TRAINING FEEDFORWARD NEURAL NETWORKS

FEEDFORWARD NEURAL NETWORKS: EFFICIENCY AND PERFORMANCE OF BACKPROPAGATION AND EVOLUTIONARY ALGORITHMS

Kasper van Maasdam

Thesis supervisor
T. Lokhorst
Department of Mathematics

Stedelijk Dalton Lyceum Overkampweg
Dordrecht, Zuid-Holland, Netherlands

2021-10-13

ABSTRACT

Artificial neural networks are important in everyday life and are becoming more widespread. For this reason, it is crucial they are understood and tested. This paper tests and compares two training methods: reinforcement learning with backpropagation and an evolutionary method. The hypothesis is that the training method using backpropagation and reinforcement learning is more efficient in training a neural network to play a game than a model trained with the evolutionary algorithm. However, the model trained with backpropagation and reinforcement learning will have lower performance than a model trained with the evolutionary algorithm. To research the hypothesis, a feedforward neural network and how it works must first be explained.

Neural networks are systems inspired by the biological brain which enables a computer to predict, model, classify and many other applications. All this by learning from some set of training data to find general relations that can be applied to unseen data. A neural network model is essentially a function with potentially thousands of parameters. Just like any other function, input values are provided and with those, the output is calculated. In a feedforward neural network, this process is called feedforward.

The process of feedforward is meaningless with a model that has not yet been configured to do anything. A neural network must first be taught to perform a certain task. This is what is accomplished with machine learning. Backpropagation is an example of a machine learning method. For backpropagation two things are required: the input and the corresponding output. Backpropagation will adjust the parameters of a model so the next time the same input is provided, the output will be closer to the desired output. This is called optimisation.

Reinforcement learning is a way to teach a neural network by giving it positive reinforcement when it does something good and negative reinforcement when it does something bad. This is used when no desired output is known so backpropagation cannot directly be applied.

An evolutionary algorithm is much more intuitive than backpropagation. It is the imitation of natural selection in biology, but with self-determined factors deciding the fitness of a model. When training a neural network with an evolutionary algorithm, a large group of random models will be generated, all performing the same task. Some models, however, will be better suited for this task than others. How well they are suited to their environment is their fitness. This will be the determining factor of who survives and can therefore reproduce and create mutated offspring. This process is repeated as many times as required to reach the desired performance.

The hypothesis of this paper has been proven wrong. Neural networks trained with an evolutionary algorithm do end up performing at a higher level than models trained with reinforcement learning and backpropagation. However, Neural networks trained with an evolutionary algorithm are also more efficient with regard to not only the number of cycles needed to reach the same performance but also with regard to the time required.

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor T. Lokhorst for his dedicated support. He has helped me complete this research paper with his insightful assistance and guidance.

Because of the many problems I have had to overcome, I have developed not only as a student but also as a person with a different view on life in the process of writing and researching this paper. It has helped me approach problems more thoughtfully and to remain patient when things don't work the way I want.

A debt of gratitude is also owed to my dear friend and classmate Evan Pacini. He has expressed a great interest in my work and has encouraged me to fully explore and develop a thorough understanding of neural networks.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

Artificial neural networks are of increasing importance in not only scientific research but also in everyday life. Therefore, we must develop a thorough understanding of the implementation of these neural networks, how they work and how they learn. This paper will examine, test and compare two ways of training feedforward neural networks. Namely backpropagation with reinforcement learning and an evolutionary algorithm. The former involves numerous complicated mathematical operations. The evolutionary algorithm, however, is much more intuitive and understandable. These two methods have been chosen because they differ greatly.

Inspired by Google's DeepMind, the neural networks will be taught to play a game. This way, the training methods can be compared by their game playing performance.

The expectation is that the training method using backpropagation and reinforcement learning is more efficient in training a neural network to play a game than a model trained with the evolutionary algorithm. However, the model trained with backpropagation and reinforcement learning will have lower performance than a model trained with the evolutionary algorithm. However, before this can be examined, firstly the question of what a neural network is, must be answered. Also, the process of backpropagation, reinforcement learning and evolutionary algorithms must be explained.

2 MOTIVATION

Neural networks have been my interest for a long time. I used to hear a lot about it on the TV, but I never understood how a computer program could learn to do things no other hand-made algorithm could do. It always seemed very complicated and far above my level of understanding. In this paper, I wanted to learn and explore the mathematics behind a neural network and backpropagation. In addition, I wanted to explore other ways of training a neural network. Other, more intuitive ways. That is why I chose the evolutionary method. It is something I had learned about in my biology class and I already envisioned how this method could be applied to neural networks.

I had once before tried to develop a neural network. At the time, however, I did not know anything about it. It was doomed to fail. So, this is my second attempt, a fresh start to do it right.

3 NEURAL NETWORKS

Neural networks are systems inspired by the biological brain which enables a computer to predict, model, classify and many other applications. All this by learning from some set of training data to find general relations that can be applied to unseen data. Image and speech recognition, text generation, text summarization and real-time route planning are examples of what computers can de because of neural networks.

Neural networks can be visualised as nodes connected by lines. There are many types of neural networks, but this paper focuses on feedforward neural networks. This means that nodes do not form a cycle. In feedforward neural networks, the nodes are ordered in vertical layers parallel to each other. Every node in a certain layer is connected to every node in the layers to its left and its right (see Figure 1). The first layer is the input layer and this is where data gets inserted into the model that needs to be processed. The processing is done in the hidden layers. These are often considered black boxes. Once the data has been put through all the hidden layers, it

is put into the output layer; notice that the number of input nodes does not need to be equal to the number of output nodes. This is the output of the model. The hidden layers are all layers between the input and the output layer.

When taking a closer look at the image, it becomes visible there is more to it than just nodes and lines. Each line represents a weight. These weights are parameters. They are what determine the influence of each node throughout the process. That is why they are called weights. Another important element of the network is the biases. Every node contains a bias. These are also parameters that influence the outcome of the model.

It is very important to differentiate each node, weight and bias. Therefore, accurate notation is crucial. To get a better understanding of the notation used in this paper, the following list is provided:

$$weights = w$$
$$biases = b$$
$$node\ values = \lambda$$
$$number\ of\ nodes\ in\ a\ layer = N_l$$
$$total\ number\ of\ layers = L$$
$$index\ used\ to\ specify\ a\ layer = l$$
$$index\ used\ to\ specify\ a\ node = j$$
$$index\ used\ to\ specify\ a\ weight = k$$

To now differentiate between layers, a superscript is used (N is a special case, as it never needs subscripts to differentiate a node). $w^{(2)}$, $b^{(2)}$, $\lambda^{(2)}$ and N_2 would give the weights, the biases, the node values and the number of nodes of the second layer respectively. $w^{(L)}$, $b^{(L)}$, $\lambda^{(L)}$ and N_L would give the weights, the biases, the node values and the number of nodes of the last layer (the output layer) respectively. Each of these in its general form can be specified as such: $w^{(l)}$, $b^{(l)}$, $\lambda^{(l)}$ and N_l.

To differentiate one step further, subscripts are used. These specify individual nodes in the layer specified in the superscript. $w_1^{(L)}$, $b_1^{(L)}$ and $\lambda_1^{(L)}$ would give the weights, the bias and the node value of the first node of the output layer respectively. $w_{N_L}^{(L)}$, $b_{N_L}^{(L)}$ and $\lambda_{N_L}^{(L)}$ would give the weights, the bias and the node value of the last node of the output layer respectively. Each of these in its general form can be specified as such: $w_j^{(l)}$, $b_j^{(l)}$ and $\lambda_j^{(l)}$.

There is one further step of differentiating. This is only for the weights. Because weights are connections between layers and each node of a layer is connected to all the nodes of the layer to its left, specifying a layer and a node will still provide an array of values. This is why it is required to differentiate the nodes of the layer to its left to specify a singular weight. This is done as such: $w_{jk}^{(l)}$. $w_{2\,5}^{(L)}$ would refer to the weight between the second node of the last layer and the fifth node of the layer to its left.

Nota bene. The first layer has no biases and there is no layer to its left with nodes to which weights can connect. This means $w^{(1)}$ and $b^{(1)}$ are non-existent. $\lambda^{(1)}$ and N_1, however, do exist as $\lambda^{(1)}$ would just refer to the input values and N_1 to the number of input nodes.

Something else to keep in mind is that w, b, λ and N are matrices and behave accordingly. When specifying l and j with w, and l with b and λ, they still return a vector. This can be shown as follows:

$$w^{(l)} = \begin{pmatrix} w_{11}^{(l)} & w_{12}^{(l)} & \cdots & w_{1k}^{(l)} \\ w_{21}^{(l)} & w_{22}^{(l)} & \cdots & w_{2k}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{j1}^{(l)} & w_{j2}^{(l)} & \cdots & w_{jk}^{(l)} \end{pmatrix} \qquad b^{(l)} = \begin{pmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_j^{(l)} \end{pmatrix} \qquad \lambda^{(l)} = \begin{pmatrix} \lambda_1^{(l)} \\ \lambda_2^{(l)} \\ \vdots \\ \lambda_j^{(l)} \end{pmatrix}$$

$$w_j^{(l)} = \begin{pmatrix} w_{j1}^{(l)} & w_{j2}^{(l)} & \cdots & w_{jk}^{(l)} \end{pmatrix}$$

3.1 THE FEEDFORWARD FUNCTION

A neural network model is essentially a function with potentially thousands of parameters. Just like any other ordinary function, input values are provided and with those, the output is generated. This process is called feedforward. This word describes the feeding of the model with information which is then moving forward from one layer to the next, each transition modifying the values according to the weights and biases until it reaches the output layer.

First, there is one more step to introduce: the activation function. This function is added to help the network learn complex patterns in the data. It takes in the output from the previous node and converts it into some form that can be taken as input to the next node (Jain, 2019).

When input data is provided, the first thing that is calculated is the values of each node of the second layer. To calculate the value of a singular node j in the second layer, the following steps are taken:

1. The sum of all the input values multiplied by their corresponding weight is calculated. This sum can be simplified with the help of the dot product of the weights vector and the input layer vector.

$$\sum_{k=1}^{N_1} (w_{jk}^{(2)} \times \lambda_k^{(1)}) = w_j^{(2)} \cdot \lambda^{(1)}$$

2. Before putting the sum through the activation function, first, the bias of node j is added. This results in an intermediate, represented by z.

$$z_j^{(2)} = w_j^{(2)} \cdot \lambda^{(1)} + b_j^{(2)}$$

3. z is now put through the activation function. This results in the final value of node j.

$$\lambda_j^{(2)} = Act(z_j^{(2)})$$

This is for just one value, but using matrix multiplication, z can be calculated for an entire layer with only one operation:

$$z^{(2)} = w^{(2)} \cdot \lambda^{(1)}$$

Notice how j is no longer specified. This is because $z^{(2)}$ is a vector containing all the values for the second layer. Note that some activation functions such as the softmax function require this z vector instead of a single z_j.

