....	44
4.4	Results .....	45
4.4.1	Baseline .....	45
4.4.2	Size of ANNs.....	48
4.4.3	Learning Rate in Reinforcement Learning.....	52
5	Conclusion .....	56
6	Acknowledgements.....	58
7	Declaration of Authenticity .....	58
8	Reflection .....	59
9	Bibliography .....	60
10	Appendix.....	63
10.1	Appendix A .....	63
10.2	Appendix B .....	63
10.3	Appendix C.....	63
10.4	Appendix D.....	63
10.5	Appendix E .....	64
10.6	Appendix F .....	64
10.7	Appendix G.....	64

## Table of Figures

Title page figure: HiFight. “FOOTSIES - HiFight,” 2018. <https://hifight.github.io/footsies/>.

Figure 1: Model of a perceptron, which takes $X_1$ to $X_n$ and $b$ as inputs and returns $Z$ .	10
Figure 2: Mathematical definition of a step function with zero as the threshold.	11
Figure 3: Graph of the Sigmoid function.	11
Figure 4: Graph of the Rectified linear function.	12
Figure 5: Architecture of an ANN. Each node represents a perceptron, and each edge represents a connection between two perceptrons with an assigned weight. The input layer is coloured blue, the hidden layers are coloured black, and the output layer is coloured green.	13
Figure 6: Simplified version of RunNN function.	15
Figure 7: Mathematical definition of the mean squared error function.	17
Figure 8: A graph showing how a weight is shifted down the gradient.	18
Figure 9: Model of a neuron with the weights $W_1, W_2, W_3$ , the inputs $X_1, X_2, X_3$ and the output $y$ . The bias is not in the graphic but would be added.	20
Figure 10: Model of a neuron with the weights $W_1, W_2, W_3$ , the inputs $X_1, X_2, X_3$ and the output $y$ . The bias is not in the graphic but would be added.	23
Figure 11: The agent-environment interaction in a Markov decision process.	25
Figure 12: A state-action value function with four states and four actions, Right, Left, Up and Down. For each state except state 3, the action with the highest expected total reward is marked green. In state 3 a random action is selected to ensure exploration.	26
Figure 13: Graph describing the relationship between the policy function as the actor, the value function as the critic and the environment.	29
Figure 14: Pseudocode for the Actor-Critic Method. $\delta$ is the advantage, $R$ is the reward, $\nabla \ln \pi(A S, \theta) = \nabla \pi(A S, \theta) / \pi(A S, \theta)$ .	30
Figure 15: Screenshot of Pong. The white rectangles on the side are the paddles controlled by the player and the white square close to the middle is the ball.	32
Figure 16: Screenshot of Footsies. In the top corners the shields represent how often you can block. The rectangles at the bottom indicate the current score in a best of five.	33
Figure 17: Performance of the reinforcement learning algorithm with the following parameters:	39
Figure 18: Performance of the Genetic Algorithm in relation to the generation. Here the Genetic Algorithm had a population size of 200, 10 lives and a mutation threshold of 0.1.	41
Figure 19: Performance of multiple learning attempts with the same parameters.	45
Figure 20: Two ways the algorithm can get stuck and stop converging. A learns at the start and then suddenly falls back to 0.25, B never improves. Both use the parameters described in Table 1.	47
Figure 21: Three learning attempts that start with the same set of weights instead of each one starting with a unique set of randomly generated weights. This shows the effect	

variables other than initialization have on the training. Parameters as specified in Table 1.	48
Figure 22: Average performance of multiple learning attempts with the same-sized ANNs. The name of each data set is x,y. X is the number of hidden layers and y is the number of neurons in each hidden layer, the rest of the parameters are the same as before. Only successful learning attempts are represented, unsuccessful attempts were discarded.	49
Figure 23: The number of episodes [in thousands] it took each size in Figure 22 to reach 0.99 accuracy versus the total number of neurons in all hidden layers.	50
Figure 24: The chance of success for each size in Figure 22 versus the total number of neurons in the hidden layers.	52
Figure 25: Performance of learning attempts with different learning rates for the actor and the critic. The name of each data set is "x, x is the learning rate of the critic. The learning rate of the actor is smaller by a factor of ten, so x/10.	53
Figure 26: Zoomed in version of Figure 25 for readability.	54
Figure 27: The number of episodes [in thousands] it took each learning attempt in Figure 25 to reach 0.99 accuracy versus learning rate. 0.0001 was left out because it made the graph unreadable.	54

## Table of Tables

Table 1: The parameters used for all the learning attempts in Figure 19.	45
Table 2: Analysis of points where the policy reaches an accuracy of 0.99. Same data set as Figure 19.	46
Table 3: Kendall's correlations between the total number of neurons and the number of episodes [in thousands] needed to reach 0.99 accuracy and between the total number of neurons and the chance of success.	50
Table 4: The parameters used for all the learning attempts in Figure 25.	53

# 1 Introduction

## 1.1 Preface

In recent years, artificial intelligence has taken the world by storm. Products like Chat GPT and Dall-E have changed how people work, and many believe that artificial intelligence is the next big innovation since the industrial revolution. But the field of artificial intelligence and machine learning has been around for a lot longer than eye-catching applications like large language models.

While many projects have practical applications, other projects are purely for research and need environments to serve as testing grounds. It is oftentimes not feasible to build real-life test environments, so simulations are used to substitute them. Oftentimes video games have been used as playgrounds for testing machine learning research since they do not require specific hardware and changes to the algorithm and the environment can be made on the fly.

The first games used in machine learning were classical board games like chess and go, but the variety of available video games offers a wide range of challenges that are also found in the real world. These video games can be used to solve problems before building expensive hardware to develop experimental software in the real world.

## 1.2 Personal motivation

My interest in computer science started with playing video games. From playing on Nintendo consoles, I switched to Windows, where I became interested in not only playing video games but also creating them. Using Unity, I started experimenting with basic character movement, combat, and pathfinding. While none of my projects were ever even close to being finished, I enjoyed mainly the programming aspect of the process.

When I went on a high school exchange year, I used this chance to attend a computer science course offered by my school in Wales (UK) and learned the proper basics I missed due to me following random tutorials and solving problems based on Google and Stackoverflow. While game design did not end up sticking, my interest in computer science and programming did. Due to this, I chose to do a programming project as my “Maturitätsarbeit”.

My first contact with machine learning was also through video games. In StarCraft II, a highly complex game I was playing at the time, AlphaStar by Google DeepMind was beating some of the best players internationally. This really impressed me, as it was able to master a game I was struggling with immensely. Algorithms that can perform such complicated tasks still fascinate me today.

## 1.3 Objective and Mission Statement

My goal for this project was to understand, and to a certain extent recreate, how it is possible for an algorithm to achieve such a high level of proficiency in such a complex task.

For this I developed the code for an artificial neural network that has the capability to learn games, i.e. the respective game logic, and is able to prevail in playing these games against human capabilities. Developing AI algorithms that can teach themselves to efficiently master standard market games (such as StarCraft II, Dota II and others) is highly complex and requires substantial time and technical resources. Moreover, these games have been developed for commercial purposes and neither their code nor the respective API's are available to the public in a way that AI integration would be possible. Out of these reasons, I decided to use open-source games with simple game mechanics for the purpose of this work. Thereby, my choice fell on two games that are simpler and open-source:

- **Pong:** A well-known 2D imitation of table tennis, where the goal is to hit a ball past your opponent. It's a very simple game that involves minimal engagement with an opponent and a very small number of possible actions.<sup>1</sup>
- **Footsies:** an open-source fighting game that tries to simplify the concept of games like Street Fighter down to a minimum. This makes it a lot easier to learn than a AAA fighting game with dozens of characters and hundreds of possible moves. *Footsies* poses a lot more challenges compared to *Pong* since it requires active engagement with the opponent and is, in general, way more complex.<sup>2</sup>

From a technical perspective, I worked with two different learning algorithms to compare the result of their application to the games mentioned above, a Genetic Algorithm and the Actor-Critic Method:

- **Genetic Algorithm:** A quite simple method that relies on randomness and evolution based on Charles Darwin's theories to find a solution to a given task.<sup>3</sup>
- **Actor-Critic Method:** A deep reinforcement learning algorithm which uses a combination of two Artificial Neural Networks, rewards and backpropagation to solve a given task. I expected the Actor-Critic Method to outperform the Genetic Algorithm since it is quite a lot more advanced and complicated.<sup>4</sup>

---

<sup>1</sup> See section 3.2.1

<sup>2</sup> See section 3.2.2

<sup>3</sup> See section 2.2.1

<sup>4</sup> See section 2.2.4



## 1.4 Executive Summary

**Result of applied Algorithms:** In summary, the evaluation of the performance of the applied algorithms provided the following results:

- **Genetic Algorithm:** The Genetic Algorithm provided good results for both *Pong* and *Footsies*. Especially for *Pong* the performance was better than what I expected. It achieved near perfect accuracy and rivalled the performance of the Actor-Critic Method. However, the Genetic Algorithm demonstrated an advantage in handling steep ball angles due to its lack of reliance on action probabilities. For *Footsies*, the algorithm achieved moderate success, learning basic strategies but struggling with the evolving nature of self-play. The performance was therefore far from optimal, and it had problems learning anything complex and reacting to the enemy.
- **Actor-Critic Algorithm:** The Actor-Critic Method worked well for *Pong*, especially after the optimization of the parameters it was able to learn fast and achieve high accuracy quite consistently. The implementation for *Footsies* had more problems, the self-play mechanism it had originally used did not work as hoped. Modifications and training against a pre-programmed bot resulted in improvements, but it maintained to have problems with consistency and was only able to achieve a limited level of performance.

Assessment of parameter impact:

- **Size of ANNs:** My tests in Section 4 prove that my assumptions about how the size of the artificial neural networks affects the learning process were wrong. Contrary to my initial assumption, it is possible to solve *Pong* with artificial neural networks of minimal complexity, and as the complexity of the artificial neural networks increases, the algorithm can learn faster.
- **Learning Rate of ANN:** I discovered that the learning rate, controlling the magnitude of updates during training, has a significant impact on the learning process of the ANN. Optimal learning rates were identified for the Actor-Critic Algorithm in *Pong*, demonstrating the sensitivity of performance to this parameter.

**Overall Assessment:** As part of my work, I successfully developed AI systems capable of mastering *Pong*, demonstrating that it is possible to apply Artificial Neural Networks of minimal complexity to such a use case and still achieve almost 100% accuracy. While the algorithms for *Footsies* showed promise, further research and optimisation would be required to achieve human-level performance.

## 2 Technical Background: Functionality of Artificial Neural Networks

At the core of all the learning algorithms I used in this paper are feed-forward artificial neural networks. I will refer to them as “Artificial Neural Networks”, or for short, “ANNs”.

---

*“Deep feedforward networks, also called feedforward neural networks, or multilayer perceptron networks (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function  $f^*$ . For example, for a classifier,  $y = f^*(x)$  maps an input  $x$  to a category  $y$ . A feedforward network defines a mapping  $y = f(x; \theta)$  and learns the value of the parameters  $\theta$  that result in the best function approximation.”<sup>5</sup>*

---

In the following chapter, I will explain how these artificial neural networks work and how they can be used to complete complex tasks.

The first deep MLPs, like what I am using, were developed in 1965 by researchers in the USSR and were based on ideas that can be dated back to 1795, where Johann Carl Friedrich Gauss developed the first linear neural networks.<sup>6</sup>

Artificial neural networks are based on the inner workings of a human brain. The brain consists of many connected neurons, these neurons communicate using electrical signals and can interpret signals and control the body. Artificial neural networks aim to mimic these functions to complete complex tasks.<sup>7</sup> Thereby, in essence, ANNs compute complex data relationships and patterns by transforming the weighted sum of multiple input data via interconnected nodes, so called neurons (perceptron), through multiple layers in a non-linear manner (using a so-called activation function) to relevant output data. While the key elements of ANNs are further explained below, here an initial overview on how ANNs essentially operate:

---

<sup>5</sup> Goodfellow, Bengio, and Courville, *Deep Learning*.

<sup>6</sup> Schmidhuber, “Annotated History of Modern AI and Deep Learning.”

<sup>7</sup> Walczak and Cerpa, “Artificial Neural Networks.”

- **Input:** The ANN receives input data, which can be anything from sensor readings to pixel values in an image.
- **Weighted Summation:** Each perceptron<sup>8</sup> in a layer receives inputs from the previous layer. Each input is multiplied by a weight associated with that connection. These weighted inputs are then summed up.
- **Activation Function:** This weighted sum is then passed through an **activation function**<sup>9</sup>. The activation function introduces **non-linearity**, enabling the network to model complex, non-linear relationships in the data, a crucial capability for more complex tasks. The output of the activation function becomes the output of the perceptron.
- **Output:** This process is repeated layer by layer, with the outputs of one layer becoming the inputs for the next, until the final layer produces the output of the entire ANN. This output can represent a decision, a prediction, or an action, depending on the task that the ANN has been assigned with.

Initially, the weights and biases of the ANN are randomly assigned and the outcome of the ANN model therefore unreliable. To make meaningful predictions or decisions, the ANN needs to be **trained**. This involves adjusting the weights and biases based on the performance of the network on a given task. The goal is to find the optimal set of weights and biases that minimize errors and maximize the network's ability to perform the desired task (see section x below).

## 2.1

---

<sup>8</sup> See section 2.1.1

<sup>9</sup> See section 2.1.2

## 2.2 Architecture of an Artificial Neural Network

### 2.2.1 Perceptrons

At the heart of the human brain are the neurons. Artificial Neural Networks consist of perceptrons, also called neurons, which simulate the functions of a human neuron.



Figure 1: Model of a perceptron, which takes  $X_1$  to  $X_n$  and  $b$  as inputs and returns  $Z$ .<sup>10</sup>

A perceptron turns multiple inputs into a single output. Each input is usually connected to either the output of another perceptron or to an input into the neural network like a sensor reading or the colour value of a pixel. Each input gets weighted, which means it is multiplied with a weight determined by the perceptron. All the weighted inputs and a bias are added together and get run through an activation function which then returns  $Z$ .<sup>11</sup>

### 2.2.2 Activation functions:

ANNs without activation functions are just a collection of linear functions and as a result will stay linear no matter what the weights and biases are; this makes them unable to complete anything other than simple tasks. The purpose of the activation function is to introduce non-linearity. This enables the ANN to approximate complex functions as required for complex tasks.<sup>12</sup>

The most basic activation function is the step-function. It converts a single real number into a one if it is above a certain threshold. Else it returns a zero.<sup>13</sup>

---

<sup>10</sup> Kinsley and Kukieta, "Neural Networks from Scratch in Python.", 13.

<sup>11</sup> Kinsley and Kukieta.

<sup>12</sup> Brownlee, "How to Choose an Activation Function for Deep Learning - MachineLearningMastery.Com."

<sup>13</sup> Codecademy, "Binary Step Activation Function | Codecademy."

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Figure 2: Mathematical definition of a step function with zero as the threshold. <sup>14</sup>

Step functions are barely used anymore as they are unable to represent data, and newer activation functions like the sigmoid function allow for more accuracy.