With the layer that has just been calculated, the next layer can be calculated in the same way. This process repeats until the values of the output layers have been calculated.

4 MACHINE LEARNING

The process of feedforward is meaningless with a model that has not yet been configured to do anything. A neural network must first be taught to perform a certain task. This is what is accomplished with machine learning. There are many methods of machine learning. These methods can be split up into two categories: Supervised Learning (SL) and Unsupervised Learning (UL). The methods that will be discussed in this paper are backpropagation, reinforcement learning and evolutionary algorithms. Backpropagation requires some advanced and intermediate mathematical operations that require some foreknowledge. The evolutionary algorithms are much more apprehensible and intuitive, but less efficient. Which method, however, is faster in practice?

4.1 BACKPROPAGATION

Backpropagation is an example of a supervised machine learning method. For backpropagation two things are required: the input and the corresponding output. Backpropagation will adjust the weights and biases in such a way that the next time the same input is provided, the output will be closer to the desired output. This is called optimisation.

For a model to know how good or bad a certain output is, a cost function is used. This is a function that takes the output of a model and the desired output of each node and returns a cost. Each problem has a fitting cost function. In backpropagation, an error function is used as the cost function.

The principal idea behind backpropagation is finding out how much each weight and bias influence the final cost. Because the entire neural network can be seen as a composite function where the output of a previous layer is the input of the next, finding the influence of each weight can be accomplished with the chain rule by taking the partial derivative.

To find out how the cost function changes in relation to the weights and the biases, the gradient of the cost function with respect to the weights and the biases need to be calculated. Imagine a cost function C and some weight $w_{jk}^{(l)}$. These two quantities are not directly proportionate to each other. This is where the chain rule comes in.

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial \lambda_j^{(l)}} \times \frac{\partial \lambda_j^{(l)}}{\partial z_j^{(l)}} \times \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

Note that

$$C = Error(\lambda^{(L)})$$

$$\lambda_j^{(l)} = Act\left(z_j^{(l)}\right)$$

$$z_j^{(l)} = w_j^{(l)} \cdot \lambda^{(l-1)} + b_j^{(l)}$$

When looking carefully at those definitions, the capital L should be noticed. This indicates that the cost function is directly proportionate to the output layer. Also, it should be noted that $w_{jk}^{(l)}$ is part of $w_j^{(l)}$ so that when taking the partial derivative,

$$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \lambda_k^{(l-1)}$$

When the change in the cost function in relation to that weight is calculated, the weight is updated. This is the optimisation process and can be described as such:

$$w_{jk}^{(l)} \coloneqq w_{jk}^{(l)} - \alpha \times \frac{\partial C}{\partial w_{jk}^{(l)}}$$

Here α is the learning rate. This is a hyperparameter: a parameter whose value is used to control the learning process.

As mentioned above, the change in C is directly proportional to the change in $\lambda^{(L)}$. This means that in cases where $l \neq L$, $\frac{\partial C}{\partial \lambda_j^{(l)}}$ cannot be calculated in one step because then, C is not directly proportional to the change in $\lambda^{(L)}$. To calculate $\frac{\partial C}{\partial \lambda_j^{(l)}}$, would be to calculate the change in cost in relation to the change of a node value in a certain layer. To calculate this value for any node in a layer where $l < L$, it would need to start at layer L and propagate back to layer l. This is where the name comes from.

Another way of describing the change in cost in relation to the change of a node j in layer l would be with a question: How much do all the nodes in layer $l + 1$ think node j in layer l should change? Answering this question would result in a sum. A sum of all the 'opinions' of the nodes in the layer to its right. This sum would look like this:

$$\frac{\partial C}{\partial \lambda_j^{(l)}} = \sum_{n=1}^{N_{l+1}} \left(\frac{\partial C}{\partial \lambda_n^{(l+1)}} \times \frac{\partial \lambda_n^{(l+1)}}{\partial z_n^{(l+1)}} \times \frac{\partial z_n^{(l+1)}}{\partial \lambda_j^{(l)}} \right)$$

Mind the indices n and j. They are very important. The n indicates a specific node in the layer to the right $(l + 1)$. While n iterates over every node in layer $l + 1$, j remains constant. This way each node in the layer to the right has a saying in how node j should change.

This derivative, however, can also not be calculated directly when $l < L - 1$, because the function would still have to know how each of the nodes in the layer to the right should change. Only then can they tell how node j should change. Following this logic, $\frac{\partial C}{\partial \lambda_j^{(l)}}$ appears to be a recursive function starting from layer L, the output layer, and ending at node j in layer l.

Keep in mind that the same steps are taken when calculating the change of the cost function in relation to the biases. Only then, $\frac{\partial C}{\partial b_j^{(l)}}$ is calculated instead of $\frac{\partial C}{\partial w_{jk}^{(l)}}$.

4.1.1 ACTIVATION FUNCTIONS

There are many types of activation functions. Each activation function influences a neural network differently, but ultimately, four factors should be considered. Because of how backpropagation works, the most important requirement for an activation function is that it needs to be differentiable. Another factor is the computational expense. Functions including exponents, logarithms or other intensive operations generally slow down the entire training process. Functions with a range between zero and one or minus one and one can cause the gradient of the first couple layers of larger models to vanish and it can cause the activation to shift towards zero. This phenomenon is called the vanishing gradient problem. Another problem is the shifting of the gradients to a certain offset. This happens if an activation function is not zero-centred.

In this paper, the last two factors are not taken into account as the models that will be used, are not big at all.

4.1.2 ITERATIONS

The process of backpropagation is specific. It calculates precisely how the weights and biases need to change to reduce the cost. This is very useful and makes fitting and interpolating data points[1] easy and efficient. However, after having performed backpropagation and having optimized the parameters, the neural network is merely one step closer to its optimum where cost is at its lowest. The process must be iterated many times. The number of iterations is very important and should be taken into consideration. Too many, and something called overfitting might occur. Too little, and underfitting might occur.

Overfitting occurs when a model fits exactly against its training data. This way, the algorithm cannot perform accurately anymore against unseen data, which is why it is trained in the first place. Overfitting is a result of too many iterations of training, too specific input data or a model too complex for its job.

Underfitting, on the other hand, occurs when a model cannot find a relation between the input and the output data. This way there is a high cost when performing against the training data as well as unseen data. This is a result of too few iterations, too insufficient input data or a model that is too simple.

The generalization of new data is why machine learning algorithms are used every day in making predictions and classifying data.

4.1.3 SHAPE

See Figure 1. The shape of the model determines how many layers a neural network has. It also determines how many nodes each layer has and how layers and nodes are connected. Big, complex models include many parameters and are therefore more time-consuming to train. While simply adding more nodes to a layer will slightly increase the number of connections between nodes and therefore increase the ability of a model to find more complex relations between the input and the output, adding a layer to a model creates many more connections, greatly increasing the number of parameters present in the model.

A layer can also be a limiting factor in the model. Take for example a model with five input nodes, a hidden layer of one node, a hidden layer with twenty nodes and an output layer. First,

[1] See Figure 2 for an example.

all the information stored in the input layer gets compressed into one node; loss of data will occur. Next, the data is then expanded over twenty nodes. Despite the model being more complex including the first hidden layer of one node, if that first layer was excluded, the loss of data would not have occurred, resulting in a model more capable of interpreting the input data.

Great care should be taken when deciding on the shape of a neural network, though the process of doing so is hard and complex.

4.1.4 Learning Rate

When applying backpropagation, the learning rate is very important. This can make or break the training process. If the learning rate is too high, the model will not reach its optimum lowest cost because the steps taken to reach this minimum are too large. The opposite can also happen. If the learning rate is too low, it will take many more iterations than necessary for the model to be trained sufficiently. Compare Figure 2 and Figure 3 to see how the learning rate has influenced the results after ten thousand iterations. In these figures, backpropagation is used to teach two identical feedforward neural networks to interpolate the training data. The only difference between the two processes is the learning rate. Yet the performance is very different. It is visible that the second model is going in the right direction, but it will take much longer before it reaches the same performance as the first model. This is because of the smaller step size that is taken each iteration to approach the cost minimum.

4.2 Reinforcement Learning

Fitting training data with backpropagation is great for teaching a neural network to better predict according to input data. However, backpropagation requires input *and* the corresponding output. Does this mean it is useless when the corresponding output data is not known? No, but it needs some help. This is where reinforcement learning comes in. Reinforcement learning is a way to teach a neural network to do more complex operations by giving it positive reinforcement when it does something good and negative reinforcement when it does something bad.

One example where reinforcement learning can be applied is when training a model to play a game. With most games when a prediction is made by the AI, it is unknown if it was a good action or a bad action. This is only known after the model is awarded positive or negative feedback – when it has earned a point for example. The crucial part of reinforcement learning is that all inputs and corresponding predictions are saved until the AI is reinforced (backpropagation is applied). If the reinforcement is positive, the model should be trained to be more likely to do the same by applying backpropagation to the model with all the saved inputs and outputs as training data. If the feedback is negative, the model should be trained to be less likely to make the same predictions.

4.3 Evolutionary Algorithms

An evolutionary algorithm is much more intuitive than backpropagation. It is the imitation of natural selection in biology, but then with self-determined factors determining the fitness of a model.

When training a neural network with an evolutionary algorithm, a large group of random models will be generated. All these models will perform the same task. Some models, however,

will be better suited for this task than others. How well they are suited to their environment is called their fitness. This can be calculated any way and will be the determining factor of who will survive and who will not.

When all models have finished, they will be put in a list ordered by fitness, from high to low. Then it should be determined what percentage will survive. Say 20%, then the top 20% will survive and the other 80% will be discarded. They are replaced by clones of the survivors, but each with random mutations. This way, the survivors remain in the model pool, ensuring that the performance of the pool cannot decrease. This new pool is then tested on how well they perform.

This process is repeated as many times as it takes to get a model that meets the requirements. This happens by natural selection. When by chance one of the mutated models ends up being better than the previous top models, it will survive and be used to generate new mutated models. If repeated enough times, the top models will become fitter and fitter. This is generally how bacteria are considered to evolve and for example, become more resilient to certain antibiotics.

Another way of approaching evolutionary algorithms is by generating new models by mating, instead of cloning. This way, the genes of the top models will be shared amongst all models. This is more representative of how animals evolve. Mutations can still occur, but this is not the only way how offspring can differ from each other. Which genes are taken from the father and which from the mother will also be random and create unique children.

In this context, genes can be interpreted as weights and biases.

A big disadvantage of evolutionary algorithms is the computational expense required to run all of the models to perform the same task. On the other hand, the many models make a reasonable solution against the problem of being unable to find the global minimum.

4.4 Local and Global Minima

Getting the lowest cost is the goal of a neural network. However, is it even possible to get the lowest cost in an environment? Generally, using backpropagation, only a local minimum can be found.

When a random model is generated, it has a certain cost. This cost can then be reduced or increased. By performing for example backpropagation, the weights and biases are changed to decrease the cost. This change makes the model a different model with a different cost. When plotting all possible models and their costs on a graph, a landscape will be seen with bumps and holes. When a model has improved a little bit, a step is taken downwards towards one of the many holes. No matter how much this model is improved by backpropagation, the lowest the model can go is as low as the hole. This hole, however, might not be the lowest hole there is. The model has then reached its local minimum.

Somewhere on the landscape, there is a hole that is lower than all other holes. This is the global minimum. So when training one random model, the chance is very slim that this model is on a path down the global minimum. Nearly always, it will reach a local minimum.

The advantage of evolutionary algorithms is that not only one model is trained, but perhaps many hundreds. This means that the result is not dependant on one path, but many paths are going to local minima. One of these paths is then the lowest of all. This model is then the fittest.

5 RESEARCH

The hypothesis of this paper describes the training method using backpropagation and reinforcement learning to be more efficient in training a neural network to play a game than a model trained with the evolutionary algorithm. However, the model trained with backpropagation and reinforcement learning will have a higher performance than a model trained with the evolutionary algorithm. To examine which of the two methods is superior, a test game has been made and a program has been made which describes the neural network and its capabilities. The code can be accessed here: https://github.com/KaspervanM/PWSFeedforwardNeuralNetworkPong

5.1 THE GAME

The game made to test the capabilities of the learning algorithms is Pong. Pong is a PvP[2] game involving two paddles hitting a ball (see Figure 4). One paddle is located on the left side of the screen and one paddle is located on the right side of the screen. The paddles can only move in the direction of the y-axis. The goal of the game is to hit the ball past the paddle of the opponent. If a ball passes a paddle, a point is awarded to the player controlling the opposing paddle.

The game is made using Python 3.9 with a library called `Pygame` and is based on an online tutorial (101 Computing, 2019). In addition to the game, the Python code also contains an interface to use the neural network. For this, the Python library `ctypes` is utilised.

When the ball hits the ceiling or the floor, it bounces while preserving its speed. When it bounces, the x-velocity remains the same, but the y-velocity is multiplied by -1. Upon hitting the paddle, the x-coordinate of the ball is multiplied by -1. However, if the paddle is moving, the y-velocity increases with the speed at which the paddle was moving. In addition, even if the paddle is stationary, there is a chance of a small random offset in the y-velocity of the ball. This was added to create an increased amount of variety in the game with the same inputs.

The game has a standard framerate of sixty frames per second. This means all objects are updated and all other game logic is calculated once every sixtieth of a second. This is a speed that looks natural for humans. However, the game can go faster. This is useful for running tests. For this reason, the function limiting the framerate to sixty frames per second can be toggled on or off. Depending on the speed of the computer and how much needs to be calculated, a normal game of pong can run at around two or three thousand frames per second. There is still a limiting factor, however. The game still shows every frame in the game window. The printing of a frame can happen quickly, but nowhere near as quick as required to keep up with the possible framerate. This then drops the framerate. To fix this problem, the printing of the screen has been made toggleable.

In the end, Pong is a simple game perfect for testing the capabilities of a neural network. It is easy to make and can therefore easily be adapted if changes need to be made during the testing and development of the neural network training algorithms.

[2] Player versus Player

5.2 THE NEURAL NETWORK

The neural network and the algorithms to use and train it are made in C++[3]. The C++ code is compiled into a DLL[4] with a C interface using Microsoft Visual Studio 2019. The C-interface is required to interface with the `ctypes` library in Python. The neural network is programmed in C++ instead of Python because C++ is much faster than Python. Training a model is therefore much less time-consuming.

The DLL is composed of four parts:

1. The `Model` class
2. The `ModelFunctions` class
3. The `TrainingData` class
4. The `ModelStorage` class

To create a `Model` object, a `modelArg` object must be passed to the class constructor[5]. This object contains the arguments for the model: the seed, the shape, and the activation functions. The shape and the activation functions are stored in a vector[6]. The `Model` class has four attributes: `weights`, `biases`, `shape` and `actFuncts` (activation functions). Attributes `shape` and `actFuncts` are set in the class constructor. The weights and biases are randomly generated based on the seed[7]. The `Model` object has two methods: one to set `weights` and `biases` and one to mutate the model.

The method to mutate the model works by changing the weights and biases according to the mutation rate and mutation degree. The mutation rate determines the chance of a weight or bias to mutate and the mutation degree determines the size of the mutations. When a weight or bias is mutated, a random number is added generated by normal distribution according to a given seed with its mean set to 0 and its standard deviation to the mutation degree. Any mutation can therefore be positive or negative.

The other functions using the model are methods in the `ModelFunctions` class. These functions include:

- The `activation` method
- The `activation_derivative` method
- The `feedforward` method
- The `cost` method
- The `cost_derivative` method
- The `backpropagation` method
- The `optimize` method
- The `train` method

[3] C++ is an object oriented general purpose programming language and, except for minor details, a superset of the C programming language. (Stroustrup, 1997)
[4] Dynamic Link Library
[5] A class constructor is a special member function of a class that is executed whenever we create new objects of that class. (Tutorialspoint, n.d.)
[6] In C++, vectors are sequence containers representing arrays that can change in size. (cplusplus.com, n.d.)
[7] Random number engines generate pseudo-random numbers using seed data as entropy source. (cppreference.com, n.d.)

All of the methods are static and can therefore be used without the need for a `ModelFunctions` object. The `activation` and `activation_derivative` method include many activation functions: the `identity` function, the `binary` activation function, the `sigmoid`, `tanh`, `ReLU`, `leaky_ReLU`, `ReLU6`, `leaky_ReLU6`, `softplus`, `softsign`, `gaussian`, `swish`, `hard_swish`, `fastSigmoid`[8] and `softmax` activation function. The functions can handle a vector of values or a single value. This way, an entire layer can easily be calculated. Keep in mind: The first two activation functions do not work with backpropagation.

The `feedforward` function calculates the values of all the nodes by forwards propagation as explained in 3.1. Then, the output layer is returned. The `cost` and `cost_derivative` methods include only one activation function. This is the Mean Squared Error (MSE) function (Pai, 2020)

The last three methods mentioned above are what train the neural network. The `train` function controls the backpropagation and optimisation. It repeats the process as many times as the user requires. The number of iterations is passed as a parameter. The function also requires a `TrainingData` object. This object contains the training data split up in batches (Figure 5).

When performing backpropagation on multiple training data samples, it is desired that each input is inserted into the same model so changes to the weights and biases can be made accordingly. Say, there is a training data set of four data points (a data point contains the input and the corresponding output). When backpropagation is performed on the first data point, the outcome is how to change the parameters of the model to better predict for this data point. So, if optimisation (changing the model parameters according to the outcome of the backpropagation to predict more closely to the desired output) would be performed, this would decrease the cost for this single data point. However, there are three other training data points left. This one change could have increased the cost of those. That would not be desired. When it is the turn of the next training data point, the model has changed already and will provide a different output than it would have before. Also, any changes done now could negate the changes done previously.

To prevent that from happening, instead of performing optimisation directly after calculating the $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of any data point, these values are stored. This provides the opportunity to take the average of all those values and perform optimisation with that. This way, perhaps not for individual points, but on average, the cost will decrease more.

Now the question: Why not put all the training data in one big batch? The answer can be found in time efficiency. Bigger batches take longer to compute. By making it variable, the user can weigh the pros and cons and decide on the best batch size.

The `train` function works precisely as explained above and calculates the average per batch to optimise the model.

When the `backpropagation` method is called, first all changes in cost with relation to the change of all node values are calculated for the entire model in a recursive function. This is stored in a C++ vector. Then, the program loops through each layer starting from the front – near the input (layer 1). There is no reason for this other than convenience because, with the

[8] The fast sigmoid function is an approximation of the sigmoid function, but faster.

change in cost with relation to the change of the node values already being calculated, all changes in cost with relation to the change in weights and biases can be calculated individually. In the loop, per layer, the derivative of the activation is calculated and stored in a vector. With these two vectors, all the changes in cost with relation to the change in biases can be calculated. To calculate the change in cost with relation to the change weights the corresponding node values need to be multiplied as well.

The code is the basis of this paper. All tests have been made using the code and the creation of it has helped with creating an understanding of what neural networks are in practice.

5.3 Backpropagation and Reinforcement Learning

The principle idea of reinforcement learning is simple: The program runs until a point is scored, saving all the inputs and outputs from the model as training data. If the neural network scores a point, save the training data as is, but if the opponent scores, save the training data with the outputs inverted. Then the model trains with backpropagation accordingly.

In the Python code of the game, there is an event loop. This is the loop in which all the new coordinates of all objects are calculated and where the pixels that will be displayed are generated. In the loop, the game also checks for potential user inputs. This is the perfect place to have the model predict and output. The model uses the following as input for the feedforward function:

- the normalized[9] x- and y-coordinates of the ball,
- the normalized coordinates of its paddle
- the y-velocity of the ball.

The output will be a number between 0 and 1 and will be rounded to an integer. This means that the prediction is either 0 or 1 which corresponds with up or down respectively. The paddle then moves accordingly. Furthermore, the input is saved alongside the prediction. When the AI scores a point, all moves leading up to that point are added to a list containing all input data and corresponding output data of all decisions up until the model is trained. If the opponent scores, the data is also added, but then the prediction is inverted (1 becomes 0 and vice versa) so it learns to do the opposite of what it did in the situations leading up to failure. This final list then functions as training data. When the model has been trained, the entire process starts again.

A model is trained when it has played a set amount of games (one cycle). Training means applying backpropagation to the model with the training data gathered in the past cycle.

5.4 Evolutionary Algorithms

For the evolutionary method, backpropagation is not used. Instead, there will be a set of randomly generated models of which some are more suited for its goal. Of these more suited models a new set will be generated, but the newly generated models will have mutations that differentiate them. In the game, the top ten to twenty per cent of the models with the highest fitness will be saved.

[9] The values are normalized by dividing them by the highest possible value they can become. This way, they are a value between 0 and 1. This is easier for the model to interpret.

5.4.1 FITNESS

Deciding what determines fitness is very important. With Pong, there are a few ways. The simplest is awarding one point to each goal scored and having each model run for a set amount of games. The models that scored the most goals, get the highest fitness scores and will survive the round. Another way of calculating the fitness, similar to the first, is by also awarding points when the paddle of the model hits the ball. For example, 5 points when a goal is scored and 1 point per hit. This way of determining the fitness will result in models playing differently than models trained with the first method.