The sigmoid function maps all real numbers into a range between zero and one. This allows it to represent more numbers with more accuracy and has been widely used in neural networks over the past few years. The most common sigmoid function is the logistics function as below, but other versions exist. <sup>15</sup>

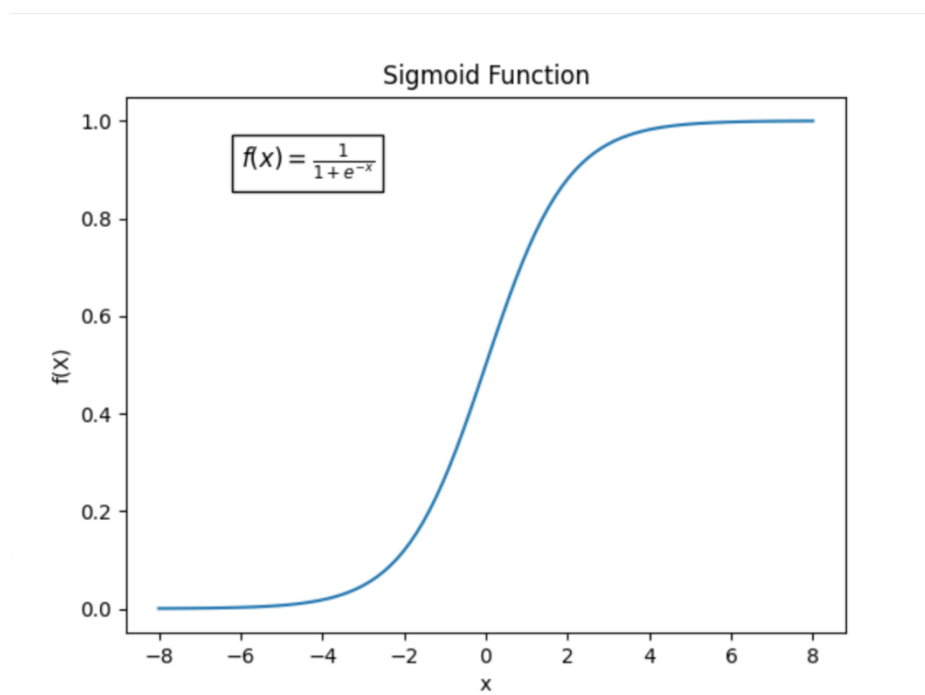


Figure 3: Graph of the Sigmoid function. <sup>16</sup>

The main issue with sigmoid functions is the vanishing gradient problem. As one can see in the graph, the curve flattens out with large positive or negative values. This means the derivative of a sigmoid function is bell-shaped, and the derivatives of large positive or large negative values are close to zero. Over multiple layers, this can lead to very small changes in weights and slow down the learning considerably. <sup>17</sup>

<sup>14</sup> Codecademy.

<sup>15</sup> Wood, "Sigmoid Function Definition | DeepAI."

<sup>16</sup> YanisaHS, "Sigmoid Activation Function | Codecademy."

<sup>17</sup> DeepAI, "Vanishing Gradient Problem Definition | DeepAI."

The rectified linear function (ReLU) solves this problem. It returns zero if the input is smaller than zero, and if not, it returns the input. This means that the derivative is either one or zero.

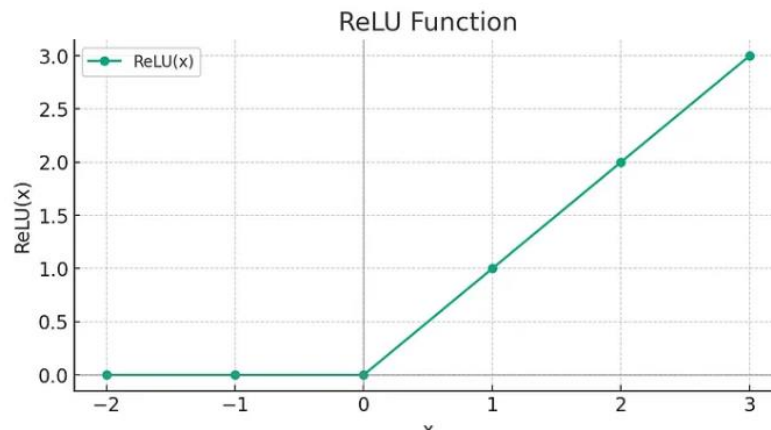


Figure 4: Graph of the Rectified linear function. <sup>18</sup>

It is also faster to compute and easier to derive. Due to it being mostly linear and being able to output a true zero, which is desirable, it is easier to train and leads to faster training. ReLU's main disadvantage is that it is unable to return negative values, but for most applications, its benefits outweigh the loss of negative values. <sup>19</sup>

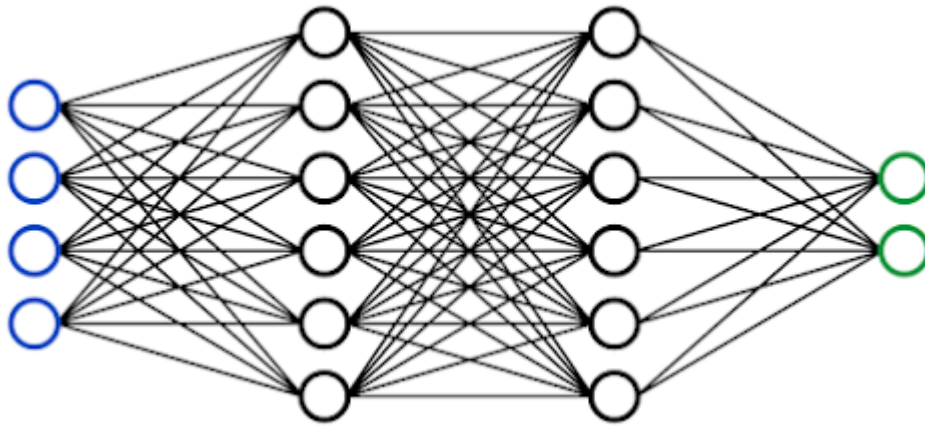
### 2.2.3 Layers

Every perceptron consists of a bias, weights, and an activation function. A layer consists of multiple perceptrons. In a fully connected network, like the one I am using, the inputs of each perceptron in layer  $n$  are connected to all the outputs in layer  $n-1$ . Similarly, the output of each perceptron in layer  $n$  is connected to the inputs of all the perceptrons in layer  $n+1$ .

---

<sup>18</sup> Patel, "Understanding the Rectified Linear Unit (ReLU): A Key Activation Function in Neural Networks | Medium."

<sup>19</sup> Brownlee, "A Gentle Introduction to the Rectified Linear Unit (ReLU) - MachineLearningMastery.Com."



*Figure 5: Architecture of an ANN. Each node represents a perceptron, and each edge represents a connection between two perceptrons with an assigned weight. The input layer is coloured blue, the hidden layers are coloured black, and the output layer is coloured green.<sup>20</sup>*

In this graph, each node represents a perceptron, and each edge represents a connection between two perceptrons with an assigned weight. The input layer is coloured blue, the hidden layers are coloured black, and the output layer is coloured green. The ANN consists of four inputs, two hidden layers with six neurons each, and two outputs.<sup>21</sup>

As you can see in the graph, there are three types of layers: the input layer, the hidden layers and the output layer:

- **Input Layer:** The purpose of the input layer is, as the name suggests, to handle the inputs. Its function is to pass on the value of each of the inputs, which could be the reading of a sensor or the positional data of an object, to the perceptrons of the first hidden layer. It does not have biases or an activation function and instead passes on the data it receives. It serves as an intermediary between the outside and the ANN.
- **Hidden Layer:** Hidden layers are usually the biggest part of the ANNs and are responsible for most of the computation in the neural network. They process the inputs according to their weights, biases, and activation functions, and tuning their parameters is how they are improved. The size of a neural network is determined by the number of hidden layers and the amount of perceptrons in each layer.

---

<sup>20</sup> Jander, "Programming a Mediocre Neural Network From Scratch."

<sup>21</sup> Jander.

- **Output Layer:** The output layer is the last layer and produces outputs that are usable as a solution to the given task. The number of outputs is determined by the task. The output layer oftentimes does not use the same activation function as the hidden layers but instead either a function like SoftMax, which normalises the values so that they add to one, or outputs the raw data. The SoftMax function is most often used to create a probability distribution for classification tasks. Classification tasks label input data with a class label. An example would be labelling as “spam” or “not spam”. The postprocessing the output layer does is needed to ensure that the outputs of the ANN are usable.<sup>22</sup>

## 2.2.4 My Implementation of a basic neural Network

The artificial neural network lies at the heart of all the algorithms I use for my project, and I had to make sure that my implementation is efficient enough to be used and trained at an at least acceptable pace. The way my implementation works is by using a four-dimensional list. That is a list of lists of lists of lists of lists. The first list contains all the data needed to form the complete artificial neural network and consists of lists that each represent a different layer.

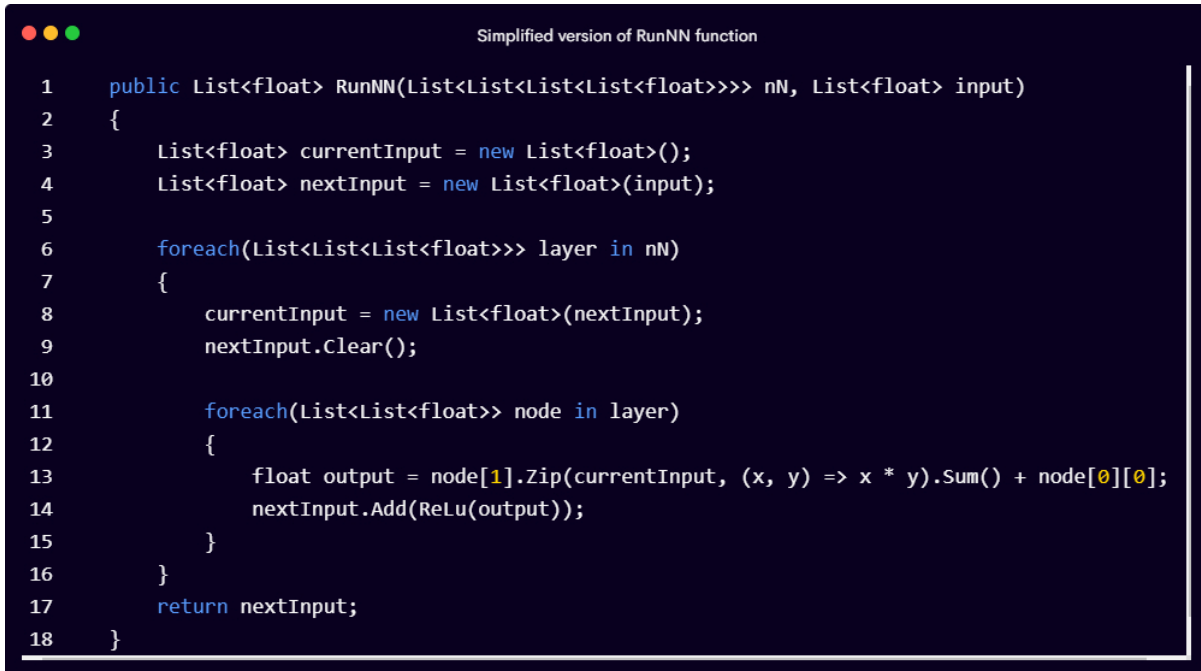
All layers, except for the last one, contain the same number of lists, each list representing a different neuron. In each neuron there is one list that contains all the weights between the input of this neuron and the output of all the neurons in the layer before, and another list that contains the bias and some information used by the training algorithm.

In this construct, the input layer does not have its own list. As it does not have an activation function or weights, this is not necessary. Instead, the number of weights in the first hidden layer corresponds to the number of inputs, so that they can be directly passed into the weights of the first hidden layer.

---

<sup>22</sup> GeeksForGeeks, “Layers in Artificial Neural Networks (ANN) - GeeksforGeeks”; Brownlee, “4 Types of Classification Tasks in Machine Learning - MachineLearningMastery.Com”; Kinsley and Kukieta, “Neural Networks from Scratch in Python.”, 13-14.





```

1  public List<float> RunNN(List<List<List<List<float>>>> nN, List<float> input)
2  {
3      List<float> currentInput = new List<float>();
4      List<float> nextInput = new List<float>(input);
5
6      foreach(List<List<List<float>>> layer in nN)
7      {
8          currentInput = new List<float>(nextInput);
9          nextInput.Clear();
10
11         foreach(List<List<float>>> node in layer)
12         {
13             float output = node[1].Zip(currentInput, (x, y) => x * y).Sum() + node[0][0];
14             nextInput.Add(ReLu(output));
15         }
16     }
17     return nextInput;
18 }

```

Figure 6: Simplified version of RunNN function.

The code in Figure 6 is responsible for calculating the output of a ANN based on a list of inputs. The function takes in two arguments, the ANN (nN), the list with all the weights and biases, and a list of inputs. It outputs one list with all the outputs. The inputs, weights, biases and outputs are all floating-point numbers.

Line 13-14 calculate the output of a single neuron. The Zip function in Line 13 multiplies each input from the currentInput list with the corresponding weight in node[1] and adds the bias in node[0][0]. In Line 14 this value is passed through the Rectified Linear function, ReLu(), and added to the end of the list that holds the inputs for the next layer.

This is repeated for every neuron in the layer, afterwards the currentInput list copies what is in the nextInput list and the nextInput list is cleared. This way the currentInput list contains the outputs of the last layer and can give the to this layer. After the last layer, the nextInput list contains the outputs of this last layer and is returned as the output of the function.

## 2.3 Training of Artificial Neural Networks

When creating an artificial neural network all the weights and biases are initialised randomly or semi-randomly. At the start the outputs are also random and complete nonsense. For them to become meaningful the ANN must first be trained by tuning the parameters. This means we try to find values for all the weights and biases which lead to the outputs representing viable solutions to our task, based on the inputs.<sup>23</sup>

As part of this work, the following training algorithms have been applied:

- **Genetic Algorithm:** One of the simplest training algorithms for neural networks and other machine learning algorithms (see below section 2.2.1).
- **Actor-Critic Method;** A particular type of reinforcement learning, which uses the backpropagation from the Gradient Descent algorithm (see below section 2.2.2).

### 2.3.1 Genetic Algorithms

Genetic Algorithms are among the simplest training algorithms for neural networks and other machine learning algorithms. They are based on natural selection as proposed by Charles Darwin and try to imitate the way genes are inherited and mutated in nature and use this to optimize algorithms.<sup>24</sup>

Genetic Algorithms start by initializing multiple individuals, in this case ANNs with different weights and biases. Each unique combination of weights and biases is called a chromosome. Each variable in these chromosomes, here each weight and each bias, is a gene. The goal of the Genetic Algorithm is to find the optimal value for each gene to combine them into a chromosome which can perfectly solve a given task. The collection of individuals forms a population.<sup>25</sup>

Each individual is assigned a fitness value. The purpose of this fitness value is to describe the chromosome's ability to solve the given task. The fitness value is determined by testing all the chromosomes, and the better they perform, the fitter they are and the larger their assigned fitness value is.

Then the next generation, another population, is created based on the achieved fitness values. There are a lot of ways to do the reproduction, but usually, for each individual in the new population, two "parents" are semi-randomly selected, with the ones with a higher fitness value having better chances at being selected. The new chromosome is

---

<sup>23</sup> Kinsley and Kukiela, "Neural Networks from Scratch in Python",131f; Walczak and Cerpa, "Artificial Neural Networks."

<sup>24</sup> AnalytixLabs, "A Complete Guide to Genetic Algorithm — Advantages, Limitations & More | Medium"; Tattersall, "Charles Darwin and Human Evolution."

<sup>25</sup> AnalytixLabs, "A Complete Guide to Genetic Algorithm — Advantages, Limitations & More | Medium."

then a mixture between the genomes of the two parents with a chance of mutation, a random change of a gene.

This way, the best-performing individuals get preserved and mixed with the hope of creating individuals that perform the same or better. This leads to a steady improvement with each generation until either the limit or a dead end is reached.<sup>26</sup>

### 2.3.2 Gradient Descent

Gradient Descent is one of the most used training algorithms and uses pre-classified data to train. Each data set consists of a set of inputs and a correct solution. Gradient Descent is an analytical approach to training a neural network.

I used a different algorithm since I do not have pre-classified data. The Actor-Critic Method uses gradient ascent, which is very close to Gradient Descent, so many of these concepts still apply and it is easier to understand the mathematical concepts behind backpropagation based on Gradient Descent than the Actor-Critic Method.<sup>27</sup>

In Gradient Descent, a loss function is used to determine how well a given set of weights and biases solves a problem.

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

*Figure 7: Mathematical definition of the mean squared error function.*<sup>28</sup>

This is done by comparing the output vector of the neural network and the “perfect” answer, which is given by the pre-classified dataset. The mean squared error function returns the mean error, the smaller the error, the better the ANN is performing.

The goal is to reduce the loss, this is already quite a lot more specific than just improving the parameters. Using differentiation, we can calculate the partial derivative of each weight and bias with respect to the loss for each set of inputs.

---

<sup>26</sup> GeeksForGeeks, “Layers in Artificial Neural Networks (ANN) - GeeksforGeeks”; Katoch, Chauhan, and Kumar, “A Review on Genetic Algorithm: Past, Present, and Future.”

<sup>27</sup> Sutton and Barto, *Reinforcement Learning : An Introduction*, 331f; Kinsley and Kukieta, “Neural Networks from Scratch in Python.”, 139.