When trying the second method, It is evident that hitting the ball as many times as possible is very rewarding. Even more so than scoring a point. After some dozen cycles, the AI had figured out that there was a flaw in the game engine. The models would exploit the collision (between ball and paddle) by vibrating very quickly and jamming the ball between the floor and its paddle. This way, the ball would get stuck in the paddle, resulting in many hit points.

5.5 TESTING THE HYPOTHESIS

The method used to test the hypothesis of this paper comprises of two parts:

1. Testing reinforcement learning with backpropagation.
2. Testing the evolutionary algorithm.
3. Comparing reinforcement learning with backpropagation to the evolutionary algorithm.

After the game and the neural network had been made, the testing could begin. First, an opponent was made. This was done by making the opponent paddle move up when the ball was above his centre and down when it was below his centre. The models could now be trained on this opponent.

How well a model performs compared to the opponent is calculated by dividing the number of wins by the number of games they have played in total. When 95% of the games are won, the model is regarded to have outperformed the opponent. With reinforcement learning, the number of games played directly influence how well the model learns. With the evolutionary method, however, this is not the case. For this reason, the number of games played per cycle with the evolutionary method can be much lower than the number of games played per cycle with reinforcement learning. This is why the performance is expressed as a ratio instead of the number of points per game.

Firstly, the method with reinforcement learning and backpropagation was investigated. When testing, however, it was evident that the models were having a hard time beating the opponent. This meant that when applying backpropagation, the model rarely scored and therefore could not be positively reinforced. It could only learn from its mistakes (see Figure 6). It was for this reason that a toggleable handicapped opponent was implemented, meaning the opponent moves twice as slow as the model. When training against this opponent, the model was making better improvements.

With finally some results worth examining, something else seemed to be out of order. The model would make a great improvement after one cycle but perform terribly the next (see Figure 7). This phenomenon could be explained by how reinforcement learning is implemented. One cycle, the model would be terrible, learn to do the opposite and perform well the next. However, in that cycle, when it does good moves, sometimes the model doesn’t

score. When it plays very good, the games are short because it scores quick and when it plays mediocrely, the games last longer. The longer games mean more training data is gathered during the cycle. This would be the reason that there is more training data of the model performing mediocrely and losing than doing bad and losing or doing good and winning. This means that it then learns that it should do the opposite of what it was doing when playing mediocrely. However, This way, the model becomes worse.

When playing around with the learning rate, it can be found that the phenomenon of alternating performances can be reduced by increasing the learning rate. Figure 7 was generated with a model that was trained with a learning rate of 0.05. This number was arbitrary. By increasing the number, the model would be changed more drastically and therefore theoretically learn more per cycle. This way, there are fewer mediocre performances. This worked so well, that it seemed to have resolved the issue of having an opponent that is too good. As seen in Figure 8, the model remains consistent much more than before even when training against an opponent that is not handicapped. When training against the handicapped opponent, the model quickly learns to outperform the other player (see Figure 9). The start is still rough, but it quickly catches up. Figure 8 is also a good example of how a neural network has reached its local minimum: The performance has plateaued.

The next training method examined was the evolutionary method. The goal was to compare this method to backpropagation and reinforcement learning. This meant that the first thing that was done, was to train against the handicapped opponent. This worked very well (see Figure 10). It even worked faster than the method using reinforcement learning. The comparison between Figure 9 and Figure 10 shows that neural networks trained with an evolutionary algorithm are more efficient than neural networks trained with backpropagation and reinforcement learning in not only the duration but also in the number of cycles needed to reach the required performance.

After having performed so well training on a handicapped opponent, it was time to remove the handicap. Without the handicap, the training method worked very consistently. It takes a long time, but generally, the top model (best model of the pool) will end up outperforming the opponent (see Figure 11). This is unlike the model trained by reinforcement learning because that model never reaches the level of outperforming an opponent that is not handicapped.

See the following video to get a better understanding of how Pong works with neural networks: https://youtu.be/TBZMXTO0hTU

5.6 Problems and Path to Success

While experimenting, many issues rose to the surface and many possible solutions were tried to fix them. Many of the solutions were never used during final testing but still contributed to the development of a better understanding of neural networks. Mostly at the beginning of development, much of the code was made and the tests were run without a thorough understanding of neural networks and backpropagation.

The first big problem was understanding reinforcement learning. At first, the code was written to follow the following: If the neural network scores a point, save it as good training data, if the opponent scored it, save it as bad training data. It seemed logical to make the learning rate a negative number when training on bad data, then the model would learn to do the opposite. Luckily, there was soon a realisation that backpropagation was all about the influence of

weights and biases on the cost function. So with a negative learning rate, the influence decreases. This is not at all the same as learning to do the opposite.

To fix this, instead of or reversing the learning rate, the training data is reversed. This way, it would learn to do the opposite of what it did when the opponent scored.

After the fix, the model was still not learning very well. To fix this, the input values were changed. Before, they were the coordinates of the ball, the coordinates of the model's paddle and the y-velocity of the ball. These were changed to be normalized values: the y-coordinates of the ball and the y-coordinates of the paddle were divided by the max y-value. This way it was a normalized input: a number between 0 and 1, in this case. This is easier to handle for the model. This worked!

It worked surprisingly well. However, after a few more training cycles, it started doing worse. The issue was that the paddle started doing the opposite of what it was supposed to do: It evaded the ball. The model was very good at it too.

To investigate whether it was the backpropagation that was faulty, models were tested by fitting functions. This resulted in weird issues. The model was resetting after a certain amount of iterations (see Figure 12). To find out what was wrong, the code was investigated. During intensive debugging, nothing seemed out of the ordinary, but the cost would not decrease. While tracing the steps taken to calculate the change in weights and biases to figure out the problem, a huge mistake was found in the code when calculating $\frac{\partial z_n^{(l+1)}}{\partial \lambda_j^{(l)}}$ in the function calculating the change in cost with relation to the change of the node values[10]. The mistake was made when copying some code from other functions.

With the bug fixed, there were great improvements. No longer any inexplicable behaviour and the fitting was much closer.

Inspired by the testing of backpropagation, the function of creating graphs was added to the Pong game. This way, the fitting and interpolating could become visible. For this, the input of the model had to be reduced. Each extra input and output would require a new dimension in the graph. So, the graph would plot two normalized input parameters: the y-coordinate of the ball and the paddle, and one output value. Despite less data, the model performed okay. An example of the graph is in Figure 13.

Also, graphs with more input data were tested. However, none provided any clear insight and were just complicated to look at (see Figure 15).

With the new graph-plotting function it becomes clear how the model is learning. A few things stood out:

1. Per cycle, the training data was very different.
2. The model would overshoot the apparent optimum

[10] Reminder: $\frac{\partial C}{\partial \lambda_j^{(l)}} = \sum_{n=1}^{N_{l+1}} \left(\frac{\partial C}{\partial \lambda_n^{(l+1)}} \times \frac{\partial \lambda_n^{(l+1)}}{\partial z_n^{(l+1)}} \times \frac{\partial z_n^{(l+1)}}{\partial \lambda_j^{(l)}} \right)$

This is the problem later known as the problem of alternating performance levels with reinforcement learning and backpropagation. Before the solution used in final testing was found, it was first solved incorrectly.

During The tests on the backpropagation, it became apparent that the network was excellent at interpolation and thus predicting data that falls within the scope of provided training data. Extrapolation, however, a model cannot do. No function – even linear ones – could be extrapolated with the neural network. This was not an issue, because when learning to play simple games, extrapolation is not required. The idea then was that the training data for learning Pong would have to cover a considerable range of possible situations which the model should learn to deal with.

The information about the limits of my neural network matches the problem of the reinforcement learning algorithm. In one cycle, the model would learn how to deal with one subset of situations and fit the training data. In the next cycle, however, the model would train on completely different training data dealing with a different subset of situations. This way, the model would overwrite what it had learned from previous cycles. As shown in Figure 13, the training data moves from one side to another. This would confuse the model and it would only know how to deal with situations it encountered in its most recent cycle.

These issues are fixed by saving and adding the training data of each cycle to a larger, expanding set of training data on which the model will be trained (see Figure 14). Notice that the previous training data is stored and used every cycle. The positive influence this change has on the model can be spotted by the slight curve upwards on the far left side of the prediction points.

However, when taking a step back and thinking about what this solution is doing, it falls apart. A huge amount of training data is collected and saved so it can relearn data. Using this method with all the training data used, the point of a neural network is defeated. No longer does the model need to find a relation between the input and output in such a way that it can predict well on unseen data. The model has no unseen data. Nearly all possibilities are already described in the training data. This was not at all desired. What is desired is that the model only trains on new data and that it remembers what it has learned before. This was achieved by increasing the learning rate.

Another effort towards improving the training of the model was changing the way the model gets points. Instead of scoring, hitting the ball was awarded points. This was implemented in both the reinforcement learning algorithm and the evolutionary algorithm. However, never much progress was made because of glitches in the game (as explained in 5.4.117) and even when these glitches were fixed, no real improvements were to be seen.

Something else implemented at some point, but never used during final testing was a way to speed up the process of testing called `dataThinner`. This would alternately remove half of the training data a set number of times. The way it worked, seems logical: all of the data points are very close to one another. Of a group, the model only needs one to understand what to learn and less training data saves time. However, it ended up not making that big of a difference

6 CONCLUSION

Feedforward neural networks are systems including many parameters that can learn from training data to find general relations which can be applied to unseen data. This way it can predict, interpolate and fit functions and even learn to play games.

A neural network can be trained to fit training data with backpropagation by changing the parameters to better predict the desired output. When in combination with reinforcement learning, neural networks learn by positive reinforcement when it does well and negative reinforcement when it does bad. Another way to train neural networks is with an evolutionary algorithm. This method tests many models at the same time to see who is more fit for the environment. Those that survive live to reproduce. This way natural selection will ensure models performing well.

The hypothesis, namely that the training method using backpropagation and reinforcement learning is more efficient in training a neural network to play a game than a model trained with the evolutionary algorithm, but the model trained with backpropagation and reinforcement learning will have lower performance than a model trained with the evolutionary algorithm of this paper, has been proven wrong. Neural networks trained with an evolutionary algorithm do end up performing at a higher level than models trained with reinforcement learning and backpropagation. However, neural networks trained with an evolutionary algorithm are also more efficient with regard to not only the number of cycles needed to reach the same performance but also with regard to the time required. Figure 9 and Figure 10 show how reinforcement learning and backpropagation in the case of learning pong is slower and less efficient than the evolutionary algorithm at reaching a high level of performance. Figure 8 and Figure 11 show that models trained by the evolutionary algorithm reach a higher level of performance in general than models trained by the reinforcement learning algorithm and backpropagation.

7 FURTHER RESEARCH

The data seems to be definitive, and the hypothesis seems to be proven wrong. However, because the data deviates so much from its expectation, something might have gone wrong in the research. The reinforcement learning algorithm for example. This could have been wrongly applied to the situation resulting in suboptimal performance, disturbing the data. This is why the reinforcement learning algorithm should be further experimented with and investigated.

8 FIGURES

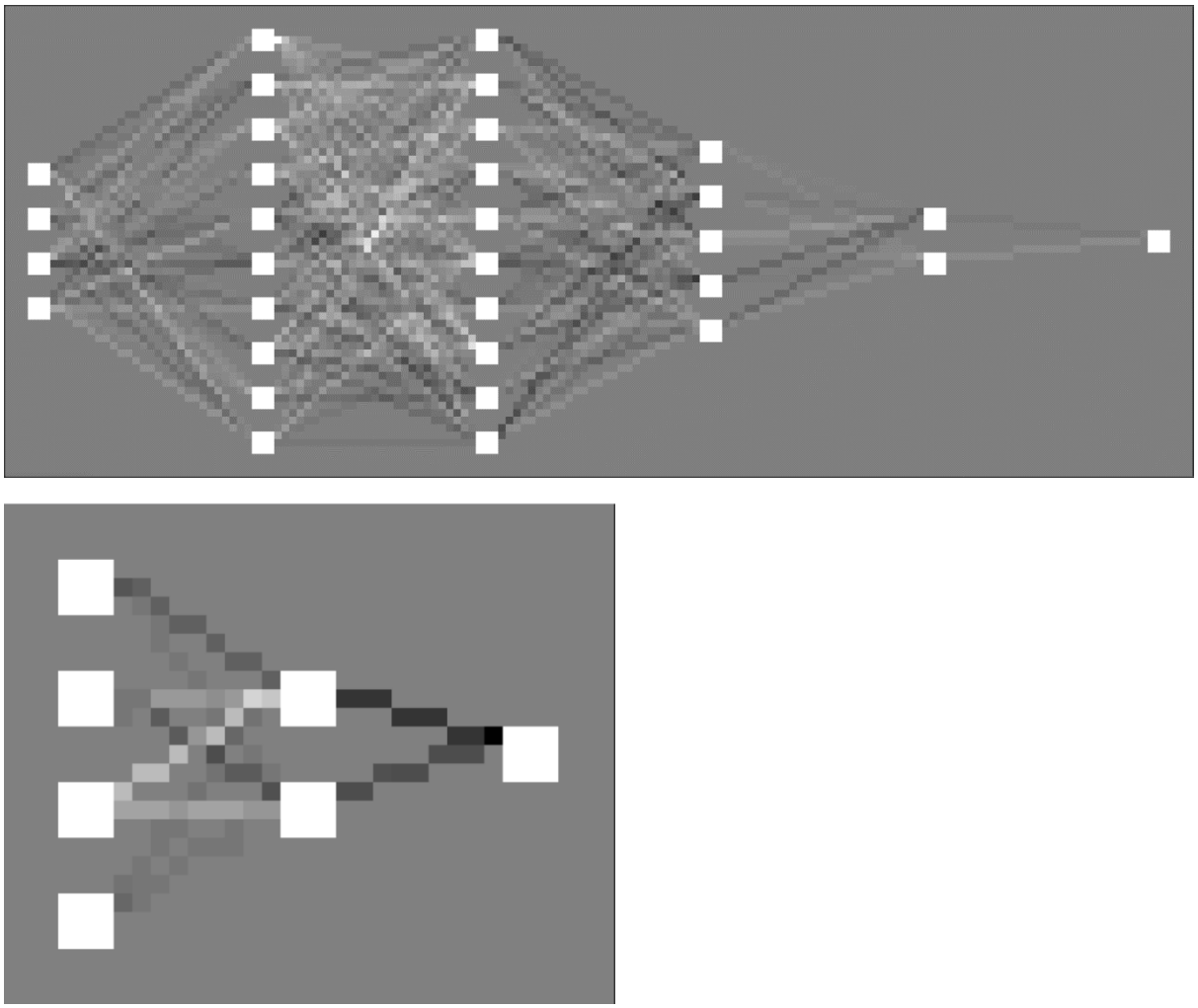

Figure 1 – Two examples of a feedforward neural network. Squares represent nodes and the lines represent the weights. The colour of the lines indicates the value of the weight: the darker, the lower, and the brighter, the higher. Grey like the background is 0*. This means all lines darker than that are negative.*

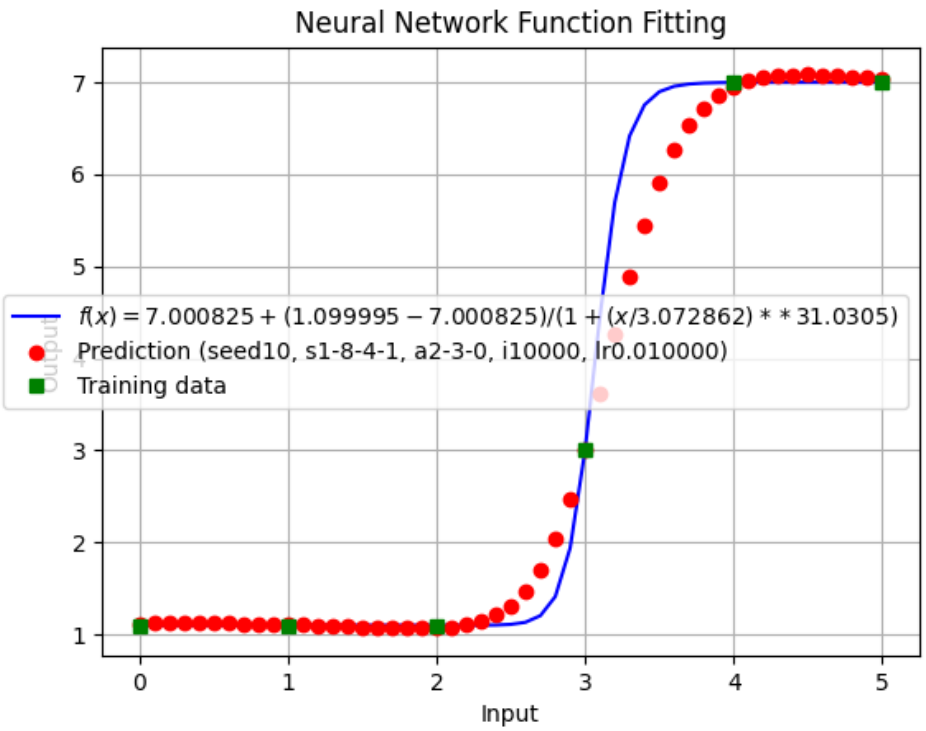

Figure 2 - A neural network interpolating some function f(x) given sparse training data with a learning rate of 0.01 after 10000 iterations of training.

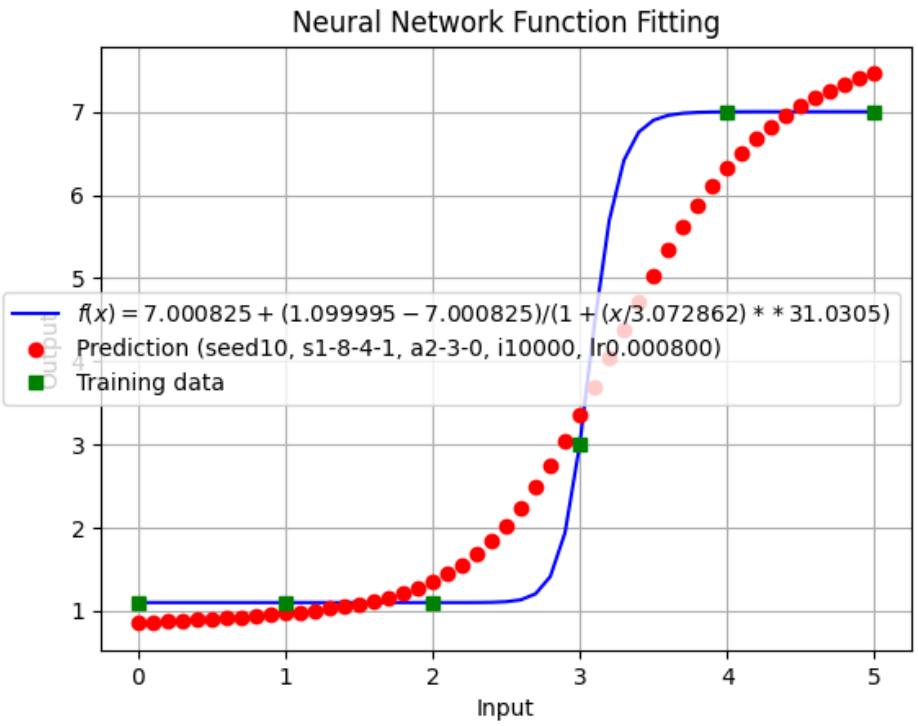

Figure 3 - A neural network interpolating some function f(x) given sparse training data with a learning rate of 0.0008 after 10000 iterations of training.