<sup>28</sup> Nielsen, “Using Neural Nets to Recognize Handwritten Digits | NeuralNetsAndDeepLearning.”

For each data set, we first propagate forward, this means we let the ANN run on a set of inputs. Using the solution from the data set, we then calculate the loss and all the partial derivatives for this set of inputs.

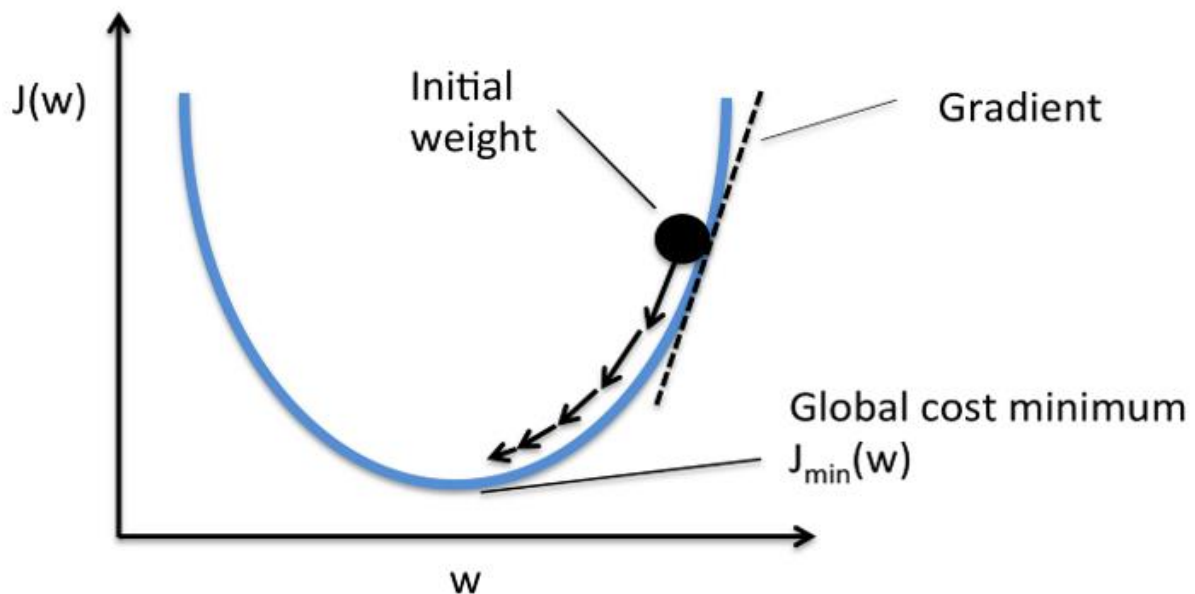


Figure 8: A graph showing how a weight is shifted down the gradient. <sup>29</sup>

The partial derivatives describe how each weight and bias affects the loss, when each weight and bias is shifted a bit against its gradient, the loss should be a bit smaller than before. This can be repeated again and again, thousands of times, until the loss is as small as possible.<sup>30</sup>

There are some problems with this: one of the biggest being local optima. It can happen that no matter what direction the weight is shifted, the weight loss will be worse than before, even though the best solution has not been reached yet. If this happens, the algorithm will be stuck and unable to move forward.<sup>31</sup>

### 2.3.2.1 Calculating the Partial Derivatives

The hard part with Gradient Descent and a lot of other training algorithms is calculating the partial derivatives. Backpropagation uses the chain rule to calculate all the partial derivatives starting in the last layer and working backwards to the first.<sup>32</sup>

---

<sup>29</sup> Kim, "PyTorch Lecture 03: Gradient Descent - YouTube."

<sup>30</sup> Nielsen, "Using Neural Nets to Recognize Handwritten Digits | NeuralNetsAndDeepLearning"; IBM, "What Is Gradient Descent? | IBM."

<sup>31</sup> Mishra, "The Curse of Local Minima: How to Escape and Find the Global Minimum | by Mohit Mishra | Medium."

<sup>32</sup> Kinsley and Kukiela, "Neural Networks from Scratch in Python.", 180.

### 2.3.2.2 The Partial Derivative

Partial derivatives are used to calculate derivatives in functions with multiple variables. This is done by treating every other variable as constant.

$$f(x) = x^2$$

$$f'(x) = 2x$$

Derivative of  $f(x) = x^2$  using the power rule.

$$f(x, y) = x^2 + y^3$$

$$f'(x) = 2x + 0 = 2x$$

Solving for the partial derivative of  $f(x, y) = x^2 + y^3$  with respect to  $x$  in a function with multiple variables by treating  $y$  as a constant.

$$f'_y = 0 + 3y^2 = 3y^2$$

Solving for the partial derivative of  $f(x, y) = x^2 + y^3$  with respect to  $y$  in the same way by treating  $x$  as a constant.

This only works as long as the other variables remain constant. When both variables are changed, the graph of a function  $f(x, y)$  does not follow the partial derivative of  $f(x, y)$  with respect to  $x$  or  $y$ .<sup>33</sup>

The partial derivative for a function  $f(x, y)$  with respect to  $x$  is written as:<sup>34</sup>

$$\frac{df}{dx}$$

### 2.3.2.3 The Chain Rule

“A special rule, the chain rule, exists for differentiating a function of another function.”<sup>35</sup>

The chain rule is used to differentiate nested functions with the form  $f(g(x))$  if  $f(x)$  and  $g(x)$  are both differentiable.

$$F(x) = f(g(x))$$

$$F'(x) = f'(g(x)) g'(x)$$

---

<sup>33</sup> MathIsFun, “Partial Derivatives.”

<sup>34</sup> NCL, “Numeracy, Maths and Statistics - Academic Skills Kit.”

<sup>35</sup> “The Chain Rule.”

The derivative for  $F(x) = f(g(x))$  with  $g(x)$  being the inner function and  $f(x)$  being the outer function.

Using the proper notation for partial derivatives, this could be written as:<sup>36</sup>

$$F'(x) = \frac{df}{dg} * \frac{dg}{dx}$$

#### 2.3.2.4 Backpropagation using the chain law

Using the chain law, we can calculate the partial derivative of each variable to get an idea of how it impacts the output. We will look at a simplified version of an artificial neural network and calculate the partial derivatives of all the weights, input, and bias.

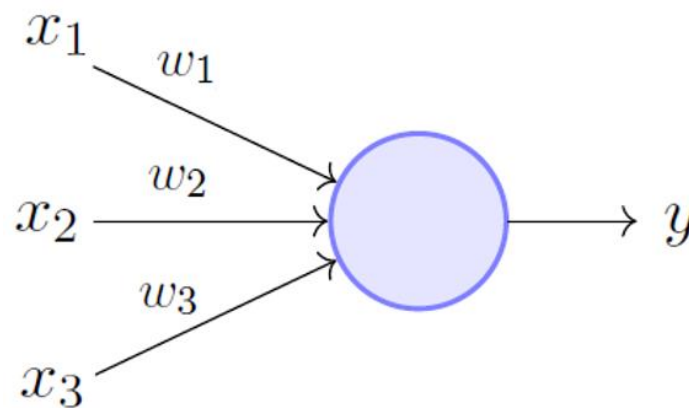


Figure 9: Model of a neuron with the weights  $W1, W2, W3$ , the inputs  $X1, X2, X3$  and the output  $y$ . The bias is not in the graphic but would be added.<sup>37</sup>

The values for all the variables are:

$$X1 = 1 \quad W1 = 6$$

$$X2 = 2 \quad W2 = 5$$

$$X3 = 3 \quad W3 = -4$$

$$\text{Bias} = 1$$

And the activation function used is the Rectified Linear function.

<sup>36</sup> Dawkins, "Calculus I - Chain Rule."

<sup>37</sup> Minsky and Papert, "Minsky-and-Papert-Perceptrons."

Doing the forward pass the result would be:

$$\begin{aligned}
 Y &= \text{ReLu}(X1 * W1 + X2 * W2 + X3 * W3 + \text{Bias}) \\
 &= 1 * 6 + 2 * 5 + 3 * -4 \\
 &= 6 + 10 - 12 = 4 \\
 \text{ReLu}(4) &= 4
 \end{aligned}$$

We first calculate the gradient of the neuron. For now, we assume the gradient is one. Here the function is written differently to make calculating the partial derivatives easier.

$$\begin{aligned}
 \frac{\partial}{\partial w0} \text{ReLu}(\text{Sum}(\text{Mul}(X1, W1), \text{Mul}(X2, W2), \text{Mul}(X3, W3), \text{Bias})) \\
 = \frac{d\text{ReLu}()}{d\text{Sum}()} * \frac{d\text{Sum}()}{d\text{Mul}(X0, W0)} * \frac{d\text{Mul}(X0, W0)}{dW0}
 \end{aligned}$$

Using chain rule the equation can be separated into small parts which can be derived separately and multiplied together. Based on the results we got from the forward pass earlier:

$$\begin{aligned}
 \text{ReLu}(x) &= \max(x, 0) \\
 \frac{d}{dx} \text{ReLu}(x) &= 1(x > 0) \\
 \frac{d\text{ReLu}()}{d\text{Sum}()} &= \frac{d}{dx} \text{ReLu}(4) = 1(4 > 0) = 1
 \end{aligned}$$

The derivative of the ReLu function with respect to x is one if x is bigger than zero, if not it is zero. In this case Sum() is 4.

$$\begin{aligned}
 \frac{d}{dx} x + c &= 1 + 0 = 1 \\
 \frac{d\text{Sum}()}{d\text{Mul}(X0, W0)} &= \frac{d}{dw0} x0 + w0 = 1
 \end{aligned}$$

The derivative of the sum of x and constants will always be one.

$$\begin{aligned}
 \frac{d}{dx} CX &= C \\
 \frac{d\text{Mul}(X0, W0)}{dW0} &= \frac{dy}{dx} x0 * w0 = x0
 \end{aligned}$$

The derivative of x multiplied with a constant, results in the constant. In this case this is the input x0.

$$\begin{aligned} \frac{\partial}{\partial w_0} \text{ReLU}(\text{Sum}(\text{Mul}(X_0, W_0), \text{Mul}(X_1, W_1), \text{Mul}(X_2, W_2), \text{Bias})) \\ = \frac{d\text{ReLU}()}{d\text{Sum}()} * \frac{d\text{Sum}()}{d\text{Mul}(X_0, W_0)} * \frac{d\text{Mul}(X_0, W_0)}{dW_0} \\ = 1 * 1 * x_0 = x_0 = 1 \end{aligned}$$

Since the partial derivative of a sum is always one, this results in the partial derivative of a weight being equal to the gradient of the perceptron, which we will calculate later, multiplied with the partial derivative of the activation function with respect to its input, multiplied with the input that the weight gets multiplied with in the forward pass.<sup>38</sup>

### 2.3.2.5 Gradient

A gradient is a vector made of all the partial derivatives of a function. The gradient of a neuron consists of all the partial derivatives of the weights, inputs, and bias. It is also used to describe the impact the output of the neuron has on the output of the ANN. Here I will explain how the gradients of the neurons are calculated.

In the last layer, the gradients are based on the postprocessing of the outputs of the ANN. If there is a loss function, for example, the partial derivative of the loss function with respect to the output of the output neurons.

In the layers before that, the gradient is the sum of the partial derivatives of the output in all the neurons in the layer after that one. The output of a neuron is used as an input in each neuron in the layer after. We can calculate the partial derivatives of the ANN with respect to these inputs and take the sum of them to calculate the impact the output of a neuron has on the ANN.

Calculating the partial derivatives of the ANN with respect to the inputs works the same as for the weights. The only part that changes is that at the end, instead of multiplying by the input, we multiply by the weight.

$$\begin{aligned} \frac{\partial}{\partial x_0} \text{ReLU}(\text{Sum}(\text{Mul}(X_0, W_0), \text{Mul}(X_1, W_1), \text{Mul}(X_2, W_2), \text{Bias})) \\ = \frac{d\text{ReLU}()}{d\text{Sum}()} * \frac{d\text{Sum}()}{d\text{Mul}(X_0, W_0)} * \frac{d\text{Mul}(X_0, W_0)}{dX_0} \end{aligned}$$

---

<sup>38</sup> Kinsley and Kukieta, "Neural Networks from Scratch in Python.", 180.



Using all of this, it is possible to calculate the impact each weight and bias has on the output for a set of inputs. If the derivative of the loss function is also factored in, we can calculate the impact each weight and bias has on the loss and decrease it based on this information.<sup>39</sup>

### 2.3.3 How I derive the partial Derivatives

In this section I will explain how I implemented the backpropagation in my projects. Before, I explained how the partial derivatives can be calculated by multiplying the different partial derivatives of the parts of the chain that come before. I used this property to separate the backpropagation into different steps that can be repeated and work in isolation.

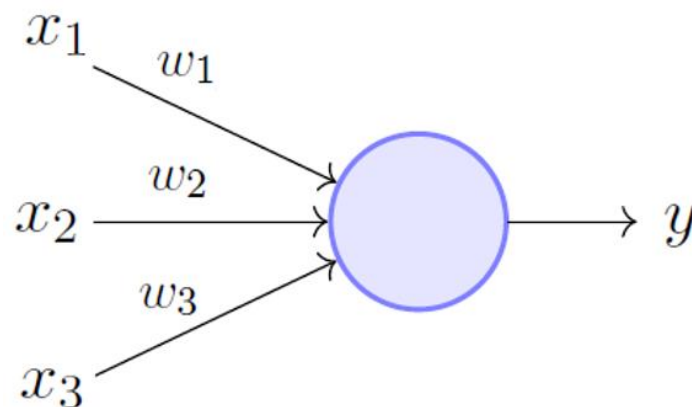


Figure 10: Model of a neuron with the weights  $w_1, w_2, w_3$ , the inputs  $x_1, x_2, x_3$  and the output  $y$ . The bias is not in the graphic but would be added.<sup>40</sup>

The algorithm needs  $x_1-x_n$ ,  $w_1-w_n$ , and the gradient of the neuron, the partial derivative of  $y$ , to calculate all the partial derivatives in a single neuron. During the forward pass, it saves all the values, and the weights are already available.

In the last layer, for which the partial derivatives are calculated first during backpropagation, the partial derivatives with respect to  $y$  are one. Depending on how the output of the ANN is used to solve a task, this partial derivative is different, but here I will assume that a single raw output is used, and as a result, the gradient is one.

---

<sup>39</sup> Kinsley and Kukieta.

<sup>40</sup> Minsky and Papert, "Minsky-and-Papert-Perceptrons."

The partial derivatives with respect to  $W_1$ - $W_n$  are calculated by multiplying the gradient of  $y$ , the partial derivative of the activation function with respect to its input and the weight corresponding  $X_1$ - $X_n$ . The partial derivative with respect to the bias is the gradient of  $y$  multiplied with the partial derivative of the activation function with respect to its input. With this all the weights and biases can be calculated if the gradient of  $y$  is known.

The gradient of  $y$  in the layers before the last one is equal to the sum of all the partial derivatives with respect to  $y$  as input to a neuron in the next layer. So, the gradient of  $y$  of the first neuron of layer  $n-1$  is the sum of the  $X_1$ s in all the neurons in layer  $n$ . The partial derivatives with respect to  $X_1$ - $X_n$  are calculated, similar to the partial derivatives with respect to  $W_1$ - $W_n$  before, by multiplying the gradient of  $y$ , the partial derivative of the activation function with respect to its input, and the weight corresponding to  $W_1$ - $W_n$ .

Starting with the last layer, this can be used to calculate the partial derivatives with respect to all the weights and biases and at the same time calculate the gradients for the neurons in the layer before. Working its way backwards, the algorithm can do this for each layer one at a time.

## 2.3.4 Reinforcement Learning and Actor-Critic Method

Reinforcement Learning is a class of solution methods that attempts to map actions to situations using trial and error to maximize a numerical reward. What separates reinforcement learning from supervised learning is that instead of labelled data, we use rewards. As a result, we do not know what the best answers are but only how good the chosen actions are.<sup>41</sup>

### 2.3.4.1 *Markov decision Process*

The basic principle of reinforcement learning is that an agent and an environment interact with each other. The agent can sense the environment and can perform actions that affect it. He also has goals, usually certain states of the environment, and it tries to achieve these goals by choosing actions based on a representation of the current state of the environment. The relationship between the agent and the environment is represented using the Markov decision process.