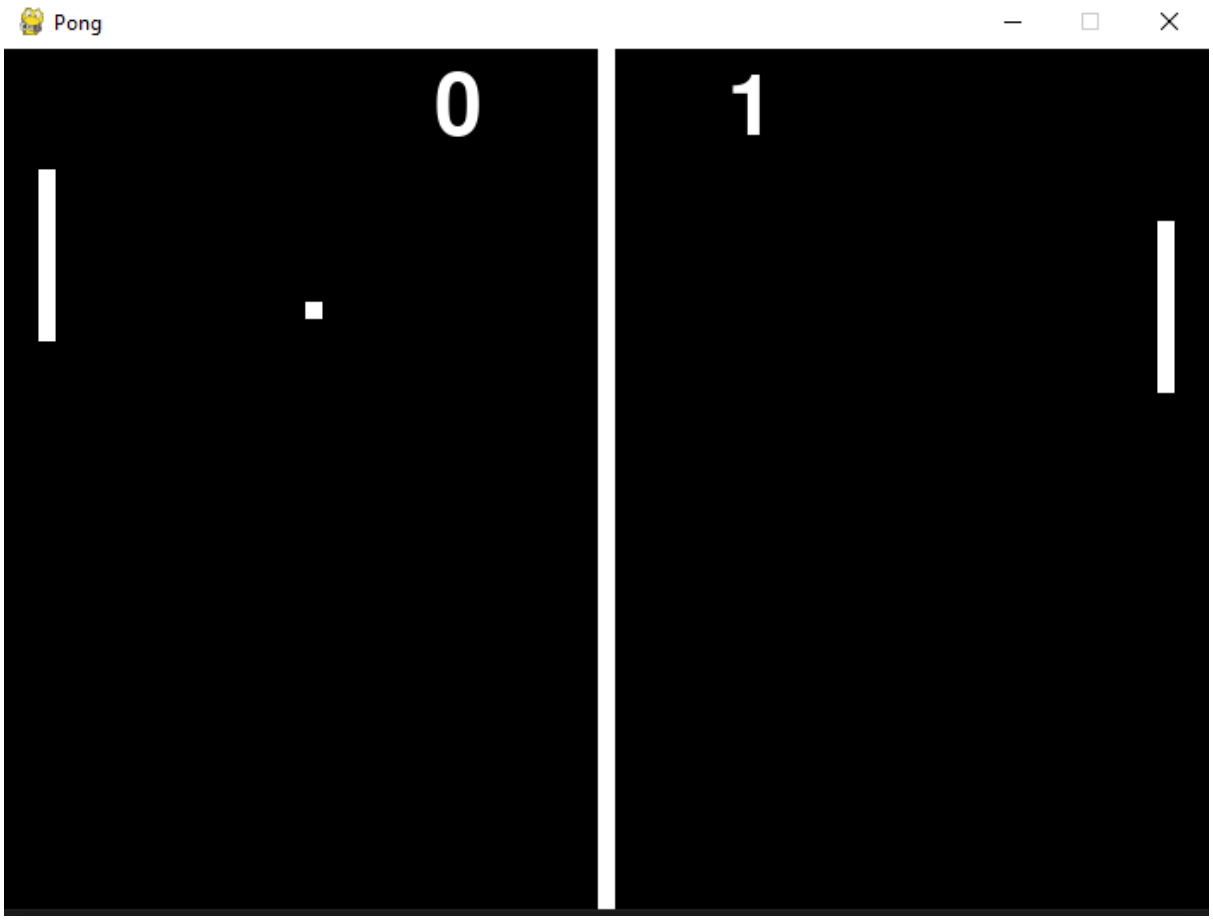

Figure 4 - A freeze-frame of Pong. Left: AI, right: human

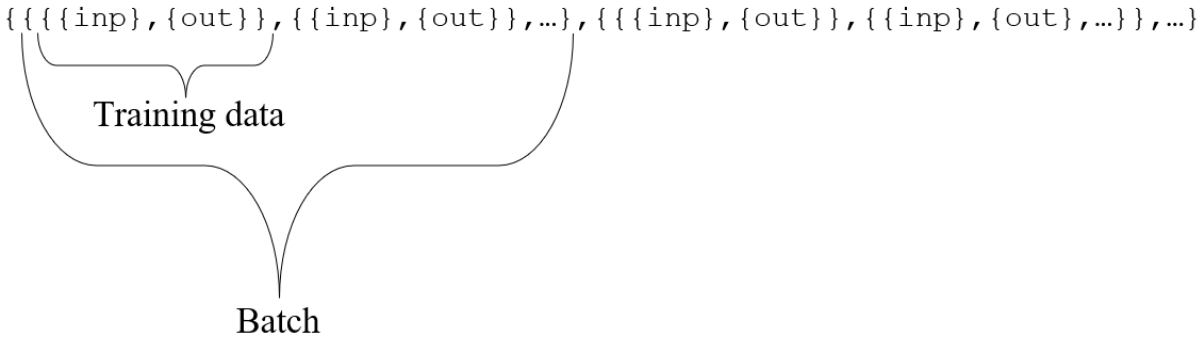

Figure 5 - The training data array format.

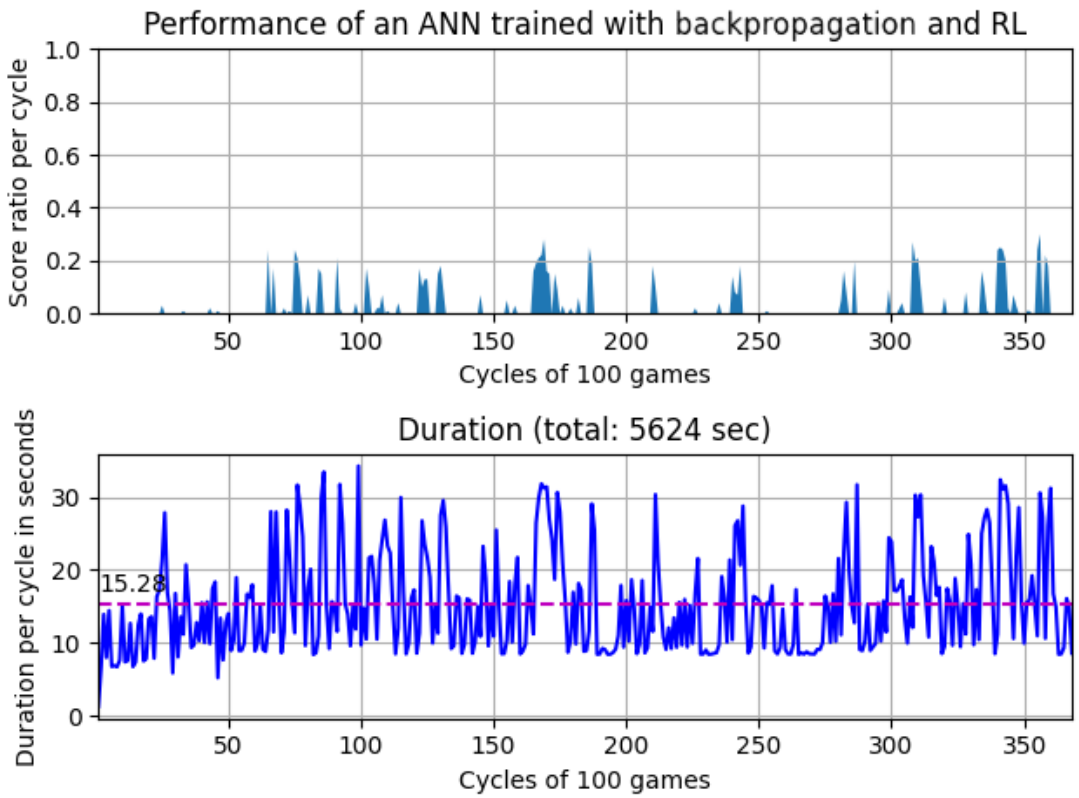

Figure 6 - Reinforcement learning: performance over time. Low performance.

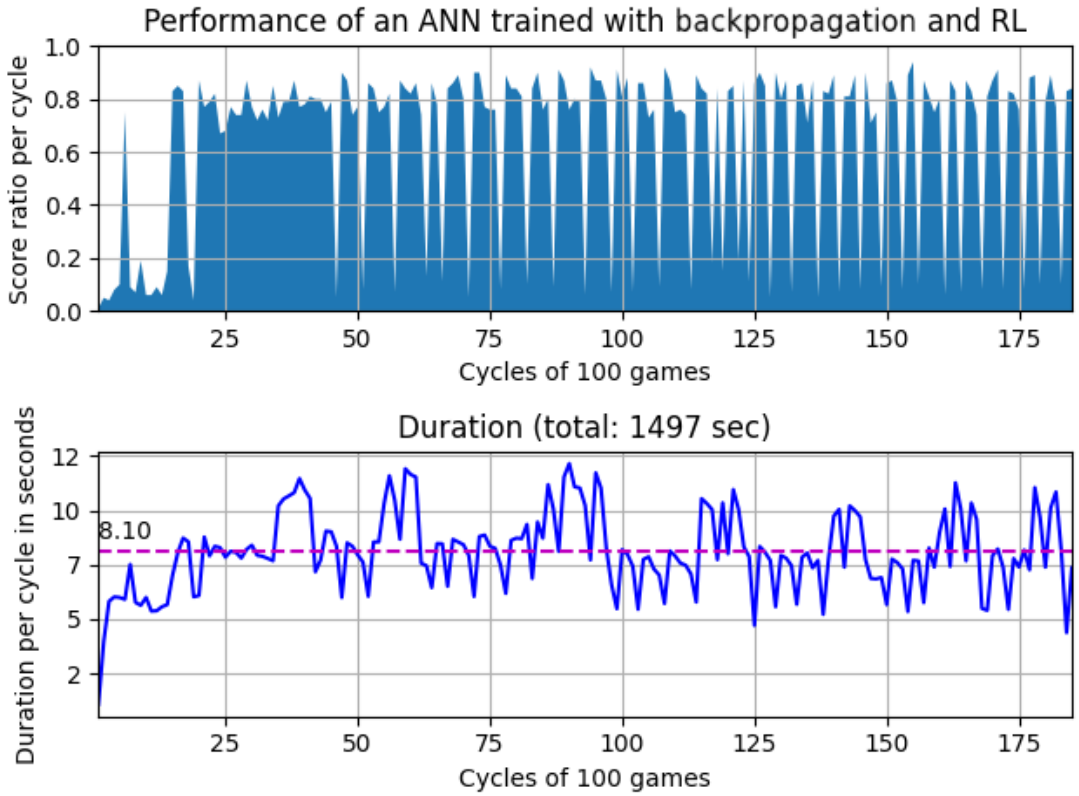

Figure 7 - Reinforcement learning: performance over time. Handicapped opponent. Learning rate: 0.05.

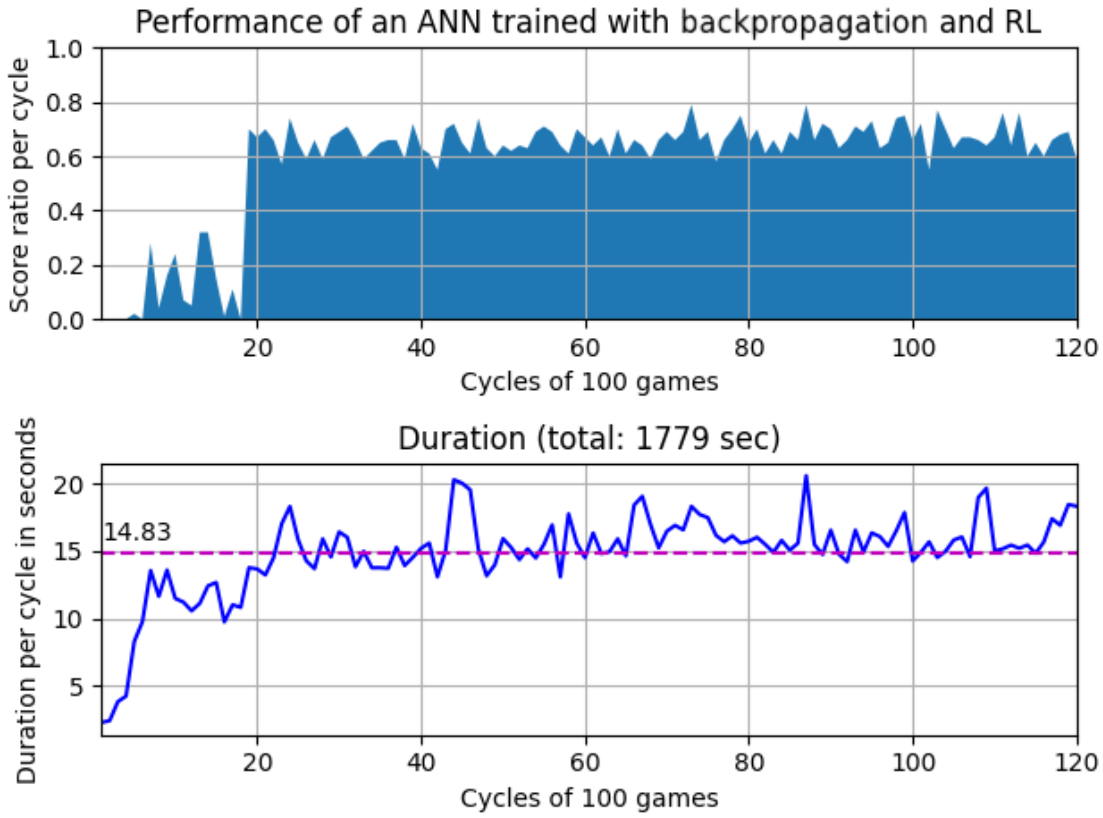

Figure 8 - Reinforcement learning: performance over time. Not a handicapped opponent. Learning rate: 0.2.

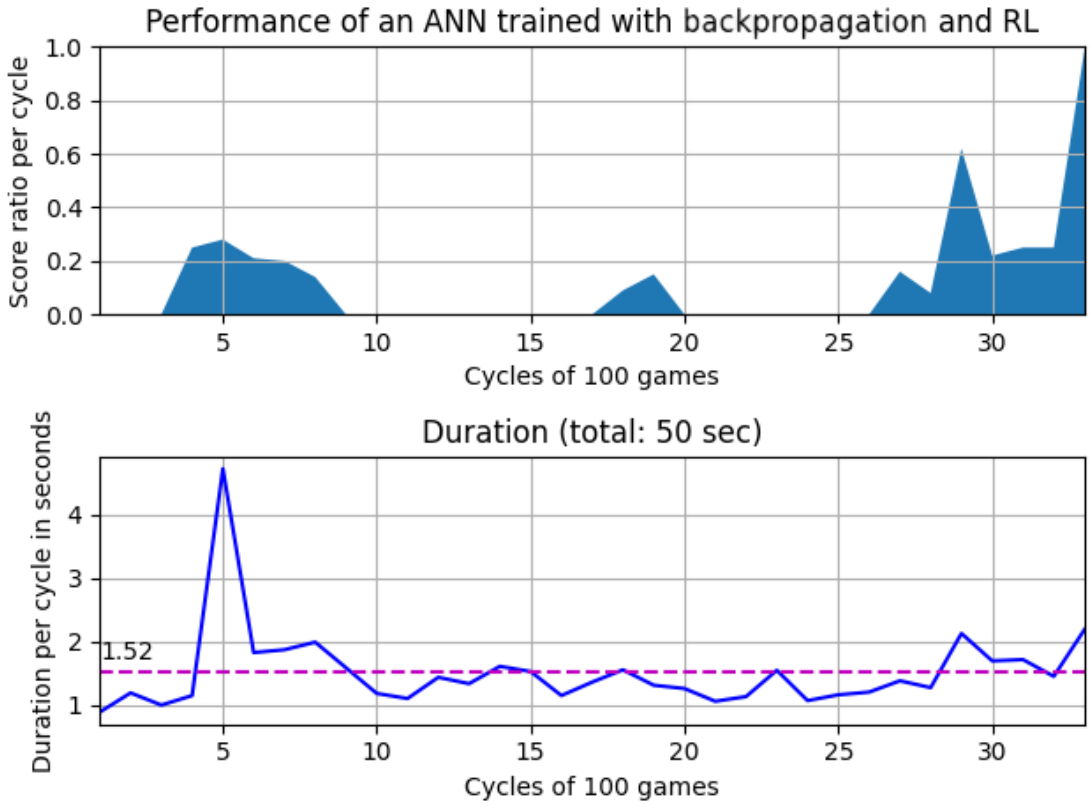

Figure 9 - Reinforcement learning: performance over time. Handicapped opponent. Learning rate: 0.2.

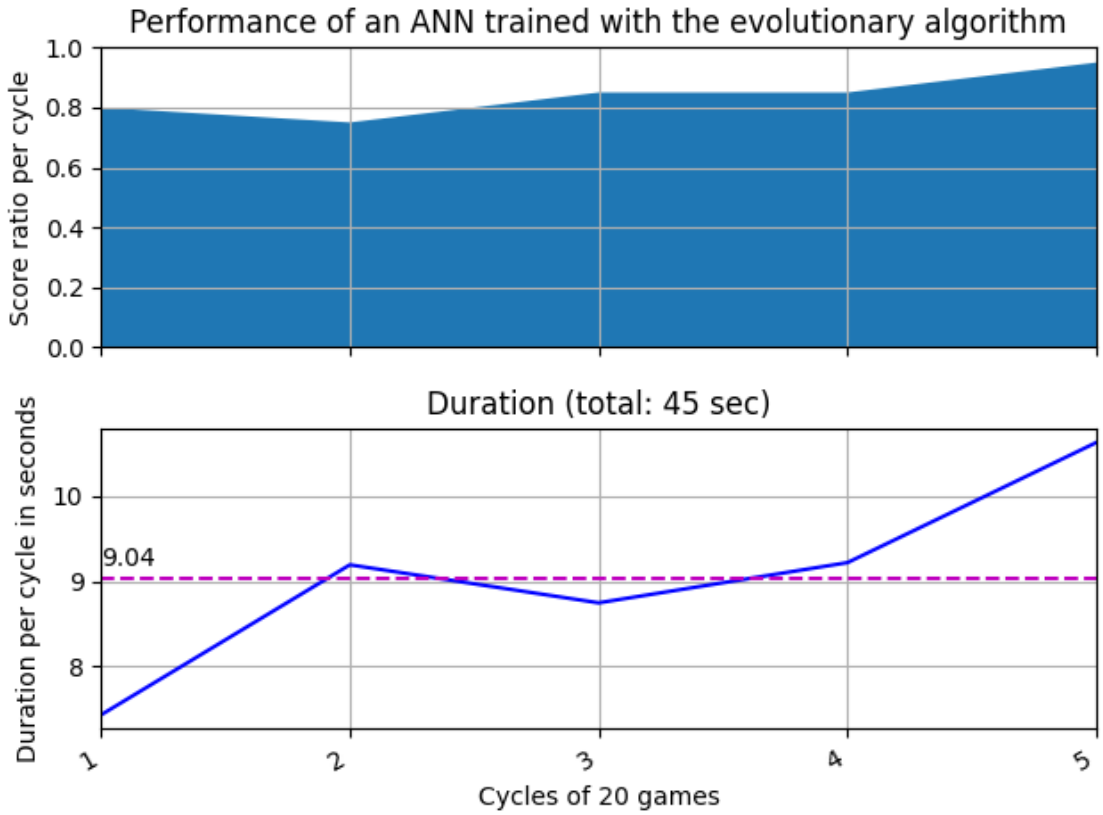

Figure 10 - Evolutionary algorithm: performance over time. Handicapped opponent.

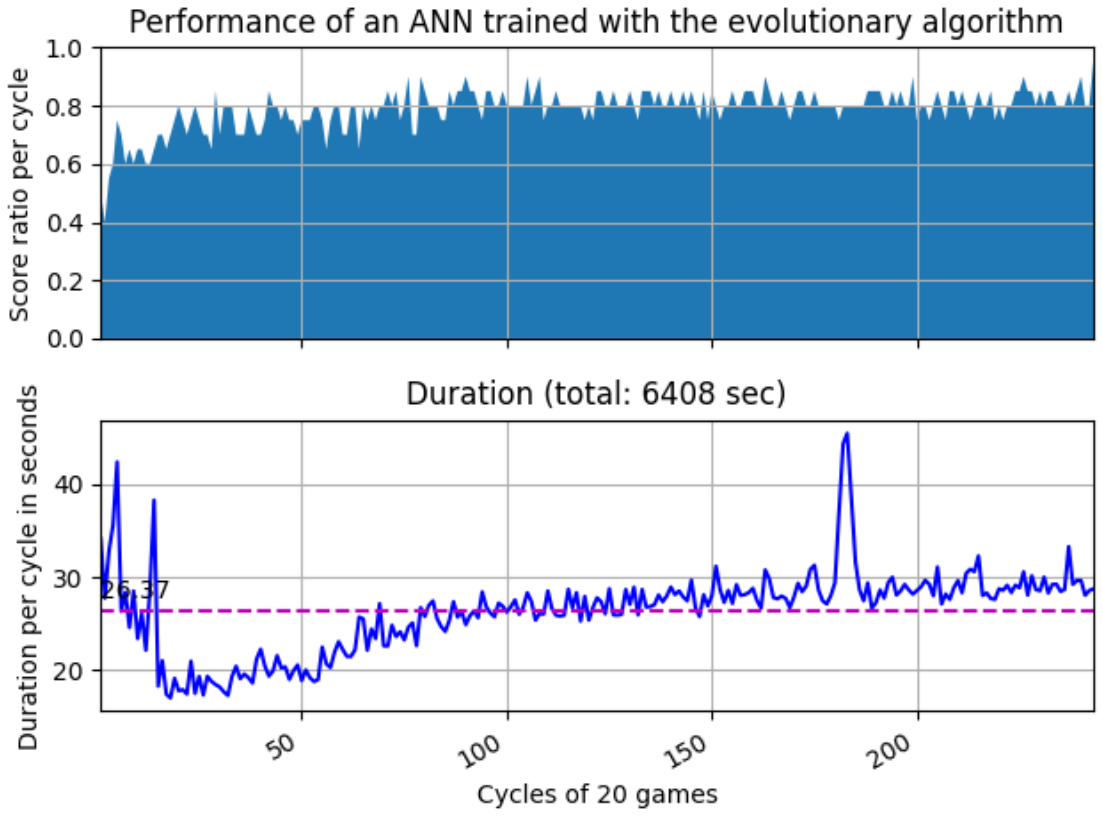

Figure 11 - Evolutionary algorithm: performance over time. Not a handicapped opponent.