---

<sup>41</sup> Sutton and Barto, Reinforcement Learning : An Introduction, 1f.

---

*“MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal.”<sup>42</sup>*

---

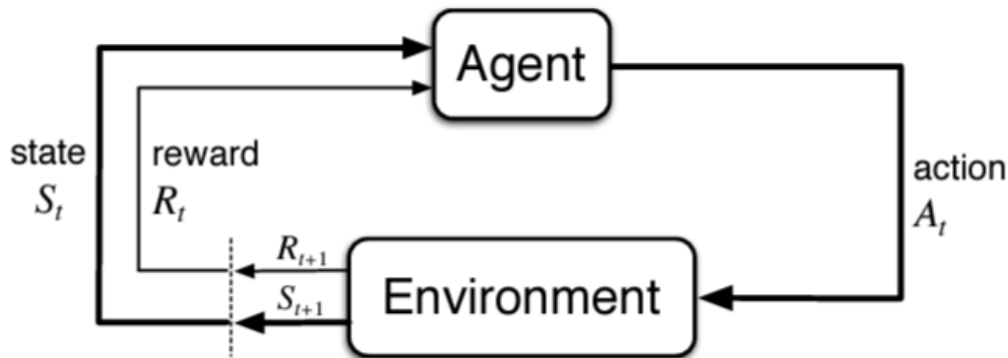


Figure 11: The agent-environment interaction in a Markov decision process.<sup>43</sup>

As depicted in Figure 11, each time step  $t$ , an agent chooses an action  $A_t$  based on the state of the environment  $S_t$ . The environment then changes from state  $S_t$  to state  $S_{t+1}$  and sends the information of state  $S_{t+1}$  and the reward  $R_{t+1}$  to the agent. The state  $S_{t+1}$  is used to decide the next action  $A_{t+1}$  while the reward  $R_{t+1}$  is used to decide if the actions taken before were good or bad and learn based on this information.

The rewards the agent receives from the environment represent the achievement of goals or can be negative in case of failures. The agent tries to maximize the total rewards, not just the immediate rewards. There are a variety of different algorithms that try to learn from experience to maximize the total reward of future attempts. It uses trial and error to learn which actions in which states result in the highest rewards.<sup>44</sup>

#### 2.3.4.2 Solving finite Markov decision processes

A finite Markov decision process consists of a finite number of states and actions and can be solved using a value function and a policy function.

For the following explanation, I will use a combination of a state-action value function and a policy that uses the values of this function to create the action probabilities. This is called Q-Learning and is one of the most intuitive algorithms in reinforcement learning. The state-action value function is a value function that assigns each combination of state and action an expected total reward when following the current policy. For finite Markov decision processes, it can be, in its simplest form, a table.

---

<sup>42</sup> Sutton and Barto, 47.

<sup>43</sup> Sutton and Barto, 48.

<sup>44</sup> Sutton and Barto, 47f.

State	Right	Left	Up	Down
1	0	0.31	0.12	0.87
2	0.98	-0.12	0.01	0.14
3	1	0.10	0.12	0.31
4	0.19	0.14	0.87	-0.12

Figure 12: A state-action value function with four states and four actions, Right, Left, Up and Down. For each state except state 3, the action with the highest expected total reward is marked green. In state 3 a random action is selected to ensure exploration.<sup>45</sup>

The policy uses this table to decide what action to take. It must balance taking the best actions with actions it predicts to be worse to ensure exploration. This is one of the main problems in reinforcement learning and is called the exploration vs. exploitation dilemma. The agent must explore actions without any known reward at the end since they might lead to the optimal reward, but they also want to exploit actions they already know lead to a reward since finding rewards is their goal.<sup>46</sup>

The policy function creates a probability for each action in a state based on the value function to determine which one should be performed. For a table as above a greedy policy function would choose the action with the highest value. A policy function that balances exploration and exploitation better would be to choose the action with the highest value nine times out of ten and the rest of the time choose a random one.

Each time a terminal state ends, this could be reaching a destination, winning, or failing, a new episode starts. Each episode is completely unrelated to the others and has a start point and an end. At the end of each episode, the total reward is calculated, and all the chosen state-action pairs that have been chosen along the way get updated according to an algorithm like the Bellman equation. They should then be closer to the optimal state-value function and their information more accurate than before. The policy is then able to choose better actions based on the value function. This is repeated until the performance converges. There are also continuous tasks that do not have episodes and go on indefinitely, but they are not relevant for this paper.

There are other value functions, like the state-value function, which will be important later. Depending on the task, different pairings of value and policy functions lead to better or worse results.

<sup>45</sup> Comi, "How to Teach an AI to Play Games: Deep Reinforcement Learning."

<sup>46</sup> Comi; Kahn, "Reinforcement Learning – Exploration vs Exploitation Tradeoff - AI ML Analytics."

I am using Artificial Neural Networks to approximate the value and policy functions, but this is far from the only option. Reinforcement learning not only looks at ANNs but has been used in other areas of machine learning.<sup>47</sup>

#### 2.3.4.3 *Actor-Critic Method*

Chat GPT is without doubt one of the most exciting examples of what machine learning is capable of. OpenAI, the company behind Chat GPT, has been researching artificial intelligence for years with the goal of ensuring that AI does not become a technology reserved for a few companies.<sup>48</sup>

One of their most successful projects was OpenAI Five, an artificial neural network that taught itself how to play Dota II, a highly complex video game that introduced new challenges compared to previously solved games like Chess and Go. It only used self-play to learn. Using Proximal Policy Optimization, a version of the Actor-Critic Method, it was able to achieve superhuman performance and beat the World Champions.<sup>49</sup>

Another successful example of the Actor-Critic Method is AlphaStar by Google DeepMind. It is an artificial neural network that managed to win against professional e-sport athletes in StarCraft II, a real-time strategy game. It was trained on professional games and achieved its success by continuing its training using self-play. In self-play an ANN is trained by playing against older versions of itself.<sup>50</sup>

The Actor-Critic Method belongs to the family of policy gradient methods. It is based on a version of the Monte Carlo policy gradient with base line and temporal difference learning.<sup>51</sup>

---

<sup>47</sup> Sutton and Barto, Reinforcement Learning : An Introduction, 47f.

<sup>48</sup> OpenAI, “OpenAI Five | OpenAI”; OpenAI, “OpenAI Charter | OpenAI.”

<sup>49</sup> OpenAI et al., “Dota 2 with Large Scale Deep Reinforcement Learning.”

<sup>50</sup> AlphaStar team, “AlphaStar: Mastering the Real-Time Strategy Game StarCraft II - Google DeepMind.”

<sup>51</sup> Sutton and Barto, Reinforcement Learning : An Introduction, 321.

#### 2.3.4.4 *Policy gradient methods*

Policy approximation is a class of reinforcement learning methods that completely separates the policy from the value function. This means that the value function is only used during training, and afterwards the policy function can solve problems without consulting the value function. This works great for Artificial Neural Networks since it simplifies the postprocessing of the output since it works in isolation. It also has the advantage of being completely online. Different from the Q-learning algorithm I described earlier, it does not wait until the end of an episode to update the ANNs, instead, they can be changed at every timestep and do not need something like a bellman equation.<sup>52</sup>

#### 2.3.4.5 *Actor-critic split*

The Actor-Critic Method is split into an actor, a policy function that decides what action to take, and the critic, a state-value function that evaluates the expected reward for the current state of the environment when following the current policy. The policy function is the agent, while the critic is an intermediary between the environment and the policy. During the learning process, the critic is used to evaluate the action taken by the actor by comparing the state value before and after the action.<sup>53</sup>

---

<sup>52</sup> Sutton and Barto, 321.

<sup>53</sup> Sutton and Barto, 331

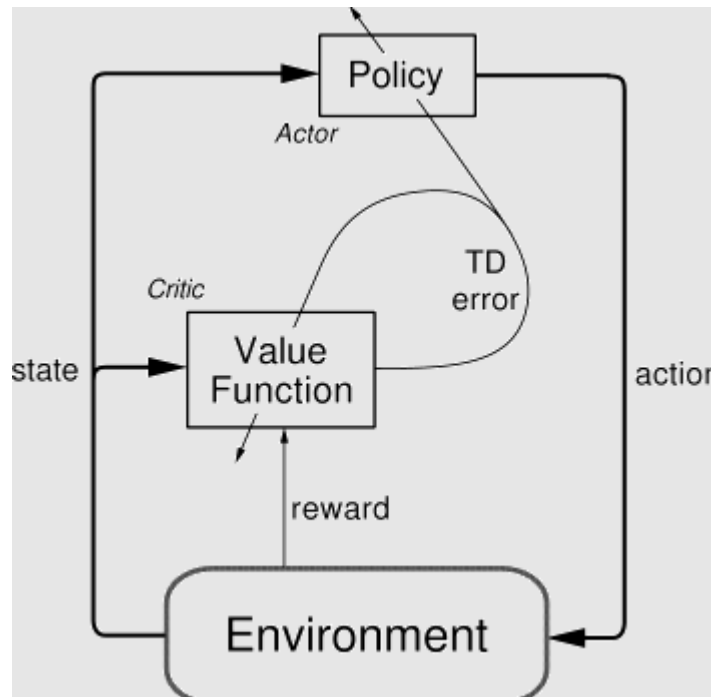


Figure 13: Graph describing the relationship between the policy function as the actor, the value function as the critic and the environment.<sup>54</sup>

Each timestep, the advantage is calculated, in the graphic, it is called TD error. The advantage is the difference between the sum of the state value after the action and the received reward and the expected reward, calculated by the value function, before the action was taken.

If the advantage is positive, the action taken was better than the average action taken by the policy in the same state, and it should be reinforced. If it is negative, it should be disincentivized. Using this, the policy is updated each step according to the feedback received by the state-value function.

At the same time, the advantage is used to improve the state value. The advantage is effectively the difference between the expected reward and a more accurate version of the expected reward since the value function is estimating a timestep less and has more information. This means that if the advantage is positive, the state value should have been a bit larger, and if it is negative, the state value should have been smaller. Using this, we can update the state-value function each step to more closely match the true state-value function.

<sup>54</sup> Lee, "6.6 Actor-Critic Methods."

In terminal states, the state value is hard set to zero, since if there are no steps afterwards, the expected reward is always zero. Also important is that the size of the advantage is important since it is used to determine how big the changes to the policy function and the state value function should be.

#### One-step Actor–Critic (episodic), for estimating $\pi_{\theta} \approx \pi_*$

```

Input: a differentiable policy parameterization  $\pi(a|s, \theta)$ 
Input: a differentiable state-value function parameterization  $\hat{v}(s, \mathbf{w})$ 
Parameters: step sizes  $\alpha^{\theta} > 0, \alpha^{\mathbf{w}} > 0$ 
Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $\mathbf{w} \in \mathbb{R}^d$  (e.g., to  $\mathbf{0}$ )
Loop forever (for each episode):
  Initialize  $S$  (first state of episode)
   $I \leftarrow 1$ 
  Loop while  $S$  is not terminal (for each time step):
     $A \sim \pi(\cdot|S, \theta)$ 
    Take action  $A$ , observe  $S', R$ 
     $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$  (if  $S'$  is terminal, then  $\hat{v}(S', \mathbf{w}) \doteq 0$ )
     $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$ 
     $\theta \leftarrow \theta + \alpha^{\theta} I \delta \nabla \ln \pi(A|S, \theta)$ 
     $I \leftarrow \gamma I$ 
     $S \leftarrow S'$ 

```

Figure 14: Pseudocode for the Actor-Critic Method.  $\delta$  is the advantage,  $R$  is the reward,  $\nabla \ln \pi(A|S, \emptyset) = \nabla \pi(A|S, \emptyset) / \pi(A|S, \emptyset)$ .<sup>55</sup>

$\nabla \pi(A|S, \emptyset) / \pi(A|S, \emptyset)$  is the gradient of the policy divided by the probability of the chosen action being chosen. This is done to ensure that an action that has a small chance of being chosen is weighted heavier to ensure it stands a chance if it outperforms other actions.

This way, the policy function should be a bit closer to the optimal policy function with each timestep, while the value function should approach the optimal value function for the current policy function.<sup>56</sup>

<sup>55</sup> Sutton and Barto, Reinforcement Learning : An Introduction, 332.

<sup>56</sup> Sutton and Barto, 331f.



## 3 Implementation of ANNs into Pong and Footsies

For the remaining part of this paper, I will discuss my implementation of these algorithms. I will go into the differences of the implementations for *Pong* and *Footsies* and elaborate on some of the problems I faced. The chapters are structured based on what learning algorithm they use to better illustrate how the game influenced the implementation of the learning algorithms.

### 3.1 Background on Programming

For both *Pong* and *Footsies*, I forked an open-source version and wrote my algorithms directly into the source code. This enabled me to directly access the variables and use them as inputs and have my output directly interface with the game engine instead of simulating key presses. Both games were made in the game engine Unity and use the computer coding language C#, which I was already familiar with.

An ANN itself consists of a four-dimensional list of floats that represent each weight and bias and some additional information like the fitness values. Each “learning attempt” is saved in a separate object where all the parameters as well as all relevant ANNs are stored. These objects form a list, which is saved in JSON files between runs. Everything is single-threaded, and barely any optimization has been done, which most likely impacts the performance quite badly. Compared to projects like PyTorch, my algorithms are inefficient and do not use the capabilities of the hardware to the full extent.

Almost all the code has been written by me based on the sources stated and the official C# and Unity documentation. The few time where I have copied parts of the code from places like stack overflow have been marked in the code, and sources have been added.

## 3.2 Game Logic and ANN Implementation Objectives

### 3.2.1 Pong

**Game logic:** *Pong* is an old flash game and was one of the first games ever to be created. It is inspired by table tennis, and the player controls a paddle that moves up and down. The goal is to hit the ball using the paddle to bounce it back. Every time a player misses, and the ball goes past the paddle, the opponent scores a point.<sup>57</sup>

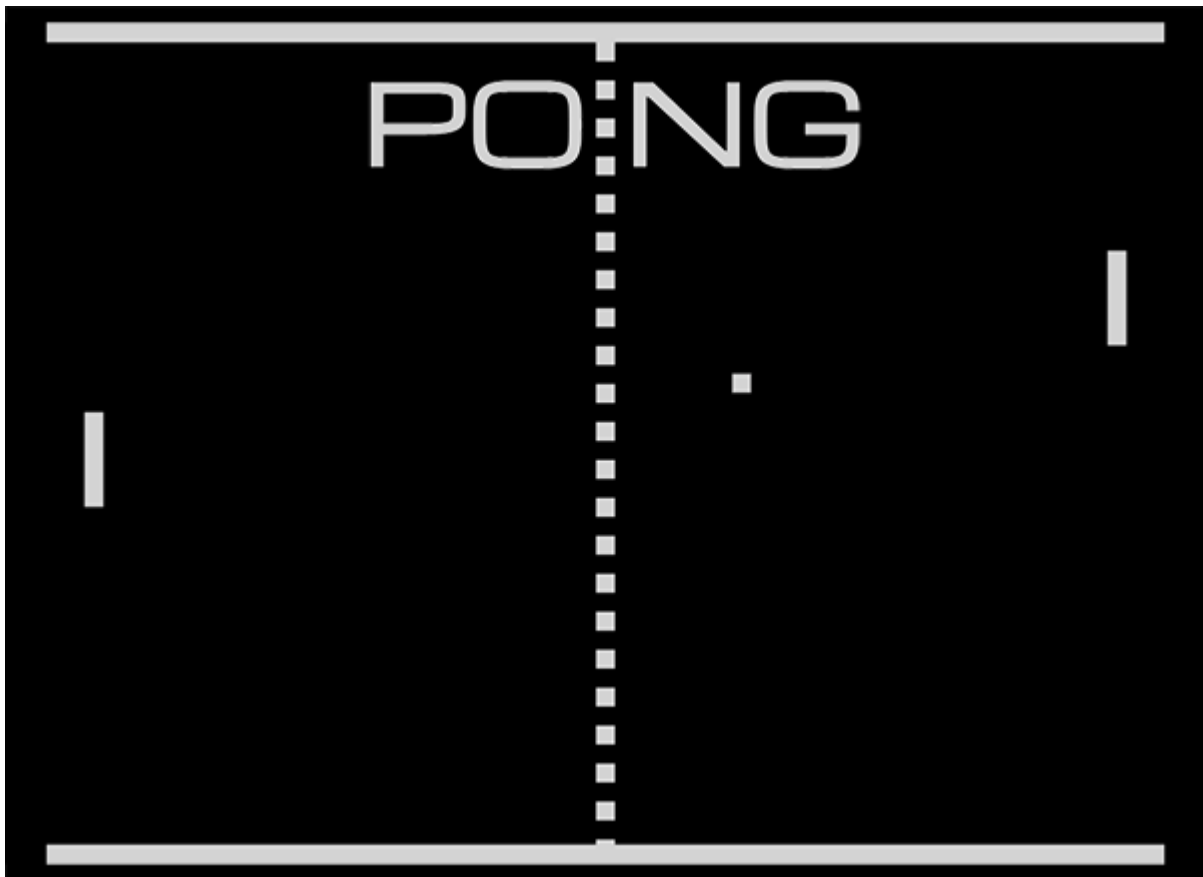


Figure 15: Screenshot of Pong. The white rectangles on the side are the paddles controlled by the player and the white square close to the middle is the ball.<sup>58</sup>

The goal is to hit the ball using the paddle to bounce it back. Every time you miss, and the ball goes past you, your opponent scores a point.<sup>59</sup>

**ANN Objective:** Based on this game logic, the goal of the implementation was to develop an ANN that would learn how to move the paddle to never miss the ball.

---

<sup>57</sup> “Pong Game.”

<sup>58</sup> “Pong - Play Game Instantly!”

<sup>59</sup> “Pong Game.”

### 3.2.2 Footsies

**Game logic:** *Footsies* is a very basic 2D fighting game similar to games like streetfighter. Using a keyboard, the player controls a character and have it move, perform attacks and blocks against another player.

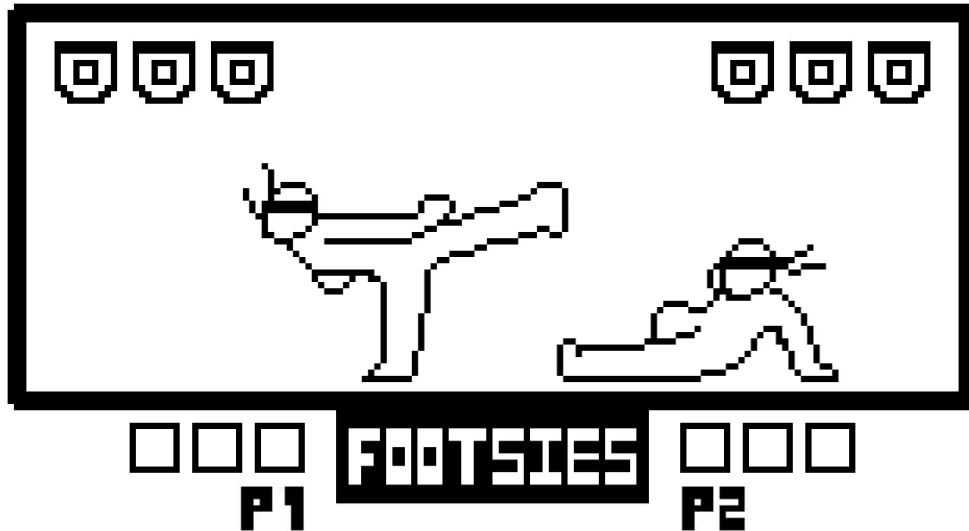


Figure 16: Screenshot of *Footsies*. In the top corners the shields represent how often you can block. The rectangles at the bottom indicate the current score in a best of five.<sup>60</sup>

The player can score by KO, hitting the opponent with a special attack. Each round a player can block three attacks; if he tries to block afterwards, he gets stunned and is unable to perform actions for a while. The move set consists of walking, a dash, blocking, two normal attacks, and two special attacks.<sup>61</sup>

**ANN Objective:** Based on this game logic, the goal of the implementation was to develop an ANN that would learn to move and perform attacks and/or blocks in a way that would enable the character to prevail in the fight against the other character.

---

<sup>60</sup> HiFight, "FOOTSIES - HiFight."

<sup>61</sup> HiFight.

## 3.3 Approach 1: Genetic Algorithm

The artificial neural network I use in the Genetic Algorithms is a fully connected network using the sigmoid activation function. Each individual is initialized with random values for all weights and biases.

### 3.3.1 Pong

#### 3.3.1.1 *Implementation*

The neural network for *Pong* has five inputs, which describe the position of the agents paddle, the position of the ball, and its movement as a 2D vector. It has a single output, which returns a value between zero and one using the sigmoid function. This value determines the chosen actions, i.e. the movement of the paddle: over 0.6 is moving up, under 0.4 is moving down, in between no action is taken. This means that the artificial neural network has absolute control over what action is taken and does not rely on randomness.

For *Pong*, the implementation of the Genetic Algorithm was straightforward:

- Each individual gets a certain number of lives, they play against a perfectly playing bot I developed, which simply keeps the paddle at the level of the ball and lose a life every time the ball touches their wall. When they run out of lives, their attempt is finished, and their fitness score is set in stone.
- During their attempt, they get rewarded for each ball they manage to hit. This reward is based on how accurately they hit it and on how far the distance they had to travel to hit it. This is to punish individuals who stay in one spot and just get lucky. Each point represents a chance at getting picked in the random draw for reproduction.
- Each new individual is based on two individuals, and each parameter is either copied from one or the other. For mutation, each individual possesses a mutation factor, a floating-point value between zero and one, which, for each parameter, is multiplied with a second random floating-point value, and if the result passes a threshold, the parameter is mutated. This creates individuals that are highly mutated and others that are close to unaltered.
- The algorithm also always keeps an unchanged version of the best-performing individual in the new generation to ensure it does not regress.

#### 3.3.1.2 *Result*

This algorithm works well and manages to find ANNs that can catch over 99.99% of balls after running for a couple of hours. The convergence time can vary quite a lot and is dependent on chance. I was surprised by the possible improvement of a solution, once

one is found that works, but it needs to first find something that works better than random movements to then improve it.

The solution it finds is quite a simple one. It applied the same logic as the bot and simply tries to keep the paddle on the same height as the ball. If the ball is beneath the paddle, it moves down, and if it is above, it moves up. While this essentially the same the perfectly playing bot I implemented does, the bot remains more accurate. The reason is that the bot is designed to always be correct, while the ANN may only achieve the same level of accuracy if trained to perfection.

### 3.3.2 Footsies

#### 3.3.2.1 Implementation

For *Footsies*, the input data consists of seventeen variables that describe the positions of both characters, their current action, and where within the action they are. The ANN has two outputs; one represents the input for the attack button, while the other one defines if the character should move. These are the same options the player has: attacking, moving forward, and moving backwards. All other actions, like blocking and special attacks, are performed by some combination of movement and attack inputs.

The algorithm for *Footsies* is, for the most part, the same as for *Pong*. The only big difference is the process of assigning fitness values. The problem with *Footsies* in this context, is, that I do not have a way of objectively assessing the performance like I do for *Pong*. So, the approach is that instead of trying to measure its absolute performance, I measure it in relation to the other individuals in the same generation.

The way I do this is by letting them play against each other. Each generation I throw all contestants into a tournament, and their ranking determines their chances when reproducing. For the tournament design, I have tried out quite a few different algorithms. I decided against a bracket-type tournament, because they can be quite complicated and have trouble ranking the worse performing individuals. The reason being their focus on who places on the podium while disregarding anything below it.

The most interesting solution I tried was repurposing a bubble sort algorithm, where instead of comparing values, to decide if I should switch the places of two individuals, I had them compete and the winner got the better position. This worked quite well. The problem was, that it was not that efficient with larger population sizes, and since the outcome of the same duel could be different, it would never flag as the correct order and only end when a hard cap was reached.<sup>62</sup>

The solution that ended up working the best was an Elo rating system. The algorithm, that competitive chess and nowadays many PvP games use, works by assigning an Elo rating

---

<sup>62</sup> “Bubble Sort Algorithm - GeeksforGeeks.”

to each player, or here individual, and searching for an opponent with a similar Elo rating. The Elo rating is used to calculate the win probabilities of each player. After the game, the win probabilities determine how the Elo ratings of each player change. A win against a stronger opponent gets rewarded more, and a loss against a weaker opponent gets punished more. This ensures, that the risks and rewards of having a stronger or weaker opponent always stay the same in relation to each other. Another big advantage is, that I can end the tournament before a final winner is determined and have multiple individuals be equal. This means I can increase the population size by a lot without raising the computation time exponentially. At the end, I use the Elo rating as fitness values.<sup>63</sup>

### 3.3.2.2 Result

The performance of the Genetic Algorithm for *Footsies* is questionable at best. While the ANNs are constantly evolving, and learn to beat their old versions, a real improvement happens only up to a certain point. The problem is that it gets stuck in a rock-paper-scissors-like situation where they adopt a dominant strategy and outperform their opponents to subsequently spread this strategy through the next generation. This strategy is dominant, until an individual finds a different strategy that counters the dominant one. This can happen over and over until a previously dominant strategy gets adopted again from where another loop will start. This is a common problem in machine learning when using self-play. It is also quite hard to make reasonable measurements of the performance, as the objective, which is beating the enemy, changes when the enemy changes.<sup>64</sup>

There are quite a few parameters, that can be adjusted to improve the Genetic Algorithm. The most important ones being the population size and the number of lives, or fights in the case of *Footsies*, each individual gets. Both increase the improvement with each generation, while also increasing the time spent learning. Especially for *Footsies*, a large population size takes a long time to set the fitness values and is very inefficient. If the number of lives is increased, the consistency of the learning rate increases since the observed data for each individual rises and the fitness values reflect the true fitness of the individual better. If the number of lives is too small, the measured accuracy makes big jumps in both directions.

---

<sup>63</sup> Kalebka, “Developing an Elo Based, Data-Driven Rating System for 2v2 Multiplayer Games | Towards Data Science.”

<sup>64</sup> Simonini, “Self-Play: A Classic Technique to Train Competitive Agents in Adversarial Games - Hugging Face Deep RL Course.”

Good parameters have a big impact on the performance of any machine learning algorithm and finding them is mostly reliant on trial and error. I was able to find working parameters for my projects, but they are probably far from perfect and limit how fast the ANNs can learn.

## 3.4 Approach 2: Reinforcement learning

The way the ANNs chose their actions for the reinforcement learning algorithm is different from the way they did for the Genetic Algorithm. Previously, the actions were chosen based on the size of the output values, if output  $a$  is bigger than 0.5, action  $a$  is performed. Now, the outputs of the ANN are normalized using a SoftMax function. This results in action probabilities. Each action gets an assigned probability, which is determined by the size of the correlating output. This is to ensure that each action always has a chance of being chosen. The Genetic Algorithms solve the problem of balancing exploration and exploitation by having a pool of different ANNs. In reinforcement learning, the action probabilities ensure that the exploration never stops.<sup>65</sup>

I have also implemented He-initialization, which initializes biases to zero and weights based on a normal distribution with the goal of keeping the variance to a minimum. This should provide a better basis for improvement. I also exchanged the sigmoid function for a rectified linear function that can return true zeros and larger numbers, which helps the ANN converge. It is also easier to differentiate and does not have the problem of vanishing gradients, which I will talk about later.<sup>66</sup>

### 3.4.1 Pong

The reinforcement learning algorithm consists of two ANNs. The policy function uses the same inputs as the ANNs of the Genetic Algorithm use but now has two outputs instead of one. This is to accommodate the action probability for moving up and down. The value function also uses the same input but only has one output, which returns the expected reward in the current state when following the current policy.

Each episode is defined as the ball crossing the court, bouncing of the opponent or his wall, and hitting the agent or his wall. This means that the ball hitting the agent, or his wall, is a terminal state, ending the episode and starting a new one. If the agent hits the ball, he gets rewarded; if he misses it, he gets punished.

---

<sup>65</sup> Lee, “2.3 Softmax Action Selection.”

<sup>66</sup> Bendersky, “The Softmax Function and Its Derivative - Eli Bendersky’s Website”; Goel, “Kaiming He Initialization. We Will Derive Kaiming Initialization... | by Shaurya Goel | Medium.”

### 3.4.2 Footsies

For *Footsies*, the policy function and the value function use the same set of inputs, which are very similar to the ones in the Genetic Algorithm. The value function has one output, and the policy function has seven: one for each action. Each episode is one round in-game. At the end of each episode, the algorithm gets rewards based on whether it has won or lost. During the episode, it can earn rewards by, for example, breaking a shield.

Originally, the algorithm was supposed to use self-play to learn. In theory, playing against old versions of itself should allow the algorithm to continuously improve, in practice, this never worked properly. The reinforcement learning algorithm for *Footsies* in general does not work as I had hoped. Most likely there are still some bugs I never found, that would solve the problems the algorithm has. It's hard to determine whether the problem with the self-play lies in the algorithm or my code. A second problem was that it was hard to measure the performance of the algorithm. There is a preprogrammed bot in *Footsies* that can be used as a practice dummy, so I switched out the self-play mechanism for this bot as a training partner.

There are a few benefits to using the bot as a training partner: The goal is clearly defined because the opponent does not change. Before, the goal was improvement in general, and the only way to measure the progress was by playing against bots that did not accurately reflect the goal. Measuring performance is easier and more reliable since the goal is now to win against one specific bot, which can also be used to measure progress.

With this change, the data indicates that the algorithm can learn a bit. It is far from perfect, it has troubles getting high win rates consistently, and it oftentimes evolves backwards for unknown reasons. But it can improve and play well enough to beat the bot more often than it loses.



### 3.5 Comparison of the Approaches

Comparing the genetic and reinforcement learning algorithms based on their performance does not make a lot of sense. The effect that the parameters have on the performance of these algorithms is too impactful to be ignored, and it is hard to test how close to optimal the chosen parameters are. I will talk more about optimizing the parameters later, but assuming my parameters are optimal and using them for comparisons between algorithms would be a mistake. Also, my implementation is far from perfect, and the level of efficiency would have to be close for comparisons, which I cannot guarantee. What I will do instead is compare the problems they face and where their advantages lie.

The reinforcement learning algorithm works quite well. At first, the algorithm needs a bit of time until the value function has achieved a basic accuracy and is effective at training the policy. Afterwards, the improvement rate ramps up quickly until it reaches an accuracy of around 85%, where the curve starts to flatten out.

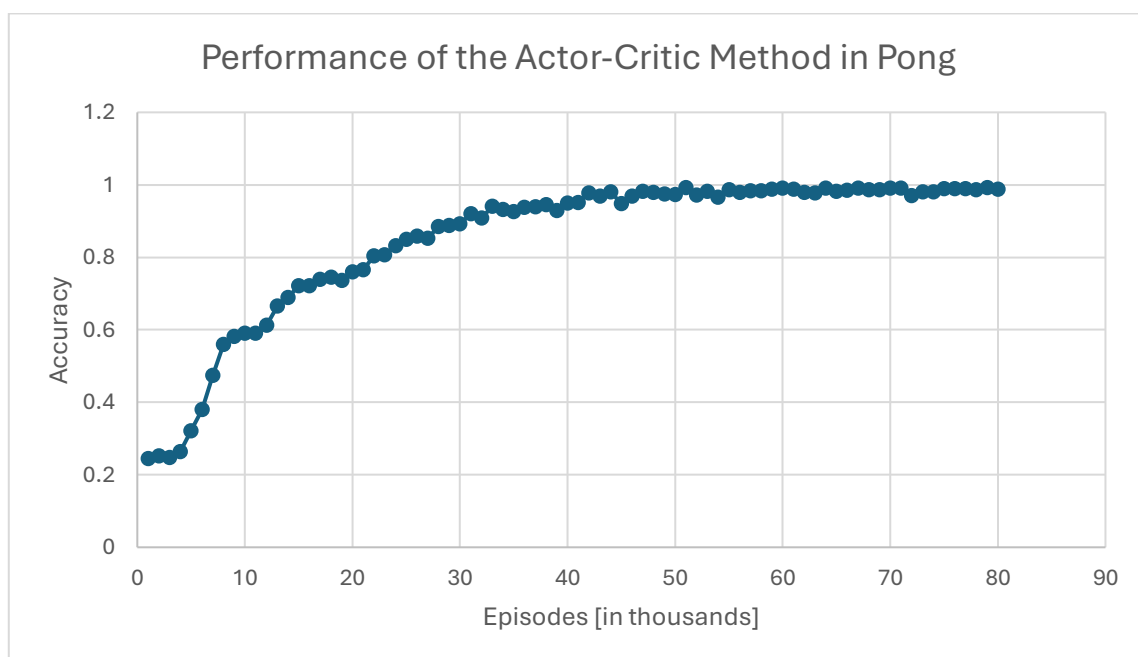


Figure 17: Performance of the reinforcement learning algorithm with the following parameters:

	Value function	Policy function
Learning rate	0.001	0.0001
Layer count	2	2
Layer size	6	6
Decay	0.99	0.99

There are two possible reasons for this: If the accuracy is high, when the agent fails, it is very close. The value function has problems distinguishing this from hitting, since the difference can be very small. This means that in comparison the approximation of the state value is barely any different for a state where the agent ends up hitting the ball to one where the agent misses it. This in turn reduces the magnitude of the advantage and slows down any changes to the neural network since the values are much smaller.

The second reason is a result of how the gradients are calculated. When the SoftMax function approaches a probability of 100% for one action, all the partial derivatives approach zero. This being the vanishing gradient problem I mentioned earlier in context to the sigmoid function.

The problem with vanishing gradients, is, that when you backpropagate, it affects all partial derivatives that come before it: in this case all the weights and biases. This means that their partial derivatives also approach zero, which immensely slows down any changes in the weights and biases. Not only that, but the partial derivatives of the SoftMax function with respect to its output approaching zero also mean that the inputs into the SoftMax function have a smaller effect on the output and changes to the inputs have a smaller impact on the output.

This is a behaviour we normally want to achieve. It ensures that exploration never fully stops, and usually this small chance of a different action being chosen barely has an effect. The problem is that in *Pong*, the speed of the ball along its x-axis is constant, so if the angle of the ball is steep, it moves faster than normally. Still, if moved perfectly, the paddle will always be able to catch the ball when following it. These steep angles pose a problem for the reinforcement learning algorithm, since it is not able to move perfectly. The Genetic Algorithm does not have this problem, for it does not use action probabilities.

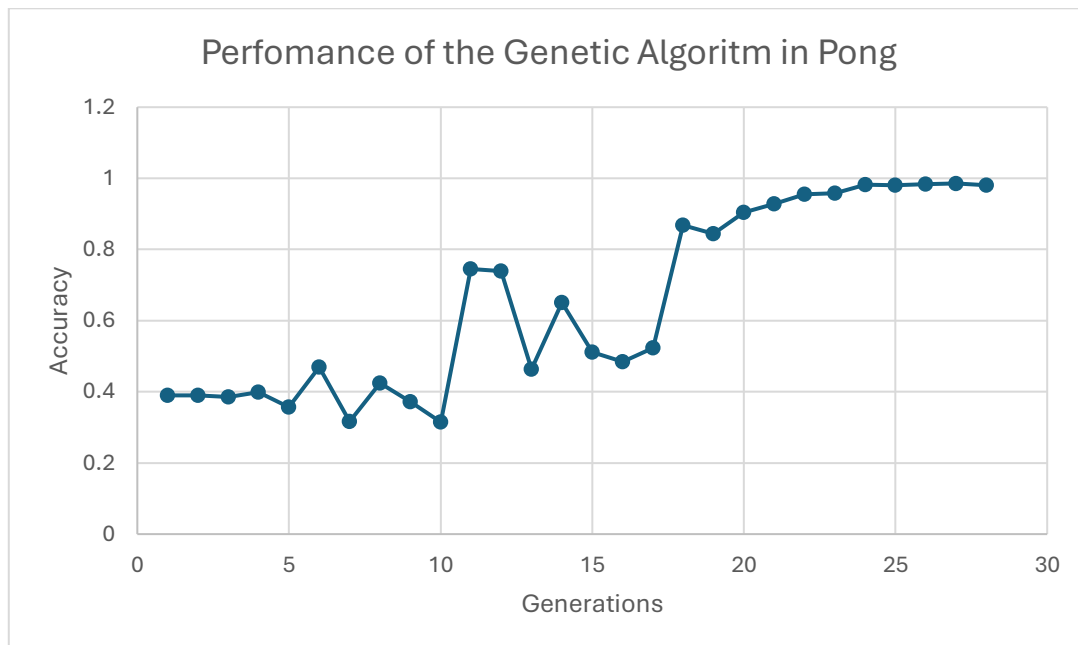


Figure 18: Performance of the Genetic Algorithm in relation to the generation. Here the Genetic Algorithm had a population size of 200, 10 lives and a mutation threshold of 0.1.

The graph for the Genetic Algorithm has fewer data points since I can only measure data at the end of each generation. I tried slowing down the progress to increase the resolution, but that leads to problems.

At the start of the learning process, the curve of the Genetic Algorithm has problems. It needs to get lucky and find an individual with a minimally better performance than the others and can then provide a starting point for future improvements. This can be improved by increasing the population size and number of lives, but they both increase the time it takes for a generation to run and decrease the resolution even further.

A second problem is that the data, which the fitness value is based on, is quite limited, and through randomness, a bad solution can achieve a high fitness value and set the next generation back.

In return, once the Genetic Algorithm has surpassed the hurdle at the start, it does not have the vanishing gradient problem and is able to achieve an accuracy of close to 100% without the same problems the reinforcement learning algorithm faces. Afterwards, it also starts having problems with improving further.

While that does not affect the curve, since it describes the relationship between accuracy and generations, the time it takes to set the fitness value for each generation increases as the accuracy increases since each individual lives longer and takes more time. I hardcapped the amount of total hits before ending the attempt. Else an ANN with 100% accuracy would live forever and essentially stop the learning algorithm before ever getting saved.

In my opinion, the Genetic Algorithm is only as successful as it is, due to the simplicity of the problem. It can easily stumble upon a solution and improve it, since the number of parameters is quite small and the solution simple. For more complex problems, the algorithm would have to try way more, and its performance would decrease. In *Footsies*, the performance is already worse than in *Pong*. It takes a long time to find an improvement, and it has troubles distinguishing between situations. It does not really change its actions based on the state and instead finds the best combination of inputs to spam. The Genetic Algorithm is also very inconsistent and unstable, small changes to the parameters can lead to complete failure.

## 4 Exploring parameters

Parameters can have a big impact on how fast an ANN is able to learn. I want to find out how they influence the learning process, and I want to try to optimize the parameters to find out what my algorithms are capable of.

### 4.1 Method

All the tests are done in *Pong*, as it is faster, easier to measure, and easier to compare results. On top the learning algorithms for *Footsies* have some problems that I was unable to fix and do not perform as well as the algorithms for *Pong*.

The learning algorithm and the measurements use a perfect enemy that catches every ball. The perfect enemy tries to always keep the same position on the y-axis as the ball, if perfected this leads to an accuracy of one and is also the strategy the learning algorithms usually try to learn.

In normal *Pong*, the rebound angle is based on the impact position of the ball on the paddle. For the perfect bot, this is changed, and the rebound angle is random within the same range as the normal rebound angle. This is to ensure that all the, in the normal game possible, states are represented in the training data.

In the training algorithm and these tests, the goal is not to win against the opponent but to catch every ball. This makes training a lot easier and, when perfected, would still make the algorithm unbeatable. I measure the performance of the algorithm with accuracy: the number of balls the algorithm managed to hit divided by the total number of tries. The goal is an accuracy of one, where every ball is hit, and the algorithm is unbeatable. A random algorithm, like standing still, is around 0.25 accuracy because the paddle covers around 25% of the wall.

All the tests are done in the same version of the algorithm. During the learning process, each episode is marked as either a success or a miss. This means that every time the ball hits the paddle the algorithm controls or the wall it is supposed to protect, the algorithm gets a reward, or a punishment and a new episode starts. Every thousand episodes the accuracy is measured. This means each accuracy value represents the performance of the policy over the last thousand episodes.

I set the goalpost for a successful learning attempt at 0.99 accuracy to have a more absolute measurement, the number of episodes it takes the algorithm to reach 0.99 accuracy. A value that is easier to compare and clearly defines a successful learning attempt. I consider a failed learning attempt one, that is stuck at the minimum 0.25 accuracy after a couple hundred thousand episodes.

## 4.2 Parameters

There are a lot of parameters and details of the algorithm that could be changed to improve the performance. I focused on just the size of the ANNs and the learning rate and left everything else constant.

I decided not to do any changes that would require rewriting parts of the code for each test, like different inputs or a different reward function. Rewriting code would have required time and potentially introduced new problems.

All the tests required a lot of time since the training is quite slow. The size of the ANNs not only has a big influence on the performance of the learning algorithm, but it also determines the complexity of the task the algorithm is able to solve. This makes it more interesting than most parameters. The learning rate is probably the parameter with the biggest impact on how long it takes the algorithm to succeed. It is also often the first parameter that people try to optimize.

## 4.3 Hypotheses

My hypotheses for the experiments are as follows:

### 4.3.1 Size of the ANNs

1. There is a minimum size the ANNs need to be able to find an optimal solution.
2. Algorithms with this minimal size will learn faster than algorithms with larger ANNs.

### 4.3.2 Learning Rate in Reinforcement Learning

3. Small learning rates will lead to a flat curve and slow down the learning process.
4. Large learning rates can lead to a worse learning process.
5. Too large learning rates can lead to complete failure of the algorithm.

## 4.4 Results

### 4.4.1 Baseline

First, I would like to establish a baseline for comparison. All the parameters are chosen based on what I noticed worked quite well, this means they are already optimized to some degree.

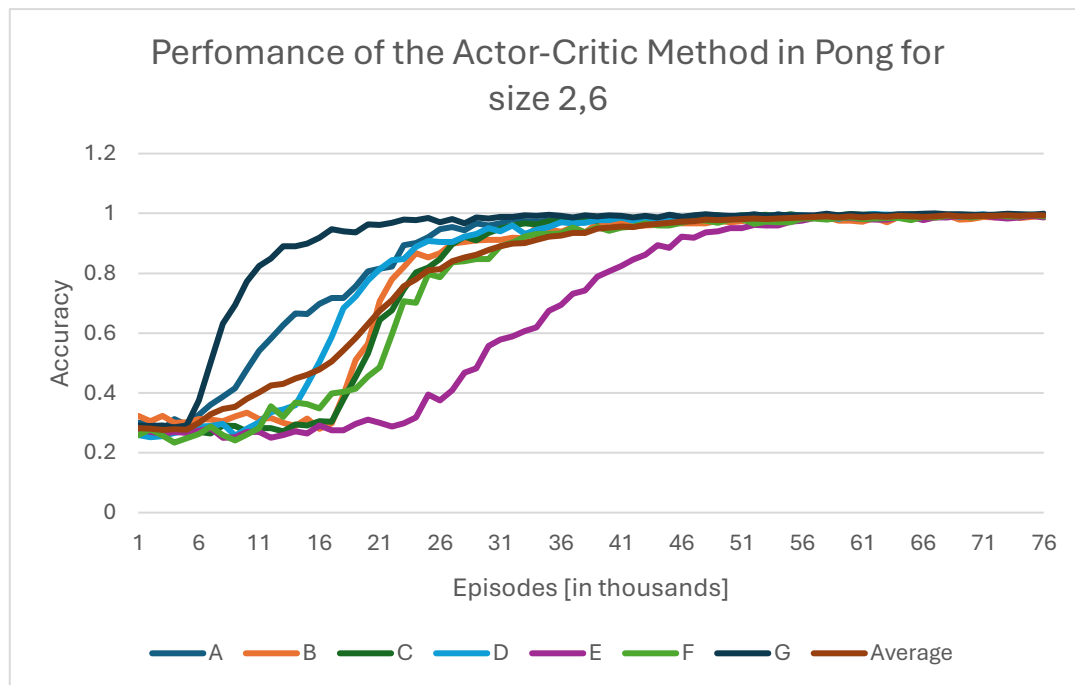


Figure 19: Performance of multiple learning attempts with the same parameters.<sup>67</sup>

Table 1: The parameters used for all the learning attempts in Figure 19.

Parameters:	Value function	Policy function
Layer count	2	2
Layer size	6	6
Learning rate	0.001	0.0001

<sup>67</sup> See Appendix D

Table 2: Analysis of points where the policy reaches an accuracy of 0.99. Same data set as Figure 19.

### Descriptive Statistics

Episodes to reach 0.99 accuracy [in thousands]	
Valid	8
Mean	53.000
Std. Deviation	12.570
Minimum	33.000
Maximum	69.000

The test data shows that the difference between learning attempts, even with the same parameters, is high. The high standard deviation and the big difference between the minimum and maximum show that repeated tests with the same parameters can have outcomes with significant differences.

A problem, that the graph does not depict, are failed learning attempts. There is an inherent instability in ANNs that can lead to not learning anything or the performance worsening.



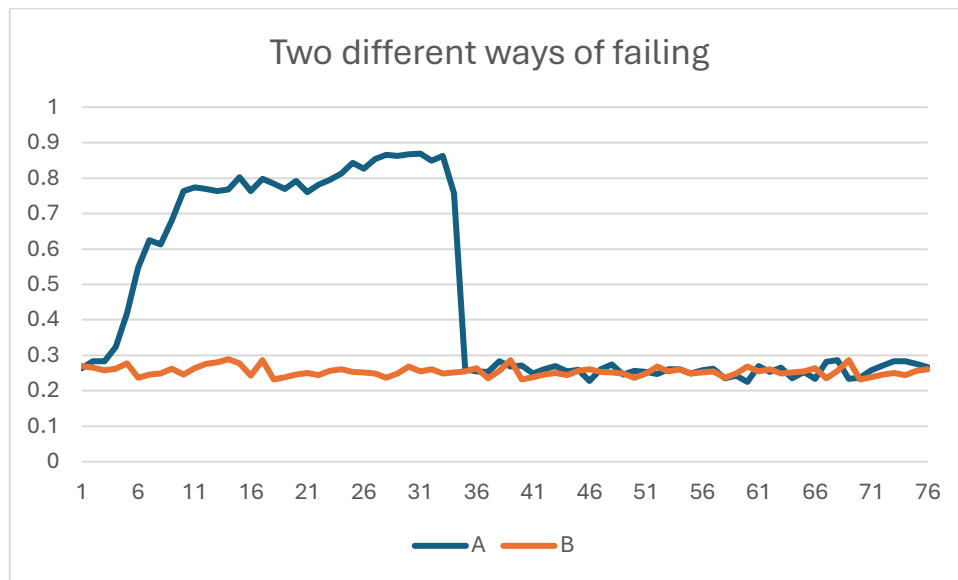


Figure 20: Two ways the algorithm can get stuck and stop converging. A learns at the start and then suddenly falls back to 0.25, B never improves. Both use the parameters described in Table 1.<sup>68</sup>

Case A happens very rarely, the algorithm starts improving, but at some point, it drops back down to 0.25 and stays there. Because it happened so rarely, it is hard to pinpoint the problem, but it is most likely due to an exploding gradient. This means that either the advantage or the partial derivative of a weight or a bias is bigger than it should be and one or multiple weights or biases are changed more than they should in a single update. This messes up the ANN and is very hard to recover from.

Case B happens more frequently, with these parameters: about 15% of all attempts are unable to improve at all and are stuck at 0.25 accuracy. This is most likely due to a bad set of weights after the initialization. When I refer to unsuccessful learning attempts in this paper, I will, unless specifically stated otherwise, refer to this specific case of never learning anything.

---

<sup>68</sup> See Appendix C

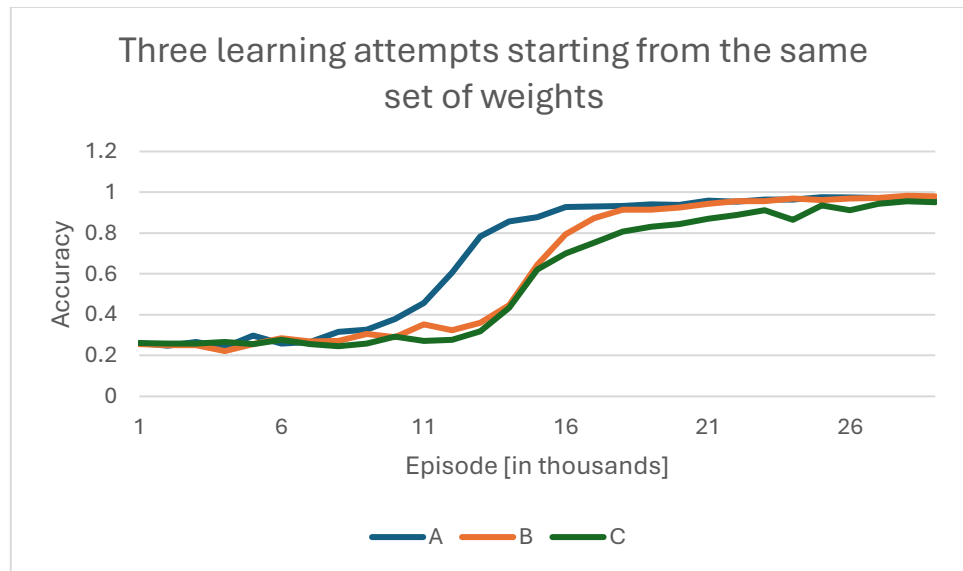


Figure 21: Three learning attempts that start with the same set of weights instead of each one starting with a unique set of randomly generated weights. This shows the effect variables other than initialization have on the training. Parameters as specified in Table 1.<sup>69</sup>

Figure 21 proves that the initialization of the weights has a big impact on the learning process of the algorithm. Compared to 19, the difference between the learning attempts in Figure 21, where they started with the same weights, is way smaller.

Figure 21 also indicates that the initialization of the weights is not the only variable that affects the training. While the graphs are closer in form than the ones in Figure 19, there is still a difference in how fast they converge. In my opinion this is due to the difference in training data. While the automatically generated training data should be about the same, it is not in the same order, which affects the training.

I repeated the test with sets of weights that were not successful in training: The results showed, that sets of weights that were unsuccessful once were never successful. But my sample size was very small, so these results could be incomplete.

#### 4.4.2 Size of ANNs

For these tests I decided to not separate the size into the number of hidden layers and the size of each hidden layer and instead look at both parameters at the same time. This still shows the relationship between the general complexity of the ANNs and the performance of the algorithm, but it probably hurts the optimization a bit.

<sup>69</sup> See Appendix B

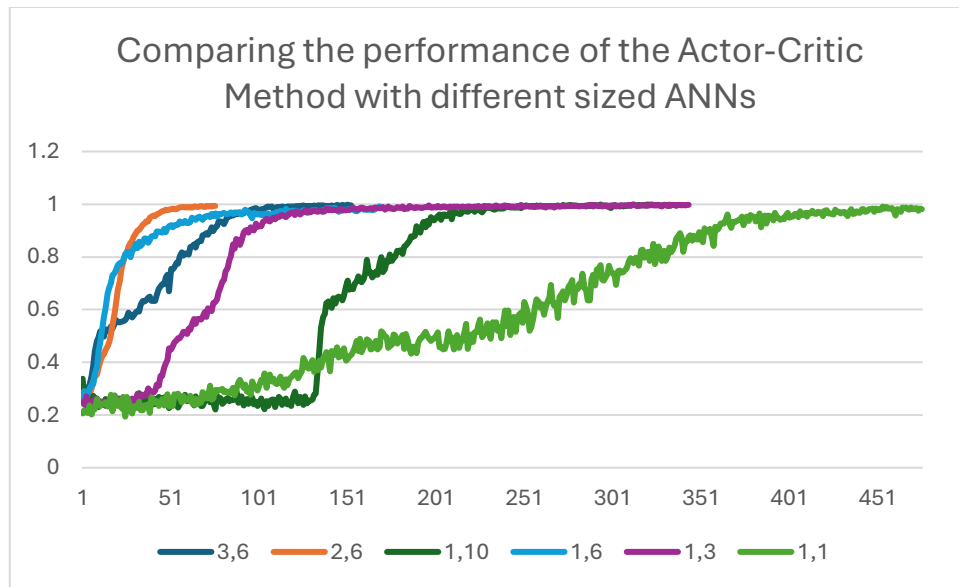


Figure 22: Average performance of multiple learning attempts with the same-sized ANNs. The name of each data set is  $x,y$ .  $X$  is the number of hidden layers and  $y$  is the number of neurons in each hidden layer, the rest of the parameters are the same as before. Only successful learning attempts are represented, unsuccessful attempts were discarded.<sup>70</sup>

<sup>70</sup> See Appendix E

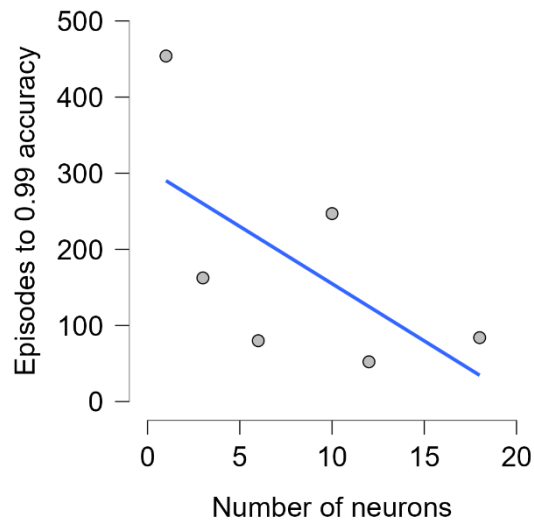


Figure 23: The number of episodes [in thousands] it took each size in Figure 22 to reach 0.99 accuracy versus the total number of neurons in all hidden layers.

Table 3: Kendall's correlations between the total number of neurons and the number of episodes [in thousands] needed to reach 0.99 accuracy and between the total number of neurons and the chance of success.<sup>71</sup>

#### Kendall's Tau Correlations

Variable		Number of neurons
1. Number of neurons	Kendall's Tau B	—
	p-value	—
2. Episodes to 0.99 accuracy	Kendall's Tau B	-0.467
	p-value	0.272
3. Chance of Success	Kendall's Tau B	0.414
	p-value	0.251

<sup>71</sup> See Appendix E

The trendline in Figure 23 shows a negative correlation between the total number of neurons in the hidden layers and the time it takes the algorithm to succeed. This means, that with added complexity, the time it takes the algorithm to succeed, is reduced. Table 3 shows that the p-value for this correlation is quite large, way above the baseline that would make it considered statistically significant.

#### *4.4.2.1 Hypothesis 1: There is a minimum size the ANNs need to be able to find an optimal solution.*

All the sizes down to 1,1 were able to find an optimal solution. Disappointingly, my algorithm is unable to create ANNs that do not have any hidden layers. Figure 22 proves that either there is not a minimal size necessary, and my hypothesis is incorrect, or the solution to *Pong* is so simple, that it can be solved with basically anything.

Literature suggests, that for every task there is a necessary minimal ANN size necessary. With this in mind, and since the solution to *Pong* is quite simple, I conclude, even though my test results in Figure 22 do not confirm this, that there is a minimum size the ANNs need to be able to find an optimal solution.<sup>72</sup>

#### *4.4.2.2 Hypothesis 2: Algorithms with this minimal size will learn faster than algorithms with larger ANNs.*

Figure 23 indicates a negative correlation between the size of the ANNs and the convergence time. Figure 22 shows that the ANN size with the fastest convergence time is 2,6, while the smaller ANNs often underperform in comparison. Especially 1,1 is slow, and the graph looks very unstable. I think the reason for the worsening performance with smaller ANNs is that they become unstable. The smaller they are, the more their graph goes up and down again instead of continuously improving. The data in Figure 22 clearly shows that the smallest working ANN size is not optimal. The p-value in Table 3 shows that the statistical significance of this test is quite low.

---

<sup>72</sup> Ho and Dinh, “Searching for Minimal Optimal Neural Networks.”

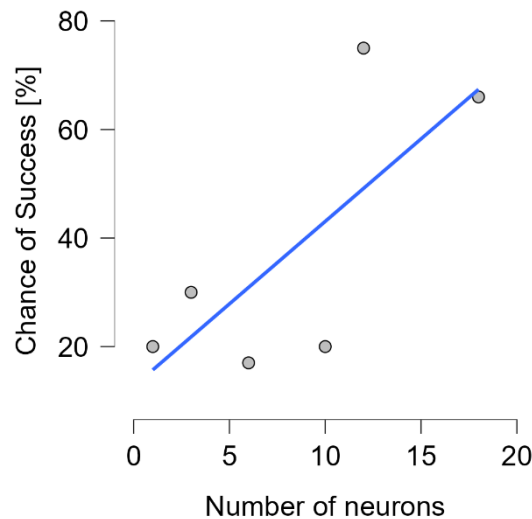


Figure 24: The chance of success for each size in Figure 22 versus the total number of neurons in the hidden layers.<sup>73</sup>

The trendline in Figure 24 states a positive correlation between the total number of neurons in the hidden layers and the chance of success. Table 3 shows that the p-value of this correlation is too large to be considered statistically significant.

I did not expect that the size of the ANN would influence the success rate of the algorithm. Figure 24 shows a positive correlation between the number of neurons and the chance of success. Again, the p-value in Table 3 is quite low, but the difference in success rates between algorithms with smaller and larger ANNs is significant.

### 4.4.3 Learning Rate in Reinforcement Learning

The learning rate controls how big the changes to the weights and biases are during gradient ascent. The Actor-Critic Method needs two learning rates, one for the agent and one for the critic. The learning rate of the critic should always be bigger than the learning rate of the agent. This is to ensure that the critic can react fast enough to changes in the actor. For all my tests, the learning rate of the critic is larger than the learning rate of the actor by a factor of ten. This is based on recommendations from online sources, and I decided that testing the effect of different ratios would take a lot of time and probably not have a significant impact.

---

<sup>73</sup> See Appendix E

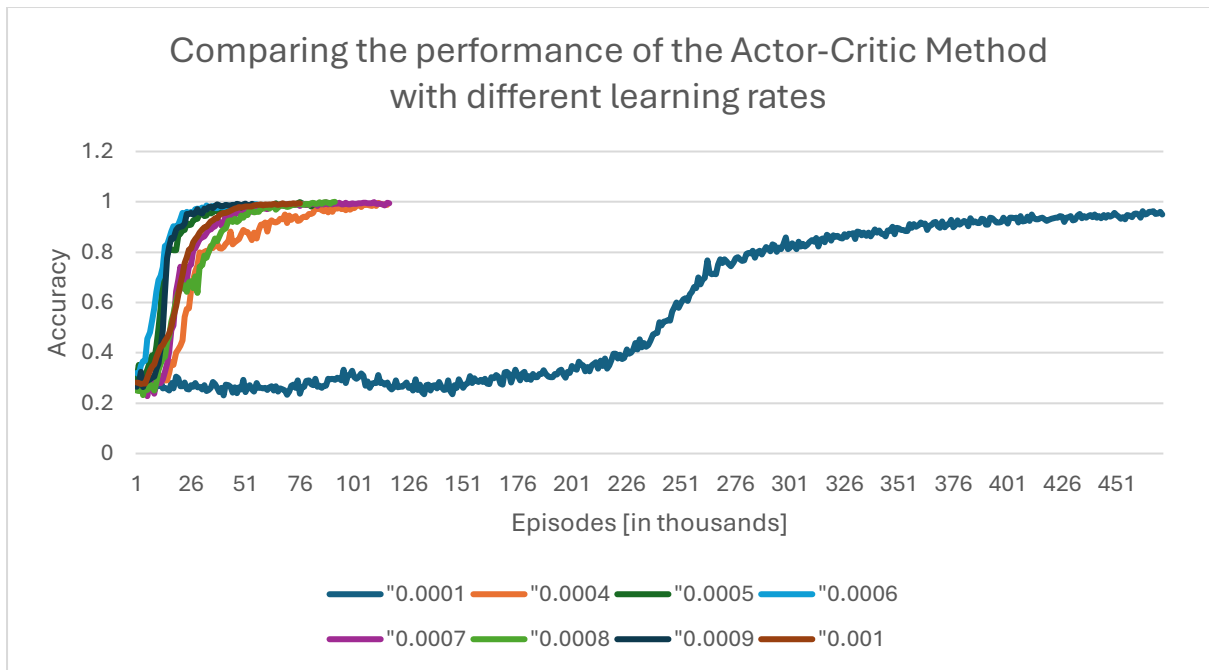


Figure 25: Performance of learning attempts with different learning rates for the actor and the critic. The name of each data set is "x, x is the learning rate of the critic. The learning rate of the actor is smaller by a factor of ten, so  $x/10$ .<sup>74</sup>

Table 4: The parameters used for all the learning attempts in Figure 25.

Parameters:	Value function	Policy function
Layer count	2	2
Layer size	6	6
Learning rate	x	$x/10$

<sup>74</sup> See Appendix A

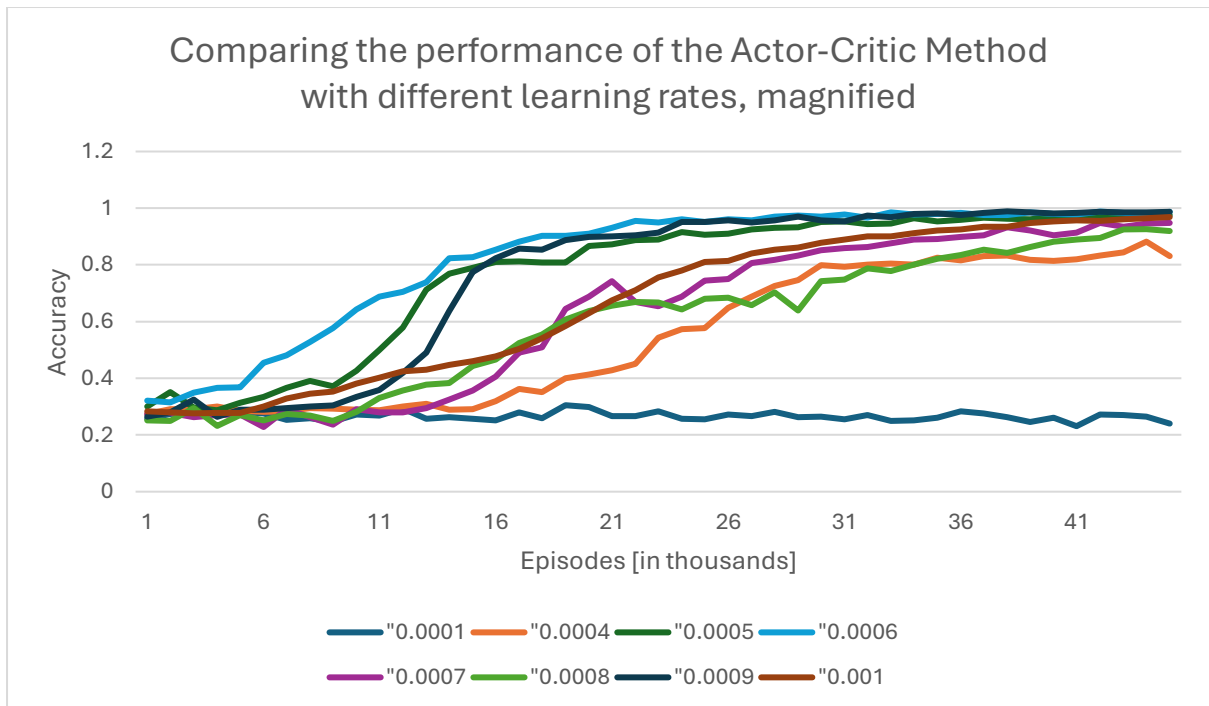


Figure 26: Zoomed in version of Figure 25 for readability.

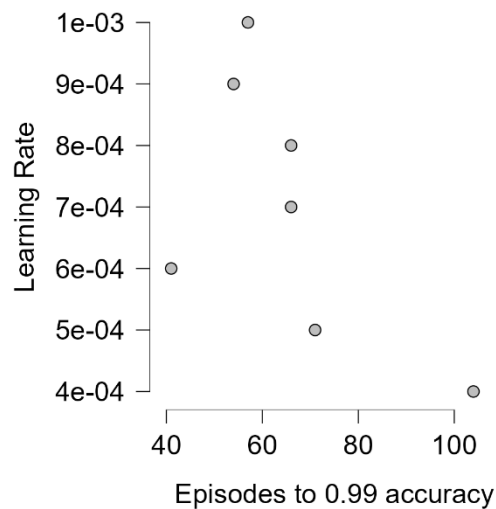


Figure 27: The number of episodes [in thousands] it took each learning attempt in Figure 25 to reach 0.99 accuracy versus learning rate. 0.0001 was left out because it made the graph unreadable.

Figures 25–27 show that the optimal learning rate is around 0.0006. Especially when the learning rate is lowered, the performance starts to decline rapidly. Higher learning rates also decrease in performance, but the effect is not as strong.



#### *4.4.3.1 Hypothesis 3: Small learning rates will lead to a flat curve and slow down the learning process.*

Figure 25 and Figure 27 show a decrease in performance with decreasing learning rates after a certain point. When compared to the best-performing learning rates, the smaller learning rates are very flat and take a long time to converge. This is due to the small updates, that need to accumulate to make a significant difference in performance.

#### *4.4.3.2 Hypothesis 4: Large learning rates can lead to a worse learning process.*

Figure 25 shows that the algorithms with the largest learning rates are not the best performing ones. I thought that this would be because of instability when each individual update is too big. Based on my sources, that is quite likely the reason why the algorithms with larger learning rates underperform, but my test results do not show this. Based on my test results, I can only conclude that large learning rates can lead to a worse learning process, but I cannot explain the reason for this behaviour.<sup>75</sup>

#### *4.4.3.3 Hypothesis 5: Too large learning rates can lead to complete failure of the algorithm.*

I did not include any failed learning attempts in my test results, except for Figure 20. They take up a lot of space and do not contain any relevant information. When the learning rate is increased too much, it results in a crash. It looks a lot like Case A in Figure 20, which absolutely makes sense. With a larger learning rate, the probability of an exploding gradient increases as well. This means, that if the learning rate is too large, it can lead to complete failure of the algorithm.

---

<sup>75</sup> Brownlee, "Understand the Impact of Learning Rate on Neural Network Performance - MachineLearningMastery.Com."

## 5 Conclusion

For this paper I had two objectives, to understand how algorithms like the Actor-Critic Method or the Genetic Algorithm work and try to implement them myself.

The Genetic Algorithm provided good results for both *Pong* and *Footsies*. Especially for *Pong* the performance was better than what I expected. It achieved near perfect accuracy and rivalled the performance of the Actor-Critic Method. However, the Genetic Algorithm demonstrated an advantage in handling steep ball angles due to its lack of reliance on action probabilities. For *Footsies*, the algorithm achieved moderate success, learning basic strategies but struggling with the evolving nature of self-play. The performance was therefore far from optimal, and it had problems learning anything complex and reacting to the enemy.

The Actor-Critic Method worked well for *Pong*, especially after the optimization of the parameters it was able to learn fast and achieve high accuracy quite consistently. The implementation for *Footsies* had more problems, the self-play mechanism it had originally used did not work as hoped. Modifications and training against a pre-programmed bot resulted in improvements, but it maintained to have problems with consistency and was only able to achieve a limited level of performance.

During the process of programming my algorithms and writing this paper I had two “phases” of learning. First, I had to learn enough about the topic to be able to write implement these algorithms from scratch, this required a deep understanding of how an Artificial Neural Network is structured, used and the algorithms needed to train it. Based on this theoretical basis and my experience in applying it I can confidently say that I understand how Artificial Neural Networks, and a lot of algorithms surrounding it work.

Afterwards I tested my implementations, turned the knobs that determine the details of how the algorithms work and took measurements. My tests showed that the solution to *Pong* is not only very simple, but also requires only a minimal ANN. If an ANN with such a limited size can solve task like *Pong*, the complexity of tasks larger ANNs, when trained perfectly, could solve, must be immense.

I thought it was perfectly logical, that the fewer variables a learning algorithm has to optimize, the faster it is able to optimize all of them. My results show that this assumption is not only wrong, but the contrary is happening. The more complex the ANN is the easier it can learn. For a biological being, this would make sense. The smarter something or someone is, the easier it is to learn something new. But an ANN is not intelligent, no matter how big it is. It is only a collection of optimized functions, it is not supposed to work like a truly intelligent being, so this still throws me off.

The effect the learning rate has on the performance matches my expectations. It has a big impact on the effectiveness of the algorithm, and I was able to find an optimal value to decrease the time it takes for the algorithm to learn.

I successfully developed AI systems capable of mastering *Pong*, demonstrating that it is possible to apply Artificial Neural Networks of minimal complexity to such a use case and still achieve almost 100% accuracy. While the algorithms for *Footsies* showed promise, further research and optimisation would be required to achieve human-level performance.

## 6 Acknowledgements

I thank my tutor Patrik Marxer for his guidance and support in writing this paper. I would also like to thank Adrian Hutter, who helped me get out of a rough spot during the programming of the practical project. I am grateful to my friend Damian Honegger for helping me with some of the statistical analysis. Finally, I would like to thank all the people who read my paper and helped me with their feedback.

## 7 Declaration of Authenticity

I declare that I completed this thesis independently and was only aided by the tools listed.

„Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und ohne Benützung anderer als der angegebenen Quellen oder Hilfsmittel verfasst bzw. gestaltet habe.“

Leon Rossi  
17.12.24



## 8 Reflection

Working on this project was a deeply rewarding yet challenging journey. The goal was to develop artificial neural networks capable of learning to play two games: Pong and a 2D fighting game, *Footsies*, using both a Genetic Algorithm and the Actor-Critic Method. While I achieved some success with Pong, the *Footsies* implementation proved to be a much more complex challenge. Reflecting on the process, several key insights stand out, both technical and personal.

One of the biggest challenges in the project was debugging and refining the algorithms. With ANN training algorithms, I had to wait for the results of the training attempt before I could judge if my changes fixed the problems or had the desired effect which could take hours. This led to the debugging taking a long time and pushing my writing back more and more to the point where I had to leave the *Footsies* project in a half-working state.

Despite its challenges, this project was an invaluable learning experience. It deepened my understanding of neural networks, Genetic Algorithms, and reinforcement learning techniques. I also gained a better appreciation for the balance between theoretical planning and practical implementation. The frustration of debugging, while often overwhelming, taught me resilience and the value of approaching problems systematically.

The experience also highlighted the importance of adaptability. When the *Footsies* algorithms failed to perform as expected, I had to shift the focus of my written work to what worked and decided to dive deeper into the topic of parameters. While this was stressful, it taught me to focus on prioritizing what was feasible within the constraints and to frame setbacks as opportunities to learn.

For the longest part of the project my time management was good. I was on top of deadlines set by myself or my tutor. After I got the feedback for my first draft, motivation was low, and my progress was slow. By the time I implemented all the feedback I got from the first draft, I only had around a week left and the whole reviewing process ended up rushed and the last few days before the deadline were very stressful. If I were to undertake a similar project again, I would leave more time for revisions.

Overall, this project taught me a lot, not only about the subject itself, but also about the challenges with bigger projects and the importance of time management.

## 9 Bibliography

AlphaStar team. "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II - Google DeepMind," January 24, 2019. <https://deepmind.google/discover/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii/>.

AnalytixLabs. "A Complete Guide to Genetic Algorithm — Advantages, Limitations & More | Medium," January 29, 2024. <https://medium.com/@byanalytixlabs/a-complete-guide-to-genetic-algorithm-advantages-limitations-more-738e87427dbb>.

Bendersky, Eli. "The Softmax Function and Its Derivative - Eli Bendersky's Website," October 18, 2016. <https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/>.

Brownlee, Jason. "4 Types of Classification Tasks in Machine Learning - MachineLearningMastery.Com," August 19, 2020. <https://machinelearningmastery.com/types-of-classification-in-machine-learning/>.

———. "A Gentle Introduction to the Rectified Linear Unit (ReLU) - MachineLearningMastery.Com," April 20, 2020. <https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/>.

———. "How to Choose an Activation Function for Deep Learning - MachineLearningMastery.Com," January 22, 2021. <https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/>.

———. "Understand the Impact of Learning Rate on Neural Network Performance - MachineLearningMastery.Com," September 12, 2020. <https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/>.

"Bubble Sort Algorithm - GeeksforGeeks," October 6, 2024. <https://www.geeksforgeeks.org/bubble-sort-algorithm/>.

Codecademy. "Binary Step Activation Function | Codecademy," July 16, 2023. <https://www.codecademy.com/resources/docs/ai/neural-networks/binary-step-activation-function>.

Comi, Mauro. "How to Teach an AI to Play Games: Deep Reinforcement Learning." <https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a>, November 2018.

Dawkins, Paul. "Calculus I - Chain Rule." Accessed December 11, 2024. <https://tutorial.math.lamar.edu/classes/calci/chainrule.aspx>.

DeepAI. "Vanishing Gradient Problem Definition | DeepAI." Accessed December 11, 2024. <https://deepai.org/machine-learning-glossary-and-terms/vanishing-gradient-problem>.

GeeksForGeeks. "Layers in Artificial Neural Networks (ANN) - GeeksforGeeks," July 19, 2024. <https://www.geeksforgeeks.org/layers-in-artificial-neural-networks-ann/>.

Goel, Shaurya. "Kaiming He Initialization. We Will Derive Kaiming Initialization... | by Shaurya Goel | Medium," July 14, 2019. <https://medium.com/@shauryagoel/kaiming-he-initialization-a8d9ed0b5899>.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <https://www.deeplearningbook.org/>.

HiFight. "FOOTSIES - HiFight," 2018. <https://hifight.github.io/footsies/>.

Ho, Lam Si Tung, and Vu Dinh. "Searching for Minimal Optimal Neural Networks," September 27, 2021. <https://arxiv.org/abs/2109.13061v1>.

IBM. "What Is Gradient Descent? | IBM." Accessed December 11, 2024. <https://www.ibm.com/topics/gradient-descent>.

Jander, Filip. "Programming a Mediocre Neural Network From Scratch." Accessed December 11, 2024. <https://www.filipjander.com/2019/02/programming-mediocre-neural-network.html>.

Kahn, Tanwir. "Reinforcement Learning – Exploration vs Exploitation Tradeoff - AI ML Analytics." Accessed December 11, 2024. <https://ai-ml-analytics.com/reinforcement-learning-exploration-vs-exploitation-tradeoff/>.

Katoch, Sourabh, Sumit Singh Chauhan, and Vijay Kumar. "A Review on Genetic Algorithm: Past, Present, and Future." *Multimedia Tools and Applications* 80, no. 5 (February 1, 2021): 8091–8126. <https://doi.org/10.1007/S11042-020-10139-6/FIGURES/8>.

Kim, Sung. "PyTorch Lecture 03: Gradient Descent - YouTube." Accessed December 11, 2024. <https://youtu.be/b4Vyma9wPHo?si=F-8kuvb4mfmMCTn2>.

Kinsley, Harrison, and Daniel Kukiela. "Neural Networks from Scratch in Python," n.d.

Kolebka, Lazare. "Developing an Elo Based, Data-Driven Rating System for 2v2 Multiplayer Games | Towards Data Science," September 6, 2023. <https://towardsdatascience.com/developing-an-elo-based-data-driven-ranking-system-for-2v2-multiplayer-games-7689f7d42a53>.

Lee, Mark. "2.3 Softmax Action Selection," January 4, 2005. <http://incompleteideas.net/book/ebook/node17.html>.

———. "6.6 Actor-Critic Methods," January 4, 2005. <http://incompleteideas.net/book/first/ebook/node66.html>.

MathsIsFun. "Partial Derivatives." Accessed December 11, 2024. <https://www.mathsisfun.com/calculus/derivatives-partial.html>.

Minsky, Marvin, and Seymour Papert. "Minsky-and-Papert-Perceptrons," 1969.

Mishra, Mohit. "The Curse of Local Minima: How to Escape and Find the Global Minimum | by Mohit Mishra | Medium," June 1, 2023. <https://mohitmishra786687.medium.com/the-curse-of-local-minima-how-to-escape-and-find-the-global-minimum-fdabceb2cd6a>.

NCL. "Numeracy, Maths and Statistics - Academic Skills Kit." Accessed December 11, 2024. <https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/core-mathematics/calculus/partial-derivatives.html>.

Nielsen, Michael. "Using Neural Nets to Recognize Handwrite Digits | NeuralNetsAndDeepLearning," December 2019. <http://neuralnetworksanddeeplearning.com/chap1.html>.

OpenAI. "OpenAI Charter | OpenAI." Accessed December 11, 2024. <https://openai.com/charter/>.

———. "OpenAI Five | OpenAI," June 25, 2018. <https://openai.com/index/openai-five/>.

OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębniak, et al. "Dota 2 with Large Scale Deep Reinforcement Learning," December 13, 2019. <https://arxiv.org/abs/1912.06680v1>.

Patel, Meet. "Understanding the Rectified Linear Unit (ReLU): A Key Activation Function in Neural Networks | Medium," April 20, 2024. <https://medium.com/@meetkp/understanding-the-rectified-linear-unit-relu-a-key-activation-function-in-neural-networks-28108fba8f07>.

"Pong - Play Game Instantly!" Accessed December 11, 2024. <https://freepong.org/>.

"Pong Game." Accessed December 11, 2024. <https://www.ponggame.org/>.

Schmidhuber, Juergen. "Annotated History of Modern AI and Deep Learning," December 21, 2022. <https://arxiv.org/abs/2212.11279v2>.

Simonini, Thomas. "Self-Play: A Classic Technique to Train Competitive Agents in Adversarial Games - Hugging Face Deep RL Course." Accessed December 11, 2024. <https://huggingface.co/learn/deep-rl-course/en/unit7/self-play>.

Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning : An Introduction*. The MIT Press, 2020.

Tattersall, Ian. "Charles Darwin and Human Evolution." *Evolution: Education and Outreach* 2, no. 1 (December 6, 2008): 28–34. <https://doi.org/10.1007/S12052-008-0098-8>.

"The Chain Rule," n.d. [www.mathcentre.ac.uk](http://www.mathcentre.ac.uk).

Walczak, Steven, and Narciso Cerpa. "Artificial Neural Networks." *Encyclopedia of Physical Science and Technology*, 2003, 631–45. <https://doi.org/10.1016/B0-12-227410-5/00837-1>.

Wood, Thomas. "Sigmoid Function Definition | DeepAI." Accessed December 11, 2024. <https://deepai.org/machine-learning-glossary-and-terms/sigmoid-function>.

YanisaHS. "Sigmoid Activation Function | Codecademy," July 7, 2023. <https://www.codecademy.com/resources/docs/ai/neural-networks/sigmoid-activation-function>.



## 10 Appendix

The raw data used for the statistical analysis in section 4 consists of very large tables that would only use unnecessary space in this paper. Instead, I decided to include an Excel file to properly represent the collected data. Each sheet in this Excel file is a different section of the appendix and its relevance will be explained below. All the appendices except for Appendix G are on tests of the performance of the Actor-Critic Method in *Pong*.

### 10.1 Appendix A

This table contains the accuracy of learning attempts over the course of their training. The values in each row represent the accuracy over the last thousand episodes.

The learning rate of the critic in each learning attempt is in the name, the learning rate of the policy is equal to the learning rate of the critic divided by ten.

### 10.2 Appendix B

This table contains the accuracy of learning attempts over the course of their training. The three learning attempts that all started with the same weights. The values in each row represent the accuracy over the last thousand episodes.

### 10.3 Appendix C

This table contains the accuracy of learning attempts over the course of their training. Both learning attempts failed, one failed after improving for a while, the other failed right from the start and never managed to learn anything. The values in each row represent the accuracy over the last thousand episodes.

### 10.4 Appendix D

This table contains the accuracy of learning attempts over the course of their training. All of them have exactly the same parameters and varied starting weights as normal. The values in each row represent the accuracy over the last thousand episodes.

## 10.5 Appendix E

This table contains the accuracy of the average learning attempts for different sizes over the course of their training. It uses the learning attempts from Appendix F and Appendix D for those averages. The names consists of “x,y. x is the number of hidden layers and y is the number of neuron per hidden layer. The values in each row represent the accuracy over the last thousand episodes.

## 10.6 Appendix F

This table contains the accuracy of learning attempts over the course of their training for different sized ANNs. The names consists of “x,y. x is the number of hidden layers and y is the number of neuron per hidden layer. The values in each row represent the accuracy over the last thousand episodes.

## 10.7 Appendix G

This table contains the accuracy of a learning attempt of the Actor-Critic Method in *Footsies* to show an example of how the Actor-Critic Method performs in *Footsies*. The values in each row represent the accuracy over the last thousand episodes.