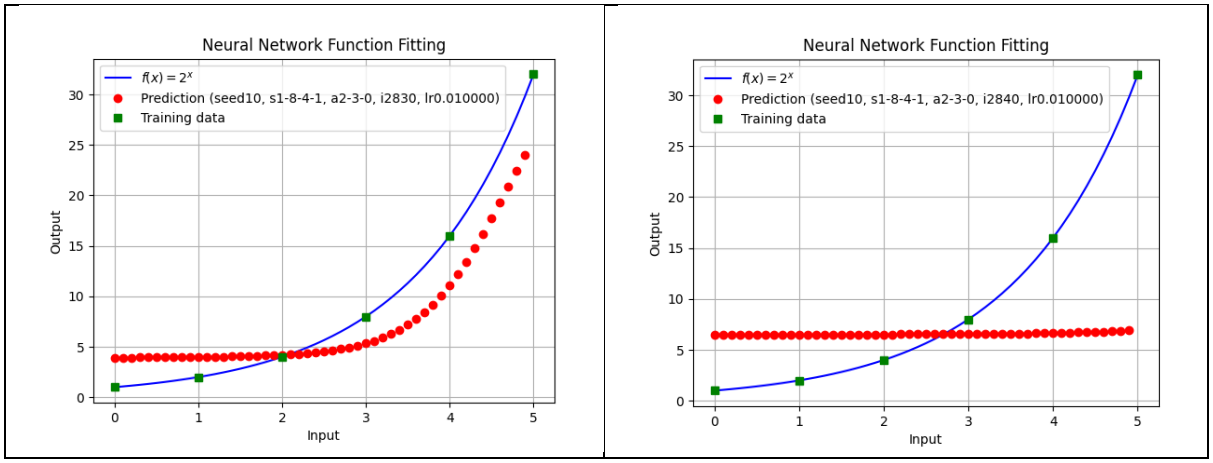

Figure 12 - model resetting between after 2830 iterations.

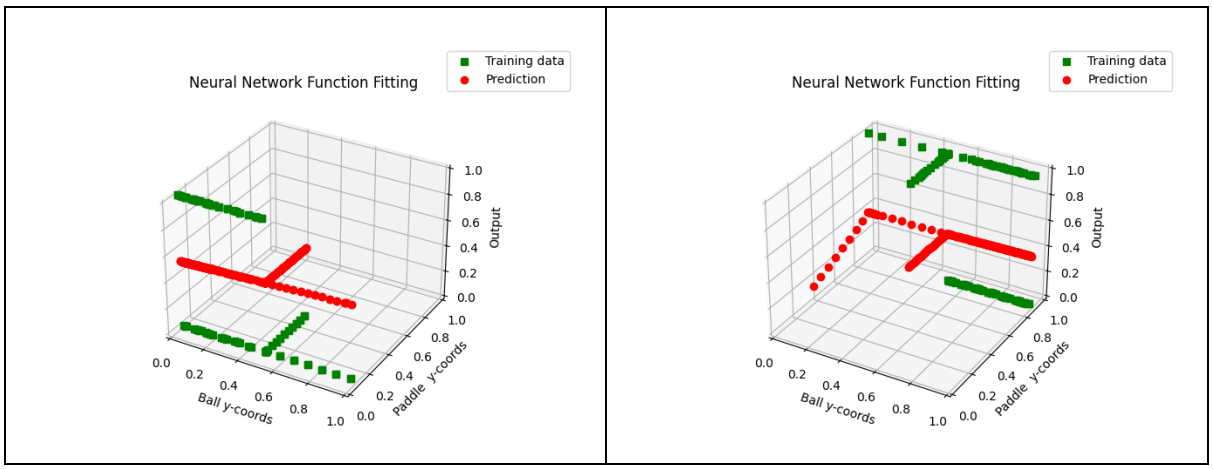

Figure 13 - The first two cycles of a model learning to play Pong without the expanding training data set.

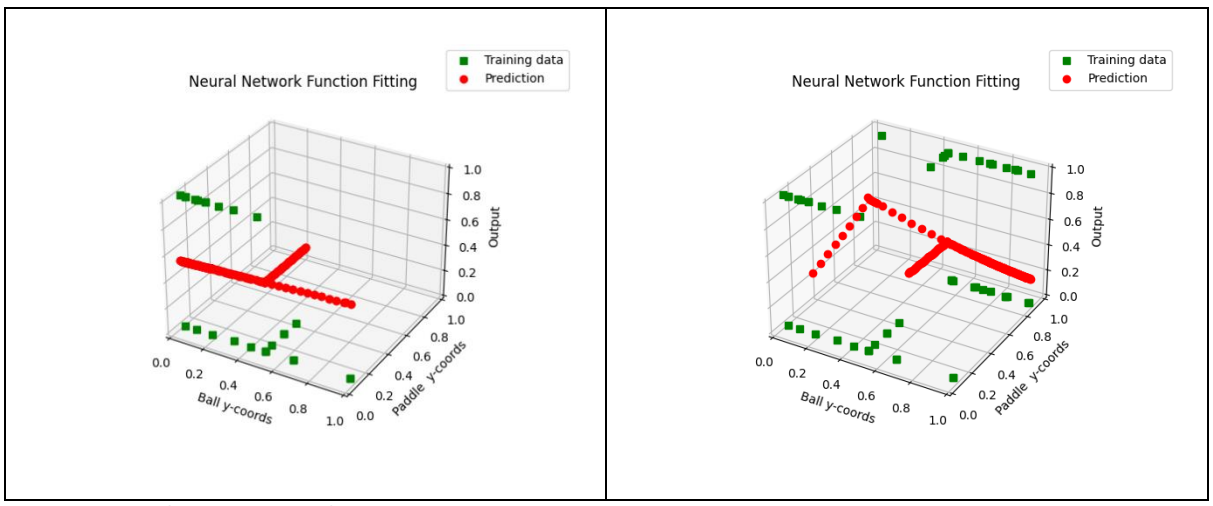

Figure 14 - The first two cycles of a model learning to play Pong with the expanding training data set.

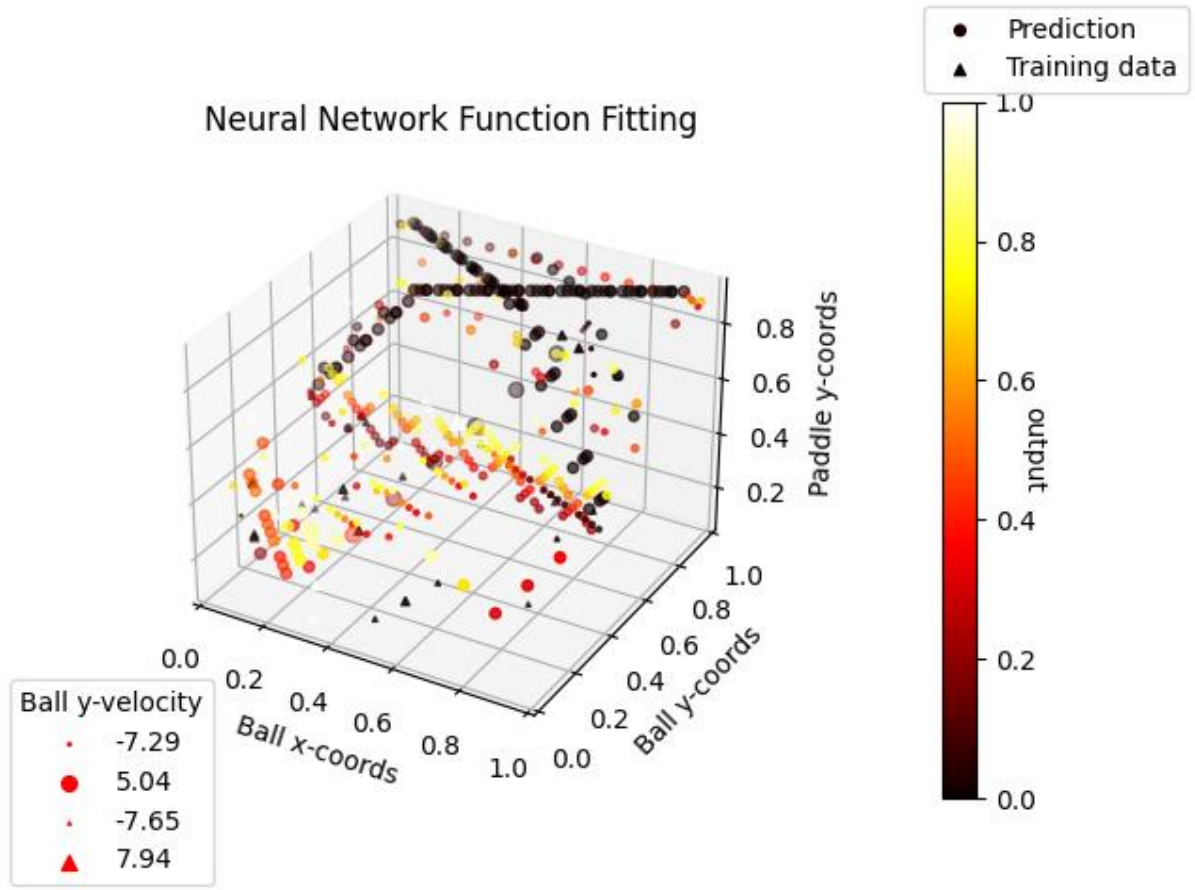

Figure 15 - A 5-D graph of a model learning Pong with four input values and one output value. Complicated and not easy to understand.

9 SOURCES

101 Computing. (2019, May 27). *Pong Tutorial using Pygame*. Retrieved from 101 Computing: https://www.101computing.net/pong-tutorial-using-pygame-getting-started/

Bendersky, E. (2016, October 18). *The Softmax function and its derivative*. Retrieved from The Green Place: https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/

cplusplus.com. (n.d.). *std::vector*. Retrieved from cplusplus.com: https://www.cplusplus.com/reference/vector/vector/

cppreference.com. (n.d.). *Pseudo-random number generation*. Retrieved from cppreference.com: https://en.cppreference.com/w/cpp/numeric/random

DeepMind. (n.d.). *About*. Retrieved from DeepMind: https://deepmind.com/about

IBM Cloud Education. (2021, March 3). *Overfitting*. Retrieved from IBM: https://www.ibm.com/cloud/learn/overfitting

Jain, V. (2019, December 30). *Everything you need to know about "Activation Functions" in Deep learning models*. Retrieved from towards data science: https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253

Karpathy, A. (2016, May 31). *Andrej Karpathy blog*. Retrieved from Deep Reinforcement Learning: Pong from Pixels: http://karpathy.github.io/2016/05/31/rl/

Maasdam, K. v. (2021, October 13). *PWSFeedforwardNeuralNetworkPong*. Retrieved from GitHub: https://github.com/KaspervanM/PWSFeedforwardNeuralNetworkPong

Maasdam, K. v. (2021, October 13). *Training Feedforward Neural Networks*. Retrieved from YouTube: https://youtu.be/TBZMXTO0hTU

Pai, A. (2020, October 26). *Mean Squared Error Cost Function*. Retrieved from Machine Learning Works: https://www.machinelearningworks.com/tutorials/mean-squared-error-cost-function

Shiffman, D. (2018, March 12). *11.1: Introduction to Neuroevolution - The Nature of Code*. Retrieved from YouTube: https://www.youtube.com/watch?v=lu5ul7z4icQ

Stroustrup, B. (1997). *The C++ Programming Language Third Edition.* Reading, Massachusetts, the United States of America: Addison-Wesley.

Tutorialspoint. (n.d.). *C++ Class Constructor and Destructor*. Retrieved from Tutorialspoint: https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm