Profielwerkstuk

Machine Learning, Neural Networks and the evolution of Neural Networks through Augmenting Topologies

Describing the underlying theory, methods and the particular application of these principles to making a computer teach itself how to play a video game

by Maxim L. van den Berg, Sam R.W. van Kampen,
Wilco M. Stam and Robin A.J. Wacanno
NG & NT Informatica

Tabor College Werenfridus WV6X & WV6Y
De Keyzerstraat 1
1624 BX Hoorn
The Netherlands

Under supervision of Mr. P.H. Harmsen

3^{rd} of February 2017

Abstract

An important question in machine learning is how to decide on a method to solve a complex problem. One worth examining is gameplay, due to its dynamic and challenging nature. We examine the method of evolving artificial neural networks and self-learning AI after discussing its precursors and underlying machine learning algorithms, and show its proficiency in solving the platforming problems in level 1-1 of Super Mario Bros. The algorithm was able to achieve significant progress in completing the level and was able to evolve the necessary structure to overcome obstacles. Due to time constraints, the program was unable to complete the level, but it is, given enough time, destined to complete it.

Contents

1 Introduction

Recently, the field of computer science has shown impressive breakthroughs in several machine learning methods. Most notably there have been recent developments in so called *Deep Learning* methods, among which the algorithm AlphaGo, a computer program that managed to beat a human professional Go player in October 2015.[1] This was achieved by use of a particularly sophisticated artificial neural network (ANN). Other, less recent developments concerning ANN's include the NEAT algorithm, an especially interesting approach to learning and a substantial part of this paper.[2] Ultimately, our goal is to use these already established concepts to make a computer teach itself to play a video game, more specifically the first level of the 1988 Nintendo game Super Mario Bros. 3.

For now, many of the used terms might seem somewhat perplexing. However, all concepts are explained in an as understandable manner as possible throughout this paper. The knowledge required to fully understand the process of making a computer learn to complete the first level of Super Mario Bros. 3, the basics and history of machine learning, artificial neural networks and the NEAT algorithm, will all be presented. For this paper, no previous knowledge of machine learning is required. However, a basic understanding of programming syntax, namely the language Octave (Matlab), Lua and Python, and the mathematical concepts of calculus and linear algebra, particularly concerning derivatives and matrix multiplication, is assumed.

As stated, this paper will concern the subfield of *computer science machine learning* (ML). Machine learning seeks to give computers the capability to

> *"...adapt to new circumstances and to detect and extrapolate patterns."*[3]

In this paper, we seek to answer the following question:

> *How can a machine learning algorithm be made to learn how to finish level 1-1 of Super Mario Bros. 3 for the Nintendo Entertainment System?*

We hypothesize the following:

> *Our algorithm will succeed in finishing level 1-1 of Super Mario Bros. 3 if given enough time.*

In order to effectively answer this question, and ensure a satisfactory understanding, this question is divided into a set of more manageable subquestions. Before more complex concepts can be described, it is important to first understand the basics of machine learning. To understand this, the following question needs to be answered:

> *How does a machine learning algorithm learn?*

To answer this question, this paper will provide a short summary of the history of and the theory behind machine learning in chapter 2, which will serve to contextualize the field of machine learning. Chapter 3 will concern the most basic form of machine learning, linear regression, and provides a clear mathematical introduction and explanation of the basic concepts. While not directly supporting the previously stated main goal of the paper, these chapters will provide essential knowledge necessary for the understanding of the more complicated concepts.

[1] Silver and D. Hassibis (2016).
[2] Stanley and Miikkulainen (2002).
[3] Russell and Norvig (2009).

Secondly, an understanding of artificial neural networks, a integral part of the eventually used learning method, must be gained. This is achieved in chapter 4 by answering the second subquestion:

How does an artificial neural network learn?

Furthermore, this will serve as a short introduction to the important and interesting concepts of neural networks. Moreover, the information provides in this chapter is the basis of the concepts of *Neuro-evolution*, the main notion behind the NEAT algorithm and the goal of this paper. A key obstacle in applying the artificial neural network learning process to video games however, is the large focus on the *topology*, the layout, of the network. Therefore, the following subquestion is posed:

How does decide upon the best possible topology of the artificial neural network for the algorithm to be able to solve our posed problem?

In chapter 5, the NEAT algorithm, which solves this problem, is discussed to answer this question.

Finally, the knowledge built up over the course of the paper can be applied, answering the last subquestion:

How do you implement Evolving Neural Networks through Augmenting Topologies to finish our posed problem?

Firstly, in chapter 5 section 2, the theory behind the application of these concepts to video games is described. Secondly, in chapter 5 section 3 and chapter 6 section 1 our NEAT implementation, written in Python, and emulator extension, written in Lua, are discussed. Additionally, in chapter 6, we take a final look at the outcome of the application, and the achievements and shortcomings of the developed programs. By answering the stated questions and subquestions, we were able to gain the understanding to successfully apply these concepts and accomplish our objective.

2 Background information

2.1 The History of the Automation of Algorithmic Problem Solving

Before giving some background information on the subjects which will be described in this report, it is useful to gain some knowledge about the historical context in this field of research. Therefore, some of the most significant advancements will be described in the following paragraphs. Additionally, their importance to the subject of this report will be discussed.

2.1.1 Bayes' Theorem

The history of machine learning begins not just decades but centuries ago. It begins with the posthumous publication of *An Essay towards solving a Problem in the Doctrine of Chances*, a work by the Reverend Thomas Bayes regarding the mathematical theory of probability. In the *Essay*, Bayes lays out a number of statistical propositions, the fifth of which delineates his seminal theorem. The *Essay* describes the following:

> *If there be two subsequent events, the probability of the* 2^{nd} $\frac{b}{N}$ *and the probability of both together* $\frac{P}{N}$*, and it being* 1^{st} *discovered that the* 2^{nd} *event has also happened, the probability I am right is* $\frac{P}{b}$*.* [4]

This symbolically implies the following [5]:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}, \text{ if } P(A) \neq 0,$$

which leads to Bayes' Theorem for conditional probabilities:

$$\Rightarrow P(A|B) = \frac{P(B|A)P(A)}{P(B)}, \text{ if } P(B) \neq 0.$$

The theorem describes the relation between a prior probability $P(A)$ and a posterior probability $P(A|B)$ given $P(B|A)$, the probability of observing B when given that A is true:

$$P(A|B) \propto P(B|A)P(A).$$

A is called a conditional probability since the occurrence of event B influences the probability of A's occurrence. An intuitive example is the classification of fruit. When given a random fruit and asked to determine whether it is an apple or an orange, the probabilities for either class is 50%. When given the additional evidence that the fruit is red, probabilities obviously shift in favour of the apple. The conditional probability of the fruit being an apple increases, given the evidence that the fruit is red.

This theorem is used in machine learning to create Naive Bayes classifiers. These are models that assign class labels to problem instances (sets of features $x_1, x_2, \ldots, X_n$), class labels drawn from a finite set, based on conditional probabilities.

Abstractly, a classifier based on conditional probability assigns instance probabilities

$$P(C_k|x_1, \ldots, x_n)$$

[4] Bayes (1763).
[5] Stigler (1982).

for each class C_k. It is, however, infeasible to base such a model on probability tables when the number of features n is large, or if a feature can take on a large number of values. We can use Bayes' Theorem to decompose the probability as follows:

$$P(C_k|x) = \frac{P(C_k)P(x|C_k)}{P(x)}$$

The numerator is equivalent to the joint probability $P(C_k, x_1, \ldots, x_n)$, which can be rewritten as follows:

$$\begin{aligned}
P(C_k, x_1, \ldots, x_n) &= P(x_1, \ldots, x_n, C_k) && (1)\\
&= P(x_1|x_2, \ldots, x_n, C_k)P(x_2, \ldots, x_n, C_k) && (2)\\
&= P(x_1|x_2, \ldots, x_n, C_k)P(x_2|x_3, \ldots, x_n, C_k)P(x_3, \ldots, x_n, C_k) && (3)\\
&= \ldots && (4)\\
&= P(x_1|x_2, \ldots, x_n, C_k)P(x_2|x_3, \ldots, x_n, C_k)\ldots P(x_{n-1}|x_n, C_k)\\
&\quad P(x_n, C_k)P(C_k) && (5)
\end{aligned}$$

Now, the "naive" part of Naive Bayes comes into play. The algorithm assumes each feature F_i is conditionally independent of every other feature F_j for $j \neq i$, given C. This means that

$$P(x_i|x_{i+1}, \ldots, x_n, C_k) = P(x_i|C_k)$$

And the joint probabilities above can be written as

$$\begin{aligned}
P(C_k|x_1, \ldots, x_n) &\propto P(C_k, x_1, \ldots, x_n) && (6)\\
&\propto P(C_k)P(x_1|C_k)P(x_2|C_k)\ldots && (7)\\
&\propto P(C_k)\prod_{i=1}^{n} P(x_i|C_k) && (8)
\end{aligned}$$

Under the above independence assumptions, the initial instance probability equation can be simplified to:

$$P(C_k|x_1, \ldots, x_n) = \frac{1}{P(x)}P(C_k)\prod_{i=1}^{n} P(x_i|C_k)$$

This probability model is then combined with a decision rule, a common one being choosing the hypothesis that is most probable, the *maximum a posteriori* decision rule. The corresponding classifier is the function that assigns a class $\hat{y} = C_k$ as follows:

$$\hat{y} = \underset{k\in\{1,\ldots,K\}}{\operatorname{argmax}}\ p(C_k)\prod_{i=1}^{n} p(x_i|C_k).$$

2.1.2 Turing Test

In the year 1950, computer scientist Alan Turing published a seminal paper in the philosophical magazine *Mind*.[6] This publication proved an important step in the development of Machine Learning algorithms and Artificial Intelligence. In this paper Turing ponders

[6]Turing (1950).

the question of whether machines are able to think. He does this by trying to rephrase the question, for a question such as *"Can a machine think?"* is too vague, containing concepts such as *'think'* and *'machine'*, which are not clearly and universally definable.

Turing proposes that, instead of asking this question, one should seek to find out whether a machine is able to play the so called *Imitation Game*. In Turing's version of this game, more commonly know as the *Turing Test*, there are three participants: a machine, a human and a judge (also a human). Let us call these A, B and C respectively. Each of the participants are held in isolated rooms and participant C is only able to communicate with participants A and B with a terminal, through which he is able to ask both of them questions. The aim of the game is for participant C to correctly identify whether participants A and B are human or machine. Participants A and B however, both try to convince participant C they are in fact human. Thus *'thinking'* has been redefined as being able to act indistinguishably from an entity that, without a doubt, is able to think: a human.

Now, Turing has only solved half of the problem; he still has to come up with a way to unambiguously define the word *'machine'*. This is vital, for there are a number of objects which could classify as a machine but would not be suitable to take this test. Turing has an answer to this however, for he thinks this theory should only apply to binary machines (which we now know as computers). According to him there are two important reasons for using these kinds of machines. For one, they are already available; their existence is not debatable. The second reason is based on one of his theories, which states that any binary or digital machine can simulate any other digital machine. This means that, if *any* digital machine is able to pass the *Turing Test*, *all* of these machine should—in theory—be able to pass the test.
Thus Turing is able to ask the following, more specific question:

> *Is there a conceivable computer D, one with the adequate amount of computing power and the right program, that is able to play the Imitation Game as participant A against participant B—who is human—in such a way that judge C is unable to correctly determine whether participant A or B is the real human?*

In this way, the Turing Test has been used for decades to test the *'intelligence'* of certain Machine Learning algorithms and Artificial Intelligences.

2.1.3 SNARC

One year after Turing's publication, Marvin Minsky finished building his Stochastic Neural Analogue Reinforcement Calculator, also known as SNARC. SNARC was a network of 40 randomly connected Hebbian synapses, synapses whose connections strengthen when the presynaptic and postsynaptic elements are active. The SNARC was based on a model created by Warren McCulloch and Walter Pitts in 1943 wherein they proposed a model of artificial neurons where each neuron is characterized as being *on* or *off*, with a switch to *on* occurring in response to stimulation by a sufficient number of neighbouring neurons. With the addition of a rule that modifies the connection strength between neurons, the network gains the ability to improve its performance.[7] This rule is now called *Hebbian Learning*.

The SNARC consisted of 40 neurons connected randomly and each neuron consisted of a long term memory and a short term memory. The long term memory is the probability that if a signal comes into on of the inputs that then a signal will come out of

[7] Russell and Norvig (2009).

the output. This probability varies from 0—if the potentiometer is turned down—to 1—if the potentiometer is turned all the way up. If a signal gets through then the short term memory activates, which consists of a capacitor, and if the operator of the SNARC liked the outcome it could reward the neurons that fired by pressing a button that changed the probability of a neuron firing based on the state of the short term memory. The network achieves this by running a chain powered by a motor along all the potentiometers which turns the shaft of the potentiometers when the electric clutch is engaged. This only happens when the neuron has a charged capacitor, and thus has received a signal.[8] In essence the SNARC was a really rudimentary artificial neural network, defined in chapter 4.

2.1.4 Perceptron

Hebb's learning methods were improved upon by Frank Rosenblatt with his perceptron. The perceptron was a machine designed for image recognition with 400 photocells randomly connected to artificial neurons. The weights of the connections were stored in potentiometers and these could be adjusted by electric motors to "learn". The perceptron convergence theorem says that the learning algorithm can adjust the connection strengths of a perceptron to match any input data, provided such a match exists.[9, 10]
The early AI developments initially seemed promising, however Marvin Minsky and Seymour Papert book proved in their book *Perceptrons* (1969) that, although perceptrons (a simple form of neural network) could be shown to learn anything they were capable of representing, they could represent very little. In particular, a two-input perceptron (restricted to be simpler than the form Rosenblatt originally studied) could not be trained to recognize when its two inputs were different (the `XOR` gate).[11] Although their results did not apply to more complex, multilayer networks, research funding for neural networks research soon diminished as the optimism for the capabilities of neural networks vanished.

2.1.5 Backpropagation

In the 1980s the rediscovery of an algorithm to adjust the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector caused a resurgence in machine learning research. It was applied to many learning problems in computer science and psychology, causing great excitement. The most important feature of backpropagation is that it organizes neurons in the hidden layers to learn to recognize different characteristics of the input. This means that if the network is presented with incomplete or noisy data it will still be able to recognise a pattern if it is present.[12]

2.2 An Introduction to Machine Learning

2.2.1 The Basics of Machine Learning

Fundamentally, machine learning is a simple concept, for which the previously described Bayesian theorem is of great importance. Machine learning algorithms are concerned with finding relations in data sets and between different kinds of data. With this they can for instance construct a function, which describes the given data as accurately as possible and from which predictions can be made concerning the data. Machine learning is a broad term, and may refer to many different algorithms, such as linear regression, as described in

[8] Sykes (2011).
[9] Block (1962).
[10] Russell and Norvig (2009).
[11] Minsky and Papert (1969).
[12] Russell and Norvig (2009).

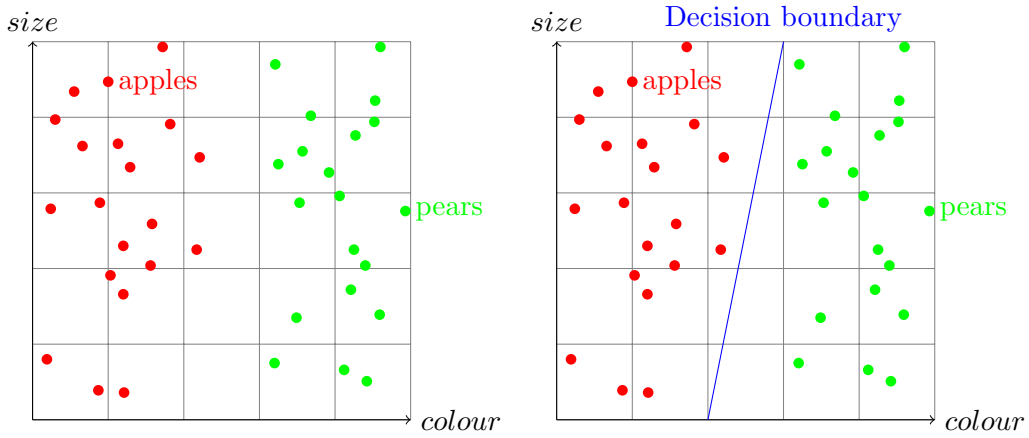

Figure 1: An example of a labelled dataset of apples and pears, and a potential decision boundary.

chapter 3, polynomial regression, and artificial neural networks. The process any machine learning algorithm is required to solve is referred to as a *learning problem*. Learning problems can vary substantially in terms of size, goal and complexity. The following examples will concern extremely basic learning problems, while our final goal, playing a video game, can be considered a complex learning problem. Machine learning is one of the largest subfields of computer science, and acquiring a great understanding of the many more complex concepts requires years of study.

2.2.2 Classification and Regression

A distinction is often made between two types of data of a learning problem. If the outcome is a discrete set of values, such as 'apple', 'pear' or 'orange' for instance, it is referred to as a *classification problem*. If the outcome is made up of continuous values, such as the price of a house for instance, it is referred to as a regression problem. A simple example of a data set of a classification problem can be seen in figure 1, and depicts a number of apples and pears with their corresponding sizes and colours. In figure 2, a simple example of a regression problem can be seen, which depicts the correlations between the size of a house and its price. Note that in reality, data sets are not as contrasting as is portrayed here, and relationships can be a lot more complex.

In this example, the red dots represent apples and the green dots represent pears. We can see that apples are mostly clustered on the left side and pears are mostly clustered on the right side. As shown in figure 1, the learning algorithm might find a *linear decision boundary* to separate the two clusters. After it has been sufficiently trained, the algorithm can determine, by using the size and colour, if a new unspecified piece of fruit is either an apple or a pear.

In figure 2, each orange dot represents a single house, from which the relationship between them can be observed. The (linear regression) algorithm fits a line through the data, staying close as possible to each data point. This line represents the average relationship in the data, and can therefore be applied to predict future relations. In this example, it can predict the price of a house based on its size. The hypothesis has calculated a linear correlation. However, it is possible for any correlation, such as an exponential function or following a sigmoid function, to be calculated, provided some more complex algorithms are used.

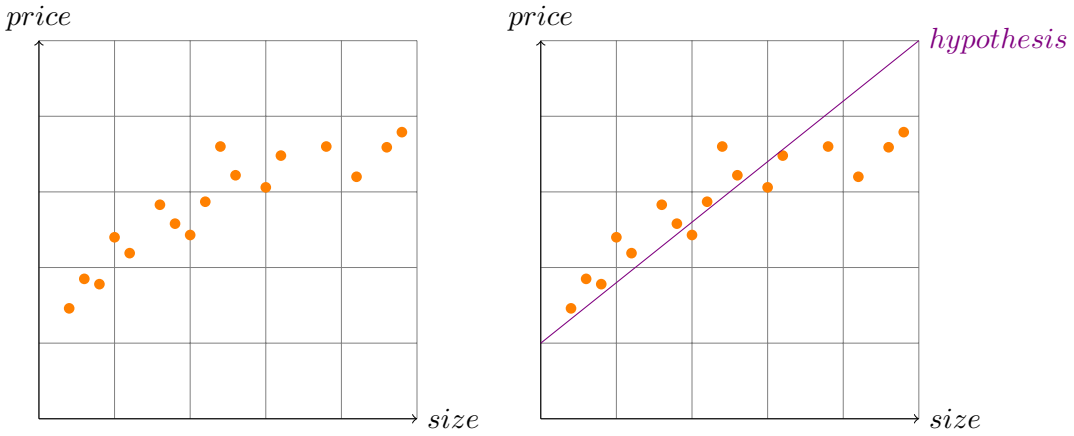

Figure 2: An example of a labelled dataset of houses and a potential hypothesis function.

2.2.3 Supervised, Unsupervised and Reinforcement Learning

> *Well-posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."* [13]

For an algorithm to be capable of improving its performance from experience with respect to some task it needs to have feedback. This feedback comes in three forms that determine the type of learning: supervised, unsupervised and reinforcement learning.

In *supervised learning*, the algorithm is given the correct outputs for any input set, and thus can learn how to fit a function using both inputs and output. Figure 1 and 2 are both examples of supervised learning. The classification algorithm knew what sizes and colours corresponded to an apple or a pear and the regression algorithm knew the price that corresponded to the size of each house. When, for certain data, the correct answer is delivered with it, the data is called *labelled data*, when it is not, it is referred to as *unlabelled data*.

In *unsupervised learning*, the algorithm learns the pattern in the data even though no explicit feedback is given. The most common unsupervised task is clustering: detecting potentially useful clusters of input examples. One function of these algorithms might be detecting specific visual elements of a picture, in order to be able to accurately detect when that object is present in any photo. Often, unsupervised learning algorithms are partly supervised as well, and a small set of the data is labelled, in order to give it directions. An example of such an algorithm is *auto-encoders*.

In *reinforcement learning*, the algorithm learns from a series of reinforcements in the form of reward and punishment. For example, taking longer to be beaten or even achieving victory in a chess game can lead to a reward. The algorithm knows that the events that led up to this moment were good. However it has to decide which actions were most responsible for the reward.

[13] Mitchell (1997).

3 Machine learning

3.1 Mathematical Theory and Model

3.1.1 Introduction and Overview

As discussed previously, machine learning refers to the study and construction of mathematical algorithms with the goal of learning from and making predictions on data.

There are two main basic types of machine learning models on which other more complex algorithms are often based, each with their own applications: linear regression and logistic regression. Both are conceptually and in terms of implementation very similar, and a good understanding of linear regression will also provide the knowledge to successfully grasp the concepts of logistic regression.

The basic function of linear regression is to model the relationship between one or more explanatory or independent values, more often referred to as input values and x-values, and a dependent variable, frequently referred to as the output value and y-value.

A simple example for this would be the amount of rooms and the surface area of a house as x-values, and the price of this house as the y-value. These values serve as a training data for the algorithm, and are used to teach the program how to make accurate predictions. These x-values and the y-value together are referred to as a data or training set, or simply *the example data*. In order for the algorithm to work properly, the application of a great amount of data and thus a large dataset is crucial. The data corresponding to a single y-value will be referred to as a single training example.

Amount of rooms x-value 1	**Surface Area in m^2 x-value 2**	**House price in € y-value**
8	20	4000

Table 1: An example of a single training example

The algorithm will use this data to compute a hypothesis function using a process called gradient descent, which will attempt to minimize a cost function.

What this means is that the linear regression algorithm will calculate a fit to the data set which is accurate to the data and not excessively complex, in order to be able to correctly predict a y-value for new, unlabelled x-values (x-values without a y-value).

Contrary to linear regression, where the dependent variable (or y-value) is a scalar, in logistic regression this variable is categorical, meaning it can only be one of a limited and usually fixed number of possible values. The most common application of this model is for binary dependent variables, were it can only take on a value of 0 or 1, or 'true' or 'false'.

This concept is essential to the implementation of neural networks, and a more in depth explanation is provided in chapter 4.

3.1.2 Linear Regression

3.1.2.1 Data Sets

In order to gain an understanding of the mathematics of linear regression, it is useful to make use of an example dataset. In this case, the relation between the age and height of various boys between the ages of two and eight.

The x- and y-values will be denoted as $x^{(i)}$ and $y^{(i)}$ respectively, where i represents the number of the training example in the data set, ranging from 1 to m. A dataset accordingly can be seen as a list of m training examples $\{(x^{(i)}, y^{(i)}); i = 1, 2, \ldots, m\}$.[14]

[14] Ng (n.d.).

Example m	**Age in years x-value**	**Height in meters y-value**
1	2.066	0.779
2	2.368	0.916
3	2.540	0.905
4	2.542	0.906
5	2.549	0.939
...	...	...
50	7.9306	1.2562

Table 2: The data set described above.

The variable i represents an index into the training set and is not an exponentiation.

This example considers only one x-value per dataset, coupled with one y-value. This is strictly for the explanation and demonstration of the basic machine learning concepts in this paper. Generally, there will always be more than one x-value, called **features** in this context, used, in order to encompass more complex data and compute more accurate and therefore useful predictions. In such scenarios the x-values are considered vectors in $\mathbb{R}^n$, where n is the number of x-values per training example. This, however, results in calculations and data of higher dimension, which are impossible to display in a clear and concise manner and are thus disadvantageous for the goals of this chapter.

Keeping this in mind, the data of the example can be displayed more effectively and concise in a graph. Figures 3 and 4 show the data set and the calculated hypothesis function through it.

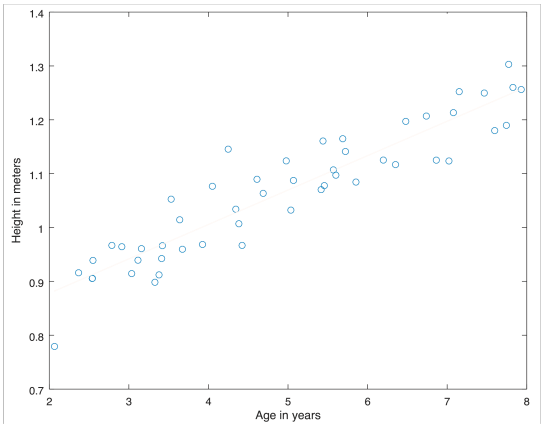

Figure 3: A data set showing the relation between age in years and height in meters of several boys between the ages of two and eight.

3.1.2.2 The Hypothesis Function

The goal of the algorithm will be to learn a function h that takes the given x-values in a training example to calculate a prediction for the y-value, where $h(x^{(i)}) \approx y^{(i)}$, meaning $h(x^{(i)})$ is a good as possible predictor of $y^{(i)}$. Traditionally, this function is called the hypothesis. Because the y-value is continuous, the learning problem of the example is called a regression problem.

The next step is defining the function h. The algorithm needs to factor in all the

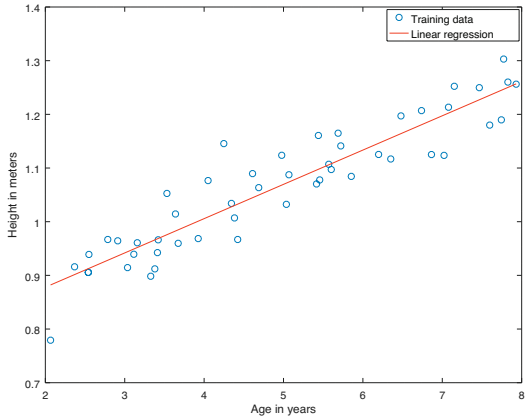

Figure 4: The data set with the calculated hypothesis function.

x-values for the prediction, while learning how to weigh each feature independently. Each x-value is therefore assigned to different variable, which the program will gradually adjust in order to increase the accuracy of the prediction. These parameters are noted as θ_a, where a is the number of the x-value it corresponds to. One extra parameter, θ_0, is also necessary to define h, as it functions as the starting position for the graph. The general function h will subsequently resemble:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

Where n is the amount of features.

Note that this function, when plotted with the x-values, will result in a linear graph (even in higher dimensions). Often, data will follow a different pattern, such as an exponential curve or a logistic curve. For this, the function can be altered accordingly to fit a specific relationship between the x-value and the y-value. For example, a term for an exponential relation might be $\theta_n(x_n)^2$, and a term for square root relation might be $\theta_n\sqrt{(x_n)}$. It is possible for the program to recognize these patterns itself, but the complex algorithms this requires extends beyond the scope of this paper.

When the first term, θ_0, is considered to be $\theta_0 x_0$ (with $x_0 = 1$), function h can be rewritten as:

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x_1$$

Where n is the amount of features.

The function for the example will simply be:

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 = \sum_{i=0}^{1} \theta_i x_i$$

3.1.2.3 The Cost Function

In order for the program to update the parameters and increase the accuracy of the prediction, it needs to know the margin of error of its previous prediction. For this

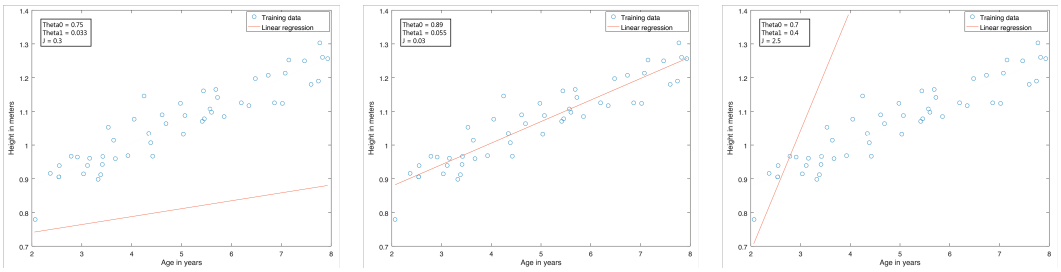

Figure 5: Several hypothesis functions with values of θ and J. The middle image shows the optimal values.

a square error term is used, which is defined as the difference between the two values squared. In this algorithm this function is referred to as the **cost function**. For a single training set it defines the cost function J:

$$J(\theta)^{(i)} = (h_\theta(x^{(i)}) - y^{(i)})^2$$

The cost function J for the whole training set can be defined as an average of the examples 1 through m. In addition, to simplify a forthcoming partial derivative, it is useful to multiply the function by $\frac{1}{2}$, for minimizing half of the error term will ultimately have the same result as minimizing the full error term:

$$J(\theta) = \frac{1}{2} \cdot \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

The goal of the algorithm will be to minimize this function, through modification of the variables θ; more specifically in the case of the example, the variables θ_0 and θ_1. The relation between the h function and the θ variables can be analysed with help of figure 5. Notice how different values of θ affect the hypothesis function and subsequently the cost function. The values of θ in the middle graph result in the lowest value of function J, indicating the function h to be a reasonable fit to the data.

3.1.2.4 Gradient Descent

To achieve the goal of minimizing the cost function, a process called **gradient descent** is used. In principle, this entails that the parameters θ are updated in order to improve the hypothesis function. An important attribute of this process is that it is repeated a considerable number of times, wherein the parameters are updated only to a small extent each iteration. This step as well as other similar algorithms which serve the same objective are the reason why the great computational speeds computers provide are essential for all machine learning algorithms.

For reasons of explanation, it is useful to visualize the cost function in a graph. The dependent variables of the function are the parameters θ. Because in the example these are comprised of θ_0 and θ_1, the figure is three dimensional.

As discussed previously, the goal of the algorithm is to minimize the cost function, which translates to finding a minimum in the plotted figure. As can be seen in figure 7, there won't always be just one minimum, but many different **local minima** can be present, while normally only one **global minimum** exists. The simple version of gradient

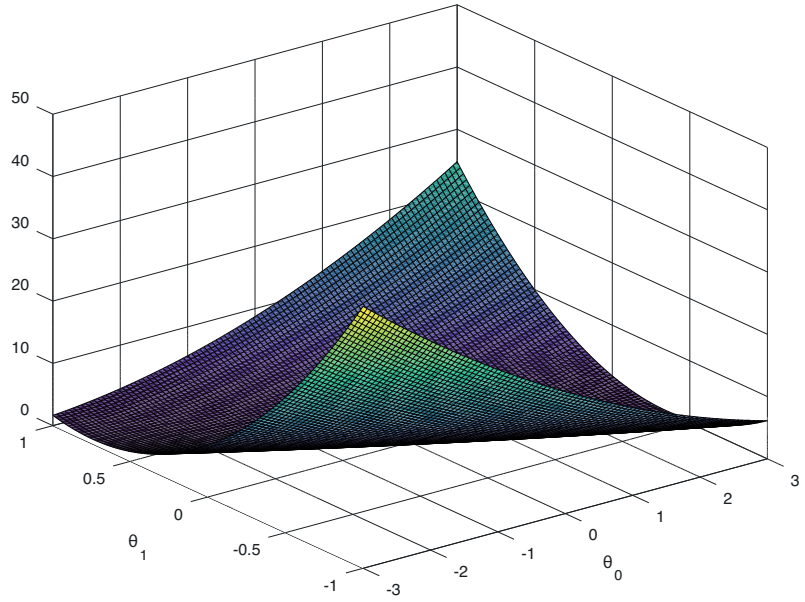

Figure 6: A visual representation of the relationship of the y value and two x values. The graph is three dimensional, since there are two features present. More features lead to graphs with dimensions of four and higher, and can therefore not be represented ordinarily.

descent in this introduction won't be able to distinguish between these. For now, however, this won't present any substantial problems. Additionally, a schematic rendering of the direction of the parameters and the cost function through the gradient descent algorithm is shown. Notice how each iteration reduces the outcome of the cost function, although in reality, the amount of iterations will be much larger instead of the 7 shown here.

Furthermore, to make finding the minimum for the example data set - which happens to only have one minimum - more apparent, it is beneficial to display the figure in a top-down view. From figure 8, the "route" the parameters will elapse can be determined easily.

In order for the program to know which direction is towards a minimum, a **partial derivative** of the cost function is used. With this, the slope of the graph is calculated, in order for the program to know which direction is "uphill" and which direction is "downhill". Additionally, this will cause the parameters to gradually slow down their descent as they get closer to the minimum, where the slope is equal to zero. Note that in this process all features are computed separately. With this, the update function for each separate parameter θ for one iteration can be defined:

$$\theta_j := \theta_j - \alpha \cdot \frac{\partial}{(\partial \cdot \theta_j)} \cdot J(\theta) \quad (for\ all\ j)$$

Where j is the number of the feature, from 0 to the total amount of features
Where $\frac{\partial}{(\partial \cdot \theta_j)}$ indicates a partial derivative over $theta_j$
Where α is parameter indicating the learning rate.

The := symbol indicates an assignment computation. Because θ_j is used in the definition as well, this means the original value of θ_j will be updated in accordance to this

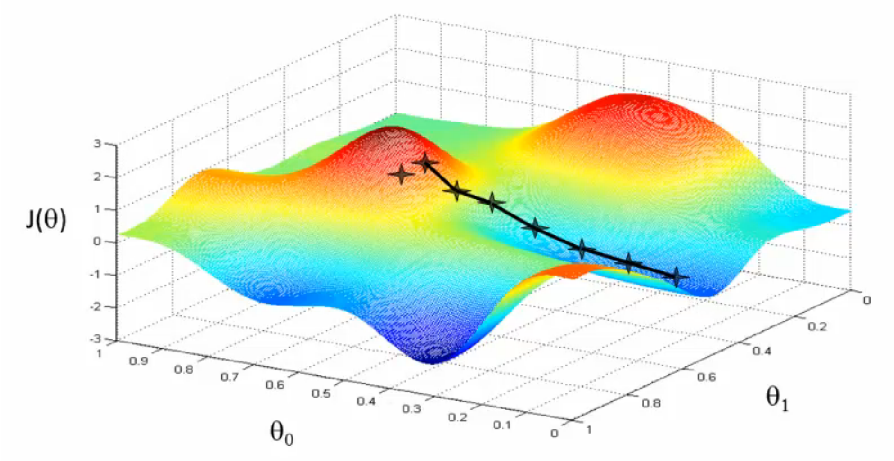

Figure 7: A data set containing multiple local optima. The cost function can be minimized to multiple of the optima, depending on where it is initiated. The crosses on the graph indicate the path the gradient descend algorithm might take in order to end up in such an optimum. Note that, in reality, the steps the gradient algorithms will take to end up at an optimum are much smaller than depicted here.

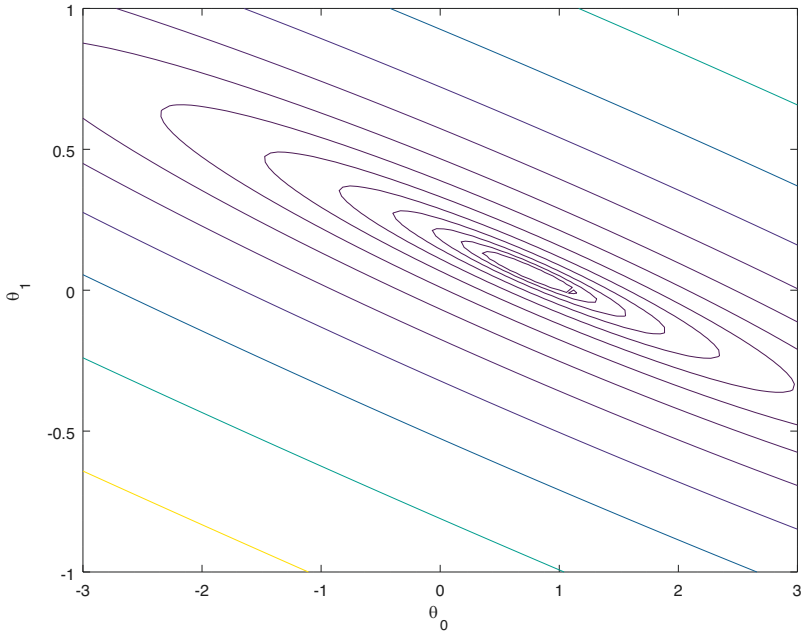

Figure 8: A top-down view of figure 6. The different colours indicate the value of the cost function. Observing a three-dimensional graph in this manner may simplify finding the optima, as well as the path gradient descend will take. The global optima is located at the centre of the ellipses.

original value.

Choosing a desirable learning rate α is an important part in this process, as it defines to which degree the parameters are updated each iteration. When it is too small, the program will take too long to compute, and when it is too large, the program might have trouble converging to a minimum, as the parameter will continually go over the it. Because all features are computed separately, it is possible to visualize this process.

The next step is to define $\frac{\partial}{(\partial \cdot \theta_j)} \cdot J(\theta)$. The inclusion of the previously defined cost function J gives:

$$\frac{\partial}{(\partial \cdot \theta_j)} \cdot \frac{1}{m} \cdot \frac{a}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

After differentiation, this formula becomes:

$$\frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

With these and as a final step, we can redefine the update formula:

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (for\ each\ j)$$

3.1.2.5 Abridgement

In practice, the program first initializes the parameters θ to random values (often 0). Following this, it will compute the hypothesis function and update the parameters for i amount of iterations using the gradient descend algorithm with learning rate α, using the following formulas, until convergence. The outcome will be a hypothesis function which accurately predicts the y-value for new x-values.

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i \cdot x_i$$

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (for\ each\ j)$$

3.1.2.6 Vectorized Implementation

For the machine learning to be as efficient as possible, the amount of calculations required for each iteration should be minimized. Especially for increasingly complex and large datasets, only a tiny decrease in the duration of a single iterations can lead to huge improvement for the whole process. An important aspect for essentially all machine learning algorithms is the vectorized implementation. Making use of a heavily optimized linear algebra library and matrix multiplications, an vectorized implementation creates a significant performance increase in any of the machine learning algorithms. For most programming languages, such a linear algebra library is readily available.

In basic terms, a vectorized implementation simply refers to an implementation where the computer calculates all learning examples 'at once', by positioning them in a single matrix. The hypothesis function will therefore be implemented as the product of two

matrices, x^T and θ, instead of the sum of the products of two integers, θ_i and x_i, for all features and subsequently all training examples. Because of the exploitation of the matrix functionality, these computations are effectively identical. The matrix multiplication, however, is many times faster than a relatively slow for-loop and more compact in the programme as well. Furthermore, by arranging the data as a matrix, data sorting and storing can be done much more efficiently and orderly.

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i \cdot x_i \quad (for\ all\ training\ examples) = x \cdot \theta$$

Note that the $\sum$ will be represented in code as a for loop. It is important for the two matrices to be the appropriate proportion, in order to allow for the matrix multiplication. The matrix x consists of all x-values of the training examples, and will be of $\mathbb{R}^{m \times n}$ dimensions, where m is the number of training examples and n the number of features (including x_0). Note the multiplication symbol expresses dimensionality, and does not indicate actual multiplication. The values for the weight values θ have just one value for each iteration, and are thus constant across the multiplications with all training examples in each iteration. θ will accordingly be of $\mathbb{R}^{n \times 1}$ dimensions. Following the multiplication, this computation will result in a matrix of dimensions $\mathbb{R}^{m \times 1}$ as well, containing the hypothesis value for all training examples.

$$\begin{array}{c|c} & \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \\ \hline \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} \\ x_0^{(2)} & x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots & \vdots \\ x_0^{(m)} & x_1^{(m)} & x_2^{(m)} \end{bmatrix} & \begin{bmatrix} x_0^{(1)} \cdot \theta_0 + x_1^{(1)} \cdot \theta_1 + x_2^{(1)} \cdot \theta_2 \\ x_0^{(2)} \cdot \theta_0 + x_1^{(2)} \cdot \theta_1 + x_2^{(2)} \cdot \theta_2 \\ \vdots \\ x_0^{(m)} \cdot \theta_0 + x_1^{(m)} \cdot \theta_1 + x_2^{(m)} \cdot \theta_2 \end{bmatrix} \end{array} = \begin{bmatrix} h_\theta^{(1)}(x) \\ h_\theta^{(2)}(x) \\ \vdots \\ h_\theta^{(m)}(x) \end{bmatrix}$$

The matrix multiplication table of the matrices x and θ, wherein the hypothesis values for all training examples are calculated. Following the standard rules for matrix multiplication, the first element of each row of the bottom left matrix is multiplied with the first element of each column of top right matrix, after which the product of the next elements of the respective row and column is added, until all each row has been included. In the bottom right matrix the result of this multiplication can be seen. This result is equal to the hypothesis function, confirming the validity of the use of the matrix multiplications of a vectorized implementation.

Accordingly, the cost function J can be updated for use of the the matrix h_θ, and a matrix y of the y-values of the training examples. Because, instead of with a one-by-one approach, all h-values are used simultaneously the sum can once again be omitted. The function can be rewritten as follows.

$$J(\theta) = \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{m} \cdot (h_\theta - y)^2$$

Note that the second exponent illustrates an element-wise exponentiation.
Lastly, since the the weight values, θ_0 to θ_n, have been converted to a matrix form as well, the weight update function must be updated similarly. It is once again possible to

exploit the functionality of matrix multiplication for a more efficient computation, as the x-value present in the update function can be used to conveniently create a matrix with the corresponding dimensions.

$$\begin{array}{c|c} & \begin{bmatrix} h_\theta^{(1)}(x) \\ h_\theta^{(2)}(x) \\ \vdots \\ h_\theta^{(m)}(x) \end{bmatrix} \\ \hline \begin{bmatrix} x_0^{(1)} & x_0^{(2)} & \dots & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \end{bmatrix} & \begin{bmatrix} \Delta\theta_0 \\ \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \end{array}$$

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (for\ each\ j) \longrightarrow \theta =: \theta - \alpha \cdot (\frac{1}{m} \cdot x^T \cdot h_\theta)$$

One dimensional integers α and $\frac{1}{m}$ are multiplied element-wise with the according matrices. The T in this formula represents the transpose of the matrix x. Because of standard matrix multiplication notation, the transposed matrix x^T is positioned before the h_θ term in the formula. With this, the complete learning algorithm can be represented with the following formula:

$$\theta := \theta - \alpha \cdot (\frac{1}{m} \cdot x^T \cdot ((x * \theta) - y)^T)$$

3.1.2.7 Sidenotes

While more data usually means more accuracy, because it is often possible that no correlation between the x and y values exists, all x-values should always be carefully considered and analyzed. Even when no correlation exists, the algorithm will still factor the data in the calculations, which will result in less accurate results.

This paper won't go into detail of many other, more complex aspects of linear regression, on account of space constraints. If a more in depth understanding is required, it is recommended to read up on these concepts, such as feature scaling and the selection of a good learning rate to name a few, from other sources.

Furthermore, it is important to realize these forms of machine learning, linear and logistic regression, are relatively basic and are not used a lot nowadays in their original form. Many more complex and efficient algorithms have since taken their place. These algorithms are still based on the same concept however, and basic understanding of these concepts is crucial in the field. Neural networks are an example of this, since they are essentially a variation on logistic regression.

3.2 Implementation in the Programming Language Octave

3.2.1 Introduction and Overview

The implementation of the linear regression algorithm is relatively simple, and only consists of a few lines. This will be a step by step commentary on a basic implementation as described in chapter 3.1 of this paper. As mentioned previously, the used programming language is GNU Octave, but any programming language is of course suitable, provided it has an linear algebra library available. Octave comes pre installed with a greatly optimized linear algebra library and provides built-in functionality for graphs and other useful figures

as well. For this program the previously described example of chapter 3.1 is used once again.

3.2.2 Non-vectorized Implementation in Octave

Before examining a vectorized implementation, it is important to understand a basic implementation. Keep in mind a non-vectorized implementation is rarely used anymore and can make for a somewhat disconcerting program.

The first step is loading the training examples and defining variables. The dataset contains 50 examples, meaning the x and y values, which are defined as an array, will have a length of 50. This example considers the data to be available as an array in a *.dat* file, although the initiating of this data depends on the manner in which it is stored on the computer. The x_0 is defined as 1.

Next, the learning rate α (alpha) and the amount of iterations i (max_iterations) are given the values 0.07 and 1500 respectively. Both values are changeable, and optimal values differ per situation. Unfortunately, the techniques for choosing the best values for these will not be discussed in this paper. For now, realize a smaller value of α will give the algorithm more accuracy, at the cost of speed. The amount of iterations is the amount of times gradient descend will run, and a higher value ensures the hypothesis function has converged to an optimal value, at the cost of the programme running for a greater amount of time.

The starting values for the weights θ are initiated as well, and are given a value of 0, although any value could be used. As a reminder, the example dataset contains only one feature, meaning 2 different θ values need to be initiated, θ_0 (theta0) and θ_1 (theta1).

```
%Inport Training Examples
x0 = 1;
x1 = load('x.dat');
y = load('y.dat');

%Set variables
m = length(y);
alpha = 0.07;
max_iterations = 1500;
theta0 = 0;
theta1 = 0;
```

The next step is defining the hypothesis function. From the general formula, $h_\theta(x) = \theta_0 \cdot x_0 + \theta_1 \cdot x_1 \ldots \theta_n \cdot x_n$, follows for the example the formula $h_\theta(x) = \theta_0 \cdot x_0 + \theta_1 \cdot x_1$. Because each training example has a hypothesis value, the function must be computed m amount of times. This is achieved by use of a for-loop, in which the hypothesis function is defined as the array h[i].

```
%The hypothesis function
for i = 1:m
  h[i] = theta0 * x0 + theta1 * x1[i];
end
```

The cost function will be defined as the already partially differentiated formula $\frac{1}{m} \cdot \sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$. For the sum, a for loop can be used, which goes over all the index values of $h[i]$ and adds them together. The $\frac{1}{m}$ is added in the next step, in order to correctly calculate the outcome. Because the formula has a weight parameter specific term ($x_j^{(i)}$), a separate cost function for each feature is necessary. In the programme, this

will be defined as follows.

```
%The cost function
J0 = 0;
J1 = 0;
for i = 1:m
  J0 = J0 + (h[i] - y[i]) * x0;
  J1 = J1 + (h[i] - y[i]) * x1[i];
end
```

Lastly, the weight parameters are updated according to the the update function $\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \cdot \sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \quad (for\ each\ j)$, in which $\sum_{i=0}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ is replaced with the in the programme defined J. Keep in mind this variable is different than the in section 3.1 defined cost function J. The parameters theta0 and theta1 will be updated as follows.

```
%The update function
theta0 = theta0 - alpha * 1/m * J0;
theta1 = theta1 - alpha * 1/m * J1;
```

The described programme functions as a single iteration of the gradient descent algorithm. In order to finish the full linear regression algorithm, a for-loop is added which encompasses the defined functions. Because for each iteration a new error term is necessary, the variables J0 and J1 need to be reset. The full programme will look as follows.

```
%Inport Training Examples
x0 = 1;
x1 = load('x.dat');
y = load('y.dat');

%Set variables
m = length(y);
alpha = 0.07;
max_iterations = 1500;
theta0 = 0;
theta1 = 0;

%Linear Regression
for i = 0:max_iterations
  for i = 1:m
    h[i] = theta0 * x0 + theta1 * x1[i];
  end

  J0 = 0;
  J1 = 0;
  for i = 1:m
    J0 = J0 + (h[i] - y[i]) * x0;
    J1 = J1 + (h[i] - y[i]) * x1[i];
  end

  theta0 = theta0 - alpha * 1/m * J0;
  theta1 = theta1 - alpha * 1/m * J1;
end
```

3.2.3 Vectorized Implementation in Octave

This basic implementation presents a number of problems. For instance, a high number of features can lead to an incredible amount of theta variables, resulting in a high number of functions or, alternatively, nested for-loops, both of which decrease the clarity and increase inefficiency of the code. A vectorized implementation will solve these problems as well as improve performance significantly, as has been described in section 3.1. Additionally, because of the nature of this implementation, the programme is compatible with all possible number of features.

To start, the x, y and h values previously defined as arrays, can be converted to matrices. The variables y and h both contain one value for each training example, resulting in matrices dimensions $\mathbb{R}^{m\times 1}$, and $\mathbb{R}^{50\times 1}$ for the example. The matrix x includes all x values, and will accordingly have a width of $n+1$ (all features and x_0), resulting in a matrix of $\mathbb{R}^{m\times n}$, and $\mathbb{R}^{50\times 2}$ for the example. The first column is made up of the x_0 values, and all its elements equal one. This column won't be part of any dataset, and needs to be added to the matrix. This can be achieved with an Octave function *ones*(), which will return an matrix of ones of the specified dimensions.

```
%Import Training Examples
x = load('x.dat');
y = load('y.dat');

%Set variables vectorized
m = length(y);
x = [ones(m, 1), x];
```

For each feature, one weight parameter θ_n needs to be defined. As described in section 3.1, this is achieved in a matrix of dimensions $\mathbb{R}^{n\times 1}$, and $\mathbb{R}^{2\times 1}$ for the example. Once again, the values of θ (theta) are initiated as zero. Because the height of the matrix needs to equal the width of the matrix x, the Octave function *width*() (which, as expected, returns the width of a matrix) is used followed by a transpose. This gives a matrix of the right dimensions. With the Octave function *zeros*(), all elements of the matrix are set to zero.

```
theta = zeros(size(x(1,:)))';
```

Next, the hypothesis function can be defined. All h values are computed at once, and no for-loop is necessary.

```
%The hypothesis function
h = x * theta;
```

The cost function is adjusted in a similar manner, and computes all values at once. Additionally, the different values for J can be represented in one function. A matrix multiplication with x^T will result in a matrix of the same dimension as the matrix theta, namely $\mathbb{R}^{n\times 1}$, and $\mathbb{R}^{2\times 1}$ for the example. In order to correctly execute this multiplication, the matrix x^T needs to be positioned before the matrix h. In Octave, a transpose is implemented with use of an apostrophe, resulting in x^T being represented as x' in the programme. For consistency, the $\frac{1}{m}$ term is again moved to the update function.

```
%The cost function
J = x' * (h - y)
```

Subsequently, the update function updates all weight parameters θ, present in different rows of the matrix theta, at once. The integers α (alpha) and $\frac{1}{m}$ are multiplied element-wise with the matrix.

```
%The update function
theta = theta - alpha * (1/m) * J;
```

Once again a for-loop is added to finish the linear regression algorithm.

```
%Linear regression vectorized implementation
for i = 1:max_iterations
  h = x * theta;
  J = x' * (h - y);
  theta = theta - alpha * (1/m) * J;
end
```

As a final step, the functions can be combined to shorten the whole algorithm to a single line.

```
%Import Training Examples
x = load('x.dat');
y = load('y.dat');

%Set variables vectorized
m = length(y);
x = [ones(m, 1), x];
alpha = 0.07;
max_iterations = 1500;
theta = zeros(size(x(1,:)))';

%Linear regression vectorized and shortened implementation
for i = 1:max_iterations
  theta = theta - alpha * ((1/m) * x' * ((x * theta) - y));
end
```

Even for a linear regression algorithm with only two features, the vectorized implementation manages to shorten and optimize the programme by a considerable amount. This example served only as an easy introduction to the idea. The concepts of vectorized implementation play a big role in the implementation of neural networks and will return in the following chapter.

4 Artificial Neural Networks

4.1 Architectural Model and Mathematical Theory

4.1.1 Introduction and Overview

Using the aforementioned methods, finding a linear relation among certain data points can be easily achieved. What should one do however, if there is no such linear relation? In that case, other forms of regressional analysis, such as *polynomial regression* can be used. Even then, the analysed data points ought to adhere to a relation which can be described using a polynomial equation or any other type of equation prescribed by the form of regression. Furthermore, what if the data points are defined by an excessive amount of independent variables? Using ordinary regression, this results in a multi-dimensional relation which would become exponentially more difficult to calculate.

There is, however, an alternative method for solving these kinds of problems. Using *Artificial Neural Networks*,[15] complex relations in large amounts of data can be found. Artificial neural networks are currently a popular and effective form of machine learning. Artificial Neural Networks (*ANN*) mimic *Biological Neural Networks* (*BNN*) in their basic functioning. A BNN consists of many neurons ,[16] the neurons have dendrites and axons. When a neuron fires it sends a signal along its axon which connects to the dendrite of another neuron. A neuron fires if the sum of the input signals is high enough to surpass a certain threshold or in the case of a sensory neuron if it is stimulated by a physical stimulus such as light, temperature or pressure. motor neurons control muscles and function as the output of a BNN. This biological analogy can be used to describe the form and function of the components of an Artificial Neural Network.

Whereas a BNN has sensory neurons, an ANN has so called input nodes. These nodes, just as their biological counterparts, also require a certain amount of stimulation in order for them to relay a signal. The links between nodes in an ANN are handled by *weighted connections*, as opposed axons and dendrites connecting neurons in a BNN. The function of the biological motor neuron

4.1.2 The Network Architecture

When trying to build an ANN to solve a supervised learning task - let us say a value y has to be predicted based on a value x - there are a few important aspects that need consideration.

First of all, these aspects are comprised of the fundamental building blocks of a ANN. As mentioned earlier, these components are the nodes and weighted connections but also crucial are the layers in which they are arranged. Second of all, the properties of these components should be considered.

To get a clear picture of Artificial Neural Networks, its components, their properties and functions will be explained using the example network shown in figure 9.

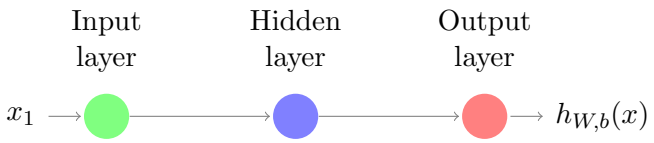

Figure 9: A three-layered model of an ANN with in each layer one node.

While this might seem like a rather simple network, which can not really be disputed.

[15] *Unsupervised Feature Learning and Deep Learning* (2013).
[16] Skaggs (2013).

It does employ the previously specified components in the various forms they can assume.

4.1.2.1 Layers

One of the first noticeable properties of an ANN, as is demonstrated in the aforementioned example, is the composition of the assorted layers in the network. While there are only three layers present in this example, it is not uncommon for a network to have a vast amount of them. The amount of layers needed is determined by the task the ANN has to perform and thus has to be decided with the networks function in mind. For too little layers would limit the networks function. Too many layers however, is not desired either considering it will make computation of the network slower and exceedingly more difficult. Finding a correct balance is imperative and this balance should also be achieved when considering the number of nodes in each layer and the inter-node connections.

In ANNs there are three types of layers, all of which are shown in the example of figure 4.1. Each layer contains nodes that fulfil a different function.

The first layer is the *input layer*. This layer contains input nodes and is thus responsible for providing the ANN with data. In an ANN there is only one input layer present.

The final layer present in an ANN is the *output layer*. This layer contains, as one might presume, the output nodes and handles the various outputs of the network. As with the input layer, in each ANN there is only one of these layers present.

The second layer is called the *hidden layer*. Of this layer there are usually more present, situated between the two outside layers. One of which is the input layer and the other one will be discussed hereafter. The hidden layers are comprised of hidden nodes. The layer is called 'hidden', since the user of the network does not directly interact with it, as opposed to the input and output layer, where data is provided or received respectively. The hidden layer is responsible for most of the computation—which depends on its size and the amount of hidden layers.

4.1.2.2 Nodes

Nodes, represented in the model above by the coloured circles, are small and relatively simple computational units. In the example network, three of the four different types of nodes are displayed. The displayed three nodes have been briefly introduced in the preceding paragraphs. Each of these distinct types of nodes has its own functions and treats data in a slightly different way.

The first type of node, represented by the green circle, is the input node. This node acts as the input of the network. By itself, the input node does not perform any computation; It merely takes the input data given to the network and passes it on to the next layer.

Somewhat similar to the input node is the bias node, a node not included in the example above but not less important. This node however, has a default value of 1. The node solely passes this 1 on to the nodes in the next layer its connected with via weighted connection. The value of a bias node is sometimes given as a separate weight itself, for any number is the solution of its multiplication with 1.

After this, the hidden node is at play, represented by the blue circle. These kinds nodes of receive inputs from the layer that came before it via the connections it has with these previous nodes. The node takes these inputs, sums them up and applies an activation function to this sum. The mathematical nature of the activation function will be discussed following section concerning the mathematical theory of ANNs. The solution of the activation function is passed on to the connected nodes in the next layer.

The final node in the network, represented by the red circle, is the output node. This node, just as the hidden node, takes the sum of its inputs and applies an activation function to this sum. However, it does not pass the result of the activation function on to nodes in the next layer. Instead its result will be given to the user as the final output of the ANN.

4.1.2.3 Weighted Connections

The function of nodes having been elaborated upon, just leaves the so called weighted connections between the nodes. These connections are represented in the example by small arrows going from one node to another. The function of these connections is to take the output of a node, multiply it by a certain weight—this is just a number prescribed when constructing a network—and server the result of the multiplication as one op the inputs of the node in the next layer.

4.1.2.4 Network Topology

As has been pointed out in the previous paragraphs briefly, deciding the structure of an ANN to be used for a certain task is essential. When talking about the placement of nodes is various layers and the different connections between nodes this is usually described as the topology of a network. The subject of network topology will be further discussed in chapter 5.

4.1.3 Underlying Mathematics

Now that the description of the ANN architecture is out of the way, explaining the mathematical basis is left to be done. For, while an ANN might not look like it, it is just a mathematical algorithm one is able to represent with one single equation. However, for the sake of clarity, this formula will be slit up into simpler and easily digestible parts linked to the various components of the ANN. When discussing these formula's, some symbols will be used to represent certain variables and parameters. These symbols are each introduced in paragraph they are featured in first.

4.1.3.1 Weight Computation

While in the structural theory nodes have been discussed before discussing connections, the choice has been made to turn this order around when describing the mathematical theory, for knowledge about weight computation is crucial to understanding node computation.

Each weighted connection is given a weight, represented as $W_{ij}^{(l)}$. Here i represents the target node, j represents the node of origin, (l) the layer of origin and thus $(l+1)$ the destination layer. This $W_{ij}^{(l)}$ is multiplied with the output of node j in layer (l) to get one of the inputs of node i in layer $(l+1)$.

The weight of connections originating from bias nodes is represented by $b_i^{(l)}$ where i represents the destinations node and (l) the layer of the destination node.

4.1.3.2 Node Computation

In the above-mentioned description of hidden and output nodes a little bit of light has been shone upon the theory of the activation function. Its algebraical properties have however not been explained yet.

As mentioned, the hidden and output nodes receive multiple weighted values from nodes situated in the previous layer. These inputs are summed by the node. This sum of

values is algebraically represented by $z_i^{(l)}$, where i represents the node and (l) the layer. Thus:

$$z_i^{(l)} = \sum_{j=1}^{n} W_{ij}^{(l)} + b_i^{(l)}$$

$f(z)$ is the activation function, also know as the sigmoid function. The equation of the sigmoid function is as follows:

$$f(z) = \frac{1}{1 + e^{(-z)}}$$

This function results in the following graph:

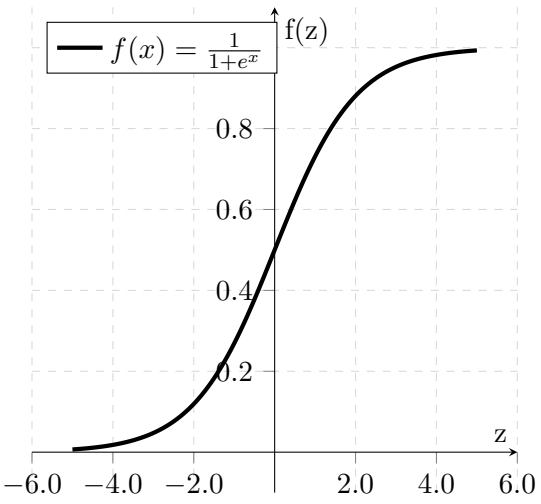

Figure 10: A graphical plot of the sigmoid function.

The nature of the sigmoid function, and therefore the node activation, is to return a 0 when the sum of inputs is below a certain value and return a 1 of it is above this value. Ordinarily this value would be around $z_i^{(l)} = 0$, but by using bias nodes this value can be shifted, due to the bias weight always being added during the nodes summation.

Some implementations of neural networks, like the one used for our experiment, use activation functions derived from the sigmoid functions. The function we use, which will be discussed later is:

$$f(z) = (\frac{2}{1 + e^{(-4.9z)}}) - 1$$

and results in the following graph:

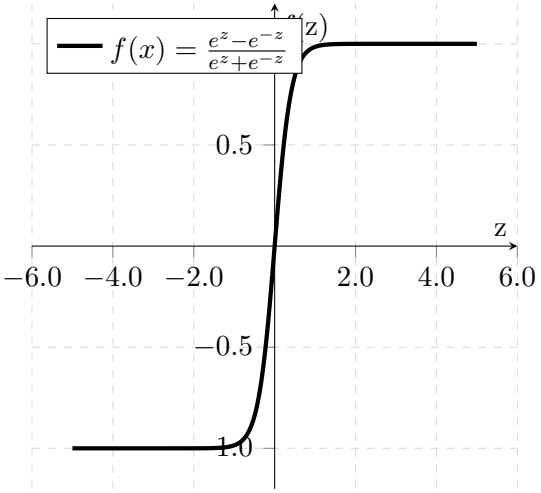

Figure 11: A graphical plot of the modified sigmoid function.

In ANN theory the activity of a node—the result of its activation function—is usually represented as $a_i^{(l)}$. i represents the number of the node inside its layer and (l) represents the number of the node's layer in the network. Thus the equation describing a layers activation is as follows:

$$a_i^{(l)} = f(z_i^{(l)})$$

4.1.3.3 The Hypothesis

Finally the outcome of the network. This value is described as $h_{W,b}(x)$, because its a value based on the x inputs and influenced by W and b the weight and bias parameters respectively. $h_{W,b}(x)$ is also know as the hypothesis function but can also be seen as the activation of the output node.

4.1.3.4 Vectorized Implementation

As with linear regression, using a vectorized implementation, ANN computation can be sped up drastically. When using a vectorized implementation the independent x_i inputs, the weights and the biases are all put into matrices. Then matrix multiplications are performed which automatically result in summations; See 3.1.2.6.

4.1.4 Backpropagation

For neural networks, the learning process is quite different from the previously described gradient descent algorithm, although they essentially serve the same purpose and share some similarities. The defining difference is that backpropagation has to calculate a different update value for all weights in the network. It goes through the network working backwards, hence the name, updating the weights of each layer using an error term in the next layer.
The parameters W and b are updated using gradient descent, using the following formulae:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)$$

The key step that proves difficult is computing the partial derivatives in the above equations. This is where the *backpropagation* algorithm comes in, an efficient way of computing these partial derivatives.

Backpropagation first runs a forward pass on the network, computing all activations including the output of the hypothesis $h_{W,b}(x)$. Then, we want to figure out how much every node in the network is responsible for the error in the output by computing an error term, $\delta_i^{(l)}$. This is fairly easy to compute for output nodes, but a little more tricky for hidden nodes.
For each output node, the following equation is used to compute the error term.

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l}} \frac{1}{2}(y - h_{W,b}(x))^2$$

$h_{W,b}(x) = f(z_i^{(n_l)})$, so $y - h_{W,b}(x)$ can be written as $y - f(z_i^{(n_l)})$. In this way, the derivative can be written as follows (derived using the chain rule, where $f'(z_i^{(n_l)})$ is the derivative of the activation function — this is the reason activation functions need to be derivable when using backpropagation):

$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)} \cdot f'(z_i^{(n_l)})$$

For every node not in the output layer, the following is computed, where s_{l+1} is the number of nodes in the next layer.

$$\delta_i^l = (\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)}) \cdot f'(z_i^{(l)})$$

Then, the desired partial derivatives for a training example (x, y) can be computed:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}$$

Using the derivates gradient descent can progress as normal using the equations mentioned at the start of the paragraph on backpropagation, where the weights for a certain node in a layer are updated. Backpropagation is a notoriously difficult algorithm to explain, understand and implement,[17] so if this explanation did not quite help you grasp it: do not worry, read other explanations from other sources — we had to, too.

4.2 Implementation in the programming language Octave

Similar to linear regression, it is useful to go through the implementation of a simple neural network in order to fully understand the actual learning process. Once again, the programming language Octave will be used. This chapter will only go over the vectorized implementation. The following neural network, illustrated in figure 4.4, will be trained to act as a simple AND gate.

[17] UFLDL (2013).

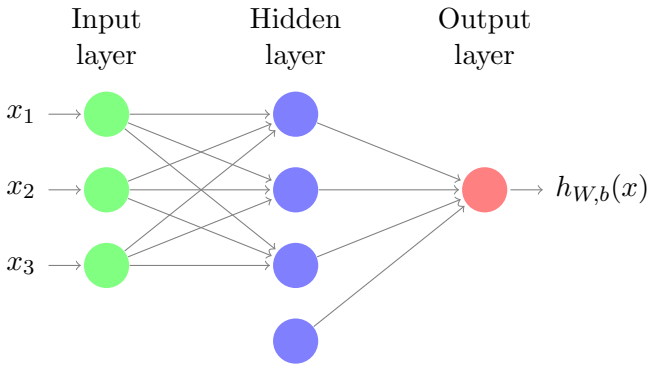

Figure 12: A graphical representation of the aforementioned ANN. In this network the bias node in the input layer is represented by x_3 and the bias node in the hidden layer is the bottom most blue node.

4.2.1 Variable Initiation

To start, a data set initialized. For a AND gate two x values per training example are required, both of them being boolean. In this example, each value of x is loaded as a separate matrix and added together later. All x matrices have 1 column and m rows. There are $2^2 = 4$ possible combinations of the two x values, meaning $m = 4$. The matrix x_1 and x_2 therefore have a dimensionality of $\mathbb{R}^{4\times1}$. Accordingly, because there is only a single output node, the y matrix has the same dimensionality. The values correspond to a AND gate as follows:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Table 3: A table showing the different x_1 and x_2 values and their result when inserted in an AND gate.

```
%The data set
x1 = [0; 0; 1; 1];
x2 = [0; 1; 0; 1];
y =  [0; 0; 0; 1];
```

The weights are initiated as a matrix of the dimensions corresponding to the amount of connections in a specific layer. The dimensions are equal to the product of the amount of nodes in the layer, excluding the bias node, and the amount of nodes in the next layer, where the weights in each column and row correspond to a single node respectively. For the example, the dimensions of the weights matrix of layer 1, W^1, is $\mathbb{R}^{2\times3}$, according to to the 2 nodes in layer 1 and the 3 nodes in layer 2. Similarly, the weight matrix of layer 2, W^2, is $\mathbb{R}^{3\times1}$. For clarity, the number present in the variable names indicates the layer number, instead of for x_1 and x_2, where it indicates the node number in the input layer. As described, the starting weights need to be initiated to different values for the purpose of symmetry breaking.
Next, the weights of the bias nodes are initiated, for which there is a values for each node in the next layer. The dimensionality of the matrix for the first layer, b_1, is $\mathbb{R}^{1\times3}$ and

the matrix for the second layer, b_2, is $\mathbb{R}^{1\times1}$. Because the bias nodes always output a value of 1, these weights equal the complete output of the bias nodes to the next layer. The matrices are therefore referred to simply as the bias node outputs, and are used in upcoming computations without a multiplication. The Octave function stdnormal_rnd() will generate a matrix of specified dimensions with random elements of normal distribution. The code will look as follows:

```
%Initialisation weights
W1 = stdnormal_rnd(2,3);
W2 = stdnormal_rnd(3,1);
b1 = stdnormal_rnd(1,3);
b2 = stdnormal_rnd(1,1);
```

The variables are initialized afterwards. α (alpha) is initiated to 1 and λ (lambda) is initiated to 0, meaning both won't have any effect on the algorithm. Since this is a simple network, it is not required to contemplate the values of these variables. Like what has been stated in chapter 3, the process of determining these values can be very complicated and will not be discussed in this paper. The amount of iterations the network will run is defined as the variable max_iterations, and is set to 5000, which is more than enough for this simple task. The amount of training examples m is equal to the amount of rows of the x matrices, and it initiated using the Octave functions rows(), which counts the amount of rows of a given matrix.

```
%Initialisation variables
alpha = 1;
labda = 0.01;
max_iterations = 5000;
m = rows(x1);
```

4.2.2 The Sigmoid Activation Function

The sigmoid activation is defined as a function, since the program needs to refer to it multiple times. Because this is a vectorized implementation, the notation ./ is used, indicating a element wise division. exp(-z) means an e exponent, or e^{-z} in this case. Similar to the sigmoid formula, the function is defined as f(z).

```
%sigmoid function
function s = f(z)
  s = 1./(1 + exp(-z));
end
```

4.2.3 Forward propagation

First the forwards propagation is implemented, where each layer is calculated one after the other. First the output of all the nodes except the bias node in layer 1, the input layer, is defined as $a^{(1)}$, where a indicates the output and (1) the layer. Naturally, this value consists of the input values x_1 and x_2, since a vectorized implementation is utilised, $a^{(1)}$ is defined as a matrix, where each column contains the outputs of a single node, either x1 or x2, and each row the data of a single training set. With this matrix and the matrices with the weights of the first layer, W1 and b1, the inputs for the nodes in the next layer are computed. Following the connections in the diagram and the formula $z = [sum](a * w) + b$, the input to each node in layer two equals the sum of the outputs of the nodes in the previous layer and the bias node. This means each node in layer 1

needs to be computed for each node in layer 2, which in the case of this example means the value of x1 and x2 are both necessary 3 times. With another exploitation of matrix multiplication this can be achieved very efficiently and without use of a for loop. The outputs of the bias node , which in this layer consists of a matrix of the dimensions , is added to the matrix thereafter. a1 will be defined as follows.

```
for i = 1:maxi

    %Forward propagation
    %Layer 1 (input)
    a1 = [exx1, exx2];

    %Layer 2
    z2 = a1 * W1 + b1;
    a2 = f(z2);

    %Layer 3 (output)
    z3 = a2 * W2 + b2;
    h = f(z3);
    a3 = h;

    %Backward propagation
    % Delta values
    delta3 = -(exy - h)   .* (a3 .* (1 - a3));
    delta2 = (delta3 * W2') .* (a2 .* (1 - a2));

    W2 = W2 - alpha * (a2' * delta3 * 1/m); % + lambda * W2);
    W1 = W1 - alpha * (a1' * delta2 * 1/m); % + lambda * W1);
    b2 = b2 - alpha * (sum(delta3) * 1/m);
    b1 = b1 - alpha * (sum(delta2) * 1/m);

end
```

4.2.4 Backward Propagation

Next, the δ values are computed. As described in the previous section, the δ values of any specific layer can be seen as an error term for that layer, are are used to update the weights of the connections to that layer. The δ of the last layer, delta3 in the example, is defined first, and is calculated according to the formula delta output layer formula.

Subsequent δ values are defined using the δ of the the following layer using the formula de volgende formula. All the δ values are calculated using the derivative of the sigmoid function, which equals the formula de afgeleide van de sigmoid. Because the values of $f'(z^{(l)})$ are already defined as a^l, those can be used instead, in order be as efficient with computational resources as possible.

```
%extra - implementation explanations
%Update terms - partial derivatives
DeltaW2 = a2' * delta3;
DeltaW1 = a1' * delta2;
Deltab2 = sum(delta3);
Deltab1 = sum(delta2);
```

The values of δ are used to update the weights. Because the connections to a layer are defined as weights of the previous layer, the value of delta of any layer is used to update

the previous layer, which for the example entails delta3 is used to update $W^{(2)}$ and delta2 is used to update $W^{(1)}$.

For the update terms, DeltaW and Deltab, all training examples of the data set need to be added together. For the weight matrices, matrix multiplication can once again help simplify the process. In this calculation, the share each node had in the error of the next layer's node is added to the computation as well, by multiplying the weights with the output values z of the same layer. Because these values equal the input values of the nodes in the next layer and more divergent values will result in a higher values, the weights will be updated to different lengths accordingly. Since the bias nodes are defined only as weights and always ensue part of the error of the next layer, they are updated with the full delta value. Again a sum of all training examples is calculated.

These values are then multiplied element wise with α (alpha) and $\frac{1}{m}$, to lower the update terms to increase accuracy. Lastly, the weights and bias weights are updated. The term including the weight decay parameter λ (lambda) is omitted (only present as comment), since, as has been mentioned before, it won't improve a small network like the one from the example.

```
%Update functions
W2 = W2 - alpha * (DeltaW2 * 1/m); % + lambda * W2);
W1 = W1 - alpha * (DeltaW1 * 1/m); % + lambda * W1);
b2 = b2 - alpha * (Deltab2 * 1/m);
b1 = b1 - alpha * (Deltab1 * 1/m);
```

As a final step, a for loop is added to complete the algorithm. Additionally, the update term is included directly in the update function.

```
%The data set
x1 = [0; 0; 1; 1];
x2 = [0; 1; 0; 1];
y =  [0; 0; 0; 1];

%Initialisation weights
W1 = stdnormal_rnd(2,3);
W2 = stdnormal_rnd(3,1);
b1 = stdnormal_rnd(1,3);
b2 = stdnormal_rnd(1,1);

%Initialisation variables
alpha = 1;
labda = 0.01;
max_iterations = 5000;
m = rows(x1);

%sigmoid function
function s = f(z)
  s = 1./(1 + exp(-z));
end


for i = 1:maxi

    %Forward propagation
    %Layer 1 (input)
    a1 = [exx1, exx2];
```

```
29
30      %Layer 2
31      z2 = a1 * W1 + b1;
32      a2 = f(z2);
33
34      %Layer 3 (output)
35      z3 = a2 * W2 + b2;
36      h = f(z3);
37      a3 = h;
38
39      %Backward propagation
40      % Delta values
41      delta3 = -(exy - h)   .* (a3 .* (1 - a3));
42      delta2 = (delta3 * W2') .* (a2 .* (1 - a2));
43
44      W2 = W2 - alpha * (a2' * delta3 * 1/m); % + lambda * W2);
45      W1 = W1 - alpha * (a1' * delta2 * 1/m); % + lambda * W1);
46      b2 = b2 - alpha * (sum(delta3) * 1/m);
47      b1 = b1 - alpha * (sum(delta2) * 1/m);
48
49  end
```

5 Evolving ANNs and self-learning AI

5.1 TWEANNs and the NEAT Model

5.1.1 Introduction and Overview

Neural networks of a fixed topology, as described in the previous chapter, have some limitations. They require a human supervisor to carefully choose the most suitable topology, which is often a difficult or impossible choice. For very complex problems, reasoning about the right topology (one which can find complex relations while being as small and thus as optimized as possible) is practically impossible.

In such a case, the subdomain of neuro-evolution may be of use. In neuro-evolution, the concept of neural networks is combined with the knowledge of biological evolution to produce self-improving neural networks, that learn through a computational version of natural selection. Neuro-evolutionary algorithms use a population of networks and mutate them according to certain rules. Afterwards, they run the networks on the problem and see which network has the highest *fitness*, defined by a *fitness function*, on the basis of which networks get selected to cross over with other networks or mutate to go on to the next generation.

A common distinction between different forms of neuroevolution is whether they merely modify the weights of a neural network (conventional neuro-evolution) or whether they modify the topology of the network along with evolving the weights (TWEANNs). These *Topology and Weight Evolving Artificial Neural Networks* learn in a completely different manner from typical neural networks described in chapter four. Instead of tweaking the weights using a process such as backpropagation, TWEANNs generate a population of networks and mutate the topology and weights randomly. After mutating this population, a fitness value is computed for each network as normal. This cycle repeats until a solution is found.

Research shows, however, that TWEANNs are less efficient than typical artificial neural networks in most scenarios, and often fail to find the necessary structure to solve the problem. An algorithm called NEAT, or *NeuroEvolution of Augmenting Topologies*, has been shown to be able to solve these issues, building upon the concept of TWEANNs and improving its efficacy. NEAT simulates nature to an even higher degree, and has close conceptual ties with actual biological evolution. By encoding networks in genomes and making use of concepts such as speciation, historical markings and crossover, the NEAT algorithm can arrive much faster at effective networks on simple control tasks than other contemporary neuroevolutionary techniques and reinforcement learning methods.[18] Furthermore, these techniques are essential to evolving an effective algorithm for playing a video games.

5.1.2 NEAT's Model for Neuro-evolution

The model of neuro-evolution used by NEAT is based on and inspired by the many systems conceived over the last two decades that have implemented both weight and topology evolution. The NEAT model can be separated into a number of different features, outlined below.

5.1.2.1 Genetic Encoding

TWEANNs have to encode the genetic information that can be used to construct a neural network in some way, and can be divided into two groups: TWEANNs that use a direct

[18] Stanley and Miikkulainen (2002).

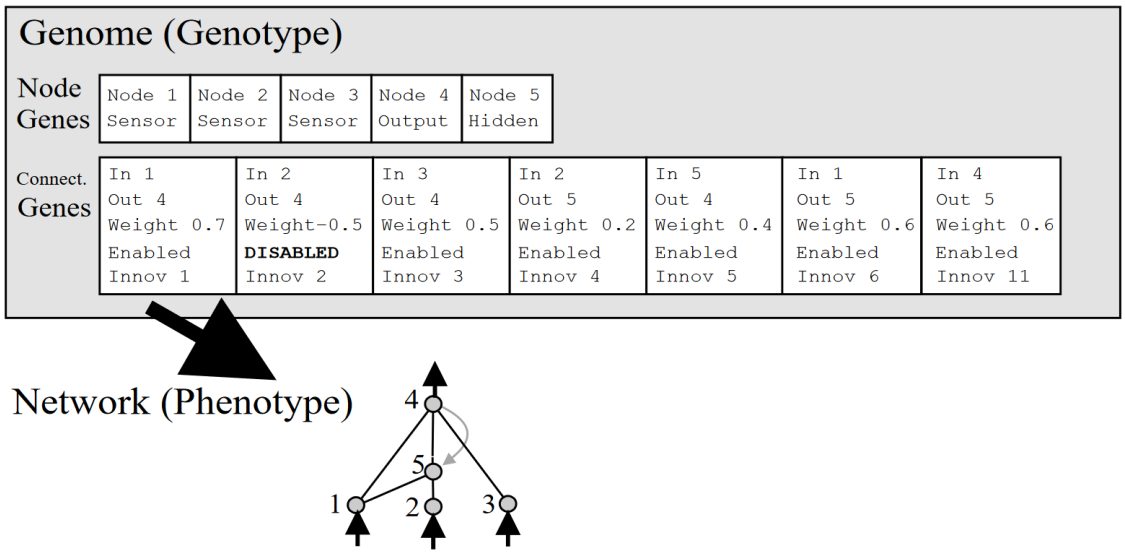

Figure 13: The way genotypes are defined in NEAT and translated into a phenotype.

encoding scheme and TWEANNs that use an indirect encoding scheme.

Direct encoding schemes specify the connections and nodes that will appear in the phenotype directly, whereas indirect encoding schemes merely specify rules for constructing a phenotype without directly specifying the connections and nodes - an example of such a rule could be rules for growing genomes through an equivalent of cell division. This can be more compact because the connections and nodes do not all have to be specified. However, they also require "more detailed knowledge of genetic and neural mechanisms.",[19] and when improperly used can lead to a biased search which finds a suboptimal set of topologies. Therefore, NEAT uses a direct encoding scheme.

Direct encoding schemes can also be differentiated into two sets: schemes based on the traditional binary encoding of genetic algorithms (GAs), such as the scheme used in *Structured Genetic Algorithm* (sGA),[20] where a bit string represents the genome of a network, and graph-based schemes, which use encodings that represent the graph structure of a neural network more explicitly, such as the scheme used in the *Parallel Distributed Genetic Programming* system.[21] NEAT uses a version of the traditional binary encoding where the genomes are defined as a linear list of nodes and a linear list of connections between them, as is displayed in figure 13. These lists are not of a fixed size, and allow for a technically limitless number of permutations.

5.1.2.2 Mutations

As discussed in the preceding sections, NEAT has the ability both to mutate weights and topology. Connection weights mutate similarly to most other NE systems - connection weights are either perturbed or reassigned from a normal distribution. NEAT has two types of topological mutations: the *add node* mutation and the *add connection* mutation.

The *add node* mutation splits an existing connection in two, and creates a new node. The old connection is disabled, and two connections are added - one from the origin of

[19] Braun and Weisbrod (1993).
[20] Dasgupta and McGregor (1992).
[21] Pujol and Poli (1998).

the new connection (henceforth *source*) to the new node, and one from the new node to the destination of the old connection (henceforth *sink*). The connection between the old source and the new node is given a connection weight of one, and the connection between the new node and the old sink is given the weight of the old connection. This minimizes the disturbance of the network due to the mutation, while still giving the network the ability to evolve new behaviours using the new node.

Over multiple generations, these mutations amount to a gradual increase in genome size, and because the mutations are based on chance, genomes of varying sizes will result. Crossover must be able to recombine these different networks, and the next section will explain how NEAT addresses this problem.

5.1.2.3 Historical Markings

One of the largest problems in neuro-evolution is the *Competing Conventions Problem* (CC). The essence of this problem is the fact that there are multiple ways of expressing the same solution to a problem with a neural network. When these ways do not have the same genetic encoding, crossover is likely to produce damaged offspring (see figure 14).

This is a fundamental problem with representing different structures and trying to apply crossover—their representations may not match up.

This problem is not new, however, and in this case the NEAT authors decided to look inspect how Mother Nature solved this problem. In nature, genomes also don't have a fixed length, but are still able to cross over successfully, crossing the right genes with each other. Nature solves this using a process called synapsis, which matches up homologous genes between two genomes before crossover. In neuro-evolution, ascertaining the homology of genes using structural analysis proves not to be easy - hence the competing conventions problem - so NEAT tries to solve CC by using historical markings. Genes are marked with an innovation number (see figure 13), which indicates its historical origin. When crossing over, genes with the same innovation number are lined up, and these are called *matching* genes. Genes that do not match can be *disjoint* or *excess*. Genes are defined as *disjoint* if they occur within the other parent's range of innovation numbers, *excess* when they do not (figure 15). Matching genes are inherited randomly from either parent, while disjoint and excess genes are inherited from the parent genome with the highest fitness.

Historical markings give NEAT the capability to perform crossover while avoiding the competing conventions problem. Genomes differing in size and topology stay compatible throughout evolution, and this methodology allows NEAT to complexify structure while maintaining compatibility. Smaller species, however, optimize more quickly than larger species, and mutating a network usually initially decreases the fitness of a network. To keep these newly augmented networks from disappearing from the population, innovation can be protected by using speciation, as explained in the next section.

5.1.2.4 Speciation

NEAT subdivides the population of genomes into species to protect their topological innovations using their topological similarity. This topological similarity can be computed by using the historical markings described in the section above. The distance δ between two networks can be computed using the following function, where E denotes the number of excess genes, D the number of disjoint genes, and $\overline{W}$ the average weight difference of matching genes:

$$\delta = \frac{c_1 \cdot D}{N} + \frac{c_2 \cdot E}{N} + c_3 \cdot \overline{W}$$

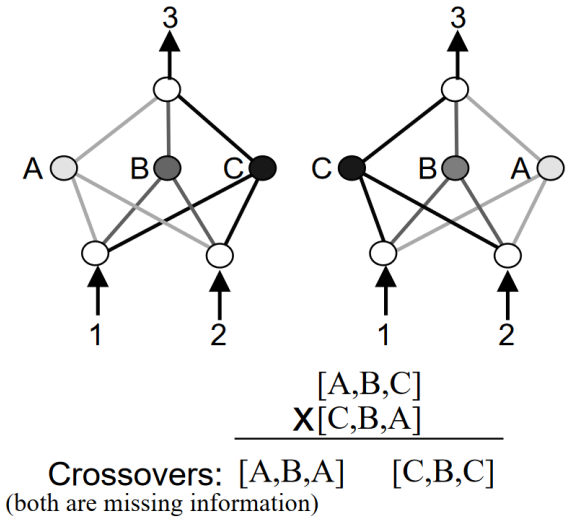

Figure 14: The competing conventions problem. Both genomes lose information when crossed over, and these are only two of the possible six representations of the same network.

The coefficients c_1, c_2 and c_3 adjust the importance of the three factors, and the factor N, the number of genes in the larger genome, normalizes for genome size.

Genomes are put into species one at a time by checking whether its distance to a randomly chosen member of the species is less than the compatibility threshold δ_t. If no species exists that satisfies the above condition, the genome is placed into a new species.

To determine the amount of offspring for a given species, NEAT uses *explicit fitness sharing*, where organisms in the same species share the fitness of their niche. This means a species cannot afford to become too big, even if many of its organisms perform well, thereby promoting topological diversity. The adjusted fitness f'_i for an organism i is computed according to the distance from every organism j in the population:

$$f'_i = \frac{f_i}{\sum_{i=j}^{n} sh(\delta(i,j))}$$

In the above equation, sh is the sharing function, which is set to 0 if $\delta(i,j) > \delta_t$ and 1 otherwise, reducing $\sum_{i=j}^{n} sh(\delta(i,j))$ to the number of organisms in the same species as i. The offspring for a given species is assigned proportionally to the sum of its members' adjusted fitnesses. This offspring consists of offspring through crossover, mutation, and interspecies crossover, the rates of which are defined in a separate configuration file.

The final goal is to perform the search for a solution as efficiently as possible, which is achieved through gradual complexification, described in the next section.

5.1.2.5 Initial Population Choice

TWEANNs usually start with an initial random population, in order to immediately introduce diversity. NEAT biases the search towards minimally dimensional spaces by starting with a population without any hidden nodes and introducing structure incrementally. Minimizing dimensionality in this way makes NEAT more performant.

5.2 The Application of NN based AI to Video Games

NEAT has been shown to be a capable learning method in many different problems NEAT, and seems to be excellent for solving our research problem. Any learning algorithm,

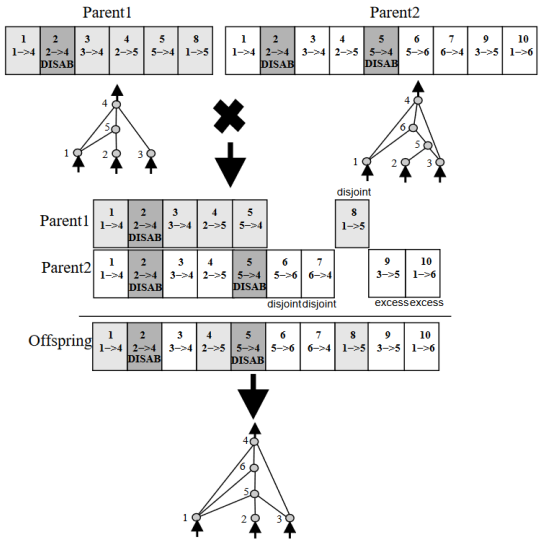

Figure 15: Genome crossover in NEAT.

especially artificial neural networks, is likely to lead to interesting results when applied to video games, in term to what it learns to perform and how it gets there. Furthermore, it can provide insights in multiple scientific fields, including machine learning, artificial intelligence, evolutionary ecology and neuroscience. But before NEAT can be applied and the experiment be initiated, some considerations are necessary, concerning a more accurate definition and description of the learning process as applied to video games, as well as a brief commentary on the selected game, Super Mario Bros. 3.

5.2.1 Games as Mathematical Problems

For NEAT, video games are a difficult challenge for numerous reasons. Because of the nature of video games, the algorithm, instead of solving only one problem at the time, often has to solve multiple in a row or even at once, while using a single topology. These problems rarely have a similar solution, and scenarios which may be similar with regards to inputs can require entirely different outputs. In addition, a tremendous amount of inputs nodes (the data from the game to the NEAT algorithm) is required in order to accurately represent the screen. Often, a topology will have to decide its outputs basis on multiple inputs in order to solve a problem and find useful correlations between actions.

Video games often present a challenge to the (human) player, and force them to improve in some manner, be it motoric proficiency, memory, critical thinking or reflexes. Playing a game requires logic from the player, and certain actions will lead to certain effects. If, for example, a player chooses to press the A button, a character might jump. Successive presses of the A button will always lead to the character jumping. This cause and effect system is a critical aspect of video games for learning algorithms to be able to improve. Another important aspect is that getting through a game often requires logic. Essentially, playing a game is nothing but problem solving. This too is important for NEAT to be able to improve, since the algorithm will have tremendous difficulty establishing useful connections in the topology otherwise.

From this can be concluded that a level in a video game can be considered a mathematical problem, in the sense that all scenarios can be broken down in simple formulas. For example, in the following situation, the enemy is observed (A), after which the decision is made to evade the enemy and jump (B). The outcome is that the enemy is avoided and

Mario survives (C).

Figure 16: Different stages of solving a problem. From left to right: event A, the enemy is detected and an input is given; event B, the decision is made and the outputs are computed; event C: the effect of the decision takes place.

This scenario can be represented in a formula as follows:

$$A \Rightarrow B \qquad B \rightarrow C$$

In this formula, the equals sign symbolizes a decision, and the arrow symbolizes the effect of the decision on the game. With use of an appropriate fitness calculation, the practicality of the effect (C) of the decision can be determined, with which the decision can be either justified or condemned. In a game, no matter how complex, there will always be some sort of information send to the player, followed by a decision, and lastly the effect of that decision. As a more complex example, a scenario with multiple inputs (A, B and C) and outputs (D and E) can be described as follows.

$$A + B + C \Rightarrow D + E \qquad D + E \rightarrow F$$

Converting a whole level of a certain game to this format is not practical however, and no such breakdown is provided in this paper. Nevertheless, it is useful and important to realize it can be done, as it, using the previously described necessities, verifies NEAT's ability to successfully learn to play video games.

5.2.2 The Way of Learning and Possible Limitations

There is an important distinction in how a human and an algorithm perceive the available data. When the player is a human brain, the problem, especially in a simple introductory level, can often be solved quickly. The data from the display of the game can be easily scanned for useful information with use of sensory neurons in the eyes - similar to input nodes in NEAT - and thereafter processed in the brain - the hidden nodes. These signals continue to motor neurons - the output nodes - in accordance with the topology of the brain. Within the complex human brain however, many intricate systems have evolved. When the player sees an obstacle in the way, for example, they, by using logical thinking and memory, will quickly understand to try and jump over it. An artificial neural network, especially one with a relatively small topology, does not have this ability. All outputs are derived directly from the inputs and the structure of the network.

The following figure demonstrates this. At the top of the screen, a visualization of a simple genome's network is visible. There is a connection from the ground below and in front of Mario to the button "Right". Because a hitbox is present on the source of the

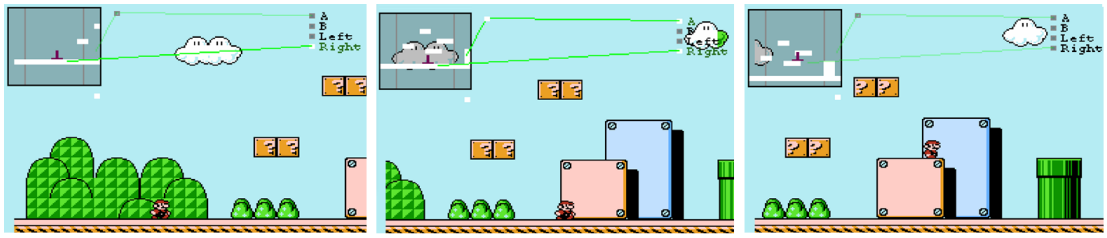

Figure 17: The effect different scenarios have on the output of the neural network. From left to right: scenario 1, Mario walks right; scenario 2, Mario jumps; scenario 3, Mario stands still.

connection, the connection is activated and Mario walks to the right (Scenario 1). After a little while, a hitbox appears at the source of another connection, which activates the button "A", and makes Mario jump (Scenario 2). But when Mario end up on the platform, and there is no hitbox at the connection's source any more, the "Right" button stops being pressed and Mario stops walking (Scenario 3). Where a human would try to push more buttons and see what happens, this genome only follows its own rules and, subsequently, it gets stuck.

In NEAT, all changes to the genomes are entirely random, caused by mutations to existing genomes. This means the algorithm does not directly learn from its mistakes as a human would. Instead, it randomly modifies genomes over time to either positive or negative effect, and gives the innovations that improve the fitness of a genome a higher chance to reproduce the next generation. In theory, all changes made to the genome need to be justified, as a disadvantageous mutation would lower the fitness score .[22] In our experiment, this is unlikely to happen, since connections to one of the high amount of inputs often have little or no impact on the fitness score.

Due to the random nature of the learning process, forming a detailed and accurate hypothesis on the results is difficult. If active for a long enough period of time, a correctly implemented NEAT algorithm should be able to find a solution to finish the level. This process, due to the difficulty of the problem, can take a very long time. As an estimate, the process might take up to 24 hours.

The critical ingenuity NEAT offers over many other learning methods, is the ability to effectively escape local optima, and over time, arrive at the global optimum. A typical level of any game will consist of many local optima, each a new challenge the learning algorithm will need to overcome. For instance, after NEAT has evolved to walk to the right, the first local optima will be located just before the first enemy. At this point, all genomes will probably die if they have not evolved a correct connection with an output to jump. The next probable local optimum is the first wall, for which the genome needs to be able to jump high enough to be able to clear. These are only basic examples of local optima, but illustrate the concept well. The algorithm will need to overcome these challenges, including more difficult obstacles, in order to eventually reach the end of the level.

Because of the high number of inputs necessary to correctly portrait the screen of which many rarely contain a value, a lot of connections will have no function. To counteract this problem, a high mutation rate is necessary. As a side effect, this will generate relatively

[22]Stanley and Miikkulainen (2002).

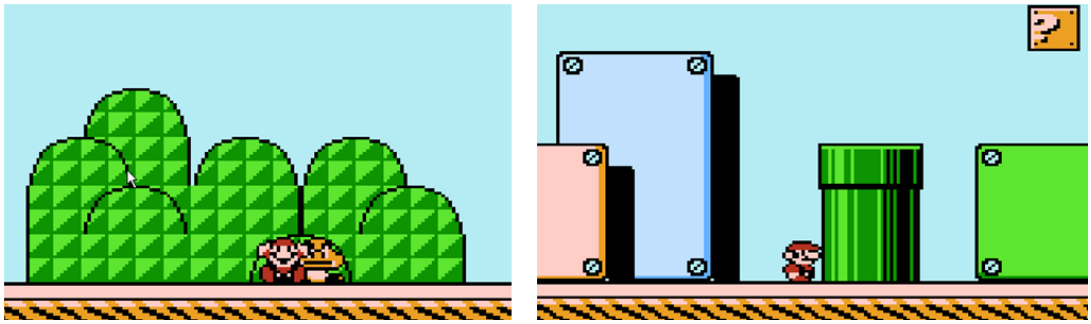

Figure 18: Different local optima in the first level of Super Mario Bros. 3. From left to right: scenario 1, the first enemy; scenario 2, the first wall (a pipe).

large genomes.

5.2.3 The Selected Video Game

With this information in mind, a fitting game needs to be selected. This game should have a logical level structure and design and clear cause and effect mechanics. Additionally, the game should not be too difficult. To further simplify the learning process, the game should have as little inputs and outputs as possible. Lastly, the fitness should be easily calculable.

The 1987 game Super Mario Bros. 3 for the Nintendo Entertainment System (NES) was decided upon, as it supports all stated requirements. The platforming genre seemed to be the best choice, since the player often only needs to move to the right, making the x coordinate qualified for use in the fitness value. Furthermore, the usage of an old game reduces the amount of inputs and outputs.

5.2.4 NEAT and Video Games as a Model for Evolution

The NEAT algorithm illustrates the development of early brains, as they emerged in the ancestral life forms of animals over 500 million years ago een of andere bron. And although current computation speeds are a long way from achieving this, the algorithm could in theory simulate a legitimate brain. SMB3 can be seen as an extremely simplified version of earth billions of years ago, when organisms, or rather their genes, possessing optimal network topology would have a higher fitness and subsequently gain an advantage for reproduction and survival in the process of natural selection. Each instance of Mario represents a single organism, and how far they have come in the level their fitness. The continuous improvement of the population can therefore be seen as a greatly simplified reconstruction of evolution and the evolution of brains. In the real world however, there the level is unending, and genomes can be improved infinitely. Even with the extreme low mutation rates, life on earth was able to evolve exceptionally complex brains, with totals of billions of neurons.

5.3 Implementation in the Programming Language Python

PyNEAT, our Python implementation of the NEAT algorithm, is subdivided into the ten files shown in the figure below. These files, and the code inside them, shall be described in the following few sections.

```
PyNEAT
├─ neat
│  ├─ genome.py - Contains genotypic classes
│  ├─ network.py - Contains phenotypic classes
│  ├─ species.py - Contains the species specification
│  ├─ pool.py - Contains the network pool
│  ├─ loader.py - Contains a loader for saved data
│  ├─ configuration.py - Contains a configuration loader
│  └─ nodetype.py - Contains an enumeration of possible node types
├─ mario.py - Contains Lua ⇔ Python code & model data
├─ Mario.toml - Contains the NEAT configuration
└─ requirements.txt - Contains the dependencies (the TOML module)
```

5.3.1 The Genotype Implementation, `genome.py`

This file contains the classes that define the genotype of a neural network: `NodeGene`, `ConnectionGene` and `Genome`, but it starts off with a prelude full of imports. The definitions on lines 11, 12 and 14 define global initial values for the connection ID, node ID and genome ID. `GLOBAL_NODES` is initialised to 248 because this happens to be the first non-preinitialised node ID for our experiment.

```
1  import random
2  import math
3  import copy
4  from statistics import mean
5  from . import configuration
6  from collections import defaultdict
7  from .nodetype import NodeType
8  import itertools
9  from pprint import pprint
10
11 GLOBAL_INNOVATION = 1
12 GLOBAL_NODES = 248
13
14 GENOME_ID = 1
```

Subsequently, the activation function used in our experiment is defined.

```
1  def modSigmoid(x):
2      return (2 / (1 + math.exp(-4.9*x)))-1
```

Then, the first proper part of the genotype, the `NodeGene` class, is defined. Nodes have to have a `nodeId`, `nodeType` (either an input, bias, hidden or output node) and an `activationFunction` attached to them. These are set in the initialisation function. Additionally, Python *magic methods* are defined which make sure that nodes with the

same node ID are hashed and compared as equal, and define a textual representation for nodes.

```
class NodeGene():
    def __init__(self, nodeType, nodeId=None, activationFunction = modSigmoid):
        global GLOBAL_NODES
        global GLOBAL_INNOVATION
        self.nodeId = nodeId or GLOBAL_NODES
        if not nodeId:
            GLOBAL_NODES += 1
        self.nodeType = nodeType
        self.activationFunction = activationFunction

    def __eq__(self, other):
        try:
            return self.nodeId == other.nodeId
        except:
            return False

    def __ne__(self, other):
        return not self.__eq__(other)

    def __hash__(self):
        return hash(self.nodeId)

    def __repr__(self):
        if self.nodeType in (NodeType.SENSOR, NodeType.BIAS):
            return "Node(#%r, %r)" % (self.nodeId, self.nodeType)
        return "Node(#%r, %r, %s)" % (self.nodeId, self.nodeType,
            self.activationFunction.__name__)
```

Next, the `ConnectionGene` class is defined. Connections have a `source`, `sink`, `weight` and `connectionId`, and an `enabled` flag, which are defined in its respective initialisation function. The subsequent lines define magic methods akin to those mentioned for `NodeGene`.

```
class ConnectionGene():
    def __init__(self, source, sink, weight = None, connectionId = None, enabled
        = None):
        global GLOBAL_INNOVATION
        self.source = source
        self.sink = sink
        self.weight = weight or random.gauss(0.0, 1.0)
        self.connectionId = connectionId or GLOBAL_INNOVATION
        if not connectionId:
            GLOBAL_INNOVATION += 1
        self.enabled = enabled or True

    def __repr__(self):
        return "Connection(#%d, Node %r => Node %r, Weight %f, %s)" \
            % (self.connectionId, self.source.nodeId, self.sink.nodeId,
                self.weight, "Enabled" if self.enabled else "Disabled")

    def __eq__(self, other):
        return self.connectionId == other.connectionId

```

```
    def __ne__(self, other):
        return not self.__eq__(other)

    def __hash__(self):
        return hash(self.connectionId)
```

Now, we have gotten to the most important class in this file, the `Genome` class. This class defines the way the lists of nodes and connections are stored, as well as the way they are mutated, the way the species difference is computed and the way genomes are crossed over. First, let us look at the data definitions.

The configuration is fetched using the `configuration.getGlobalConfig()` method described further on in this section, and the node and connection lists are represented using Python's hashmaps (also called key-value maps or dictionaries), mapping nodeId and connectionId to node and connection, respectively. The fitness and global rank are initialised to 0, and will be updated by other methods and classes.

```
class Genome():
    def __init__(self, nodes=None, connections=None):
        global GENOME_ID
        self.config = configuration.getGlobalConfig()
        self.nodes = {node.nodeId:node for node in (nodes or [])}
        self.connections = {connection.connectionId:connection for connection in
            (connections or [])}
        self.fitness = 0
        self.id = GENOME_ID
        GENOME_ID += 1

        self.globalRank = 0
```

A few convenience methods are defined for easily adding nodes and connections.

```
    def addNode(self, nodeId, nodeType):
        node = NodeGene(NodeType(nodeType), nodeId)
        self.nodes[node.nodeId] = node

    def addConnection(self, source, sink, weight=None, connectionId=None,
        enabled=None):
        connection = ConnectionGene(source, sink, weight, connectionId, enabled)
        self.connections[connection.connectionId] = connection
```

We now get to the first of the mutation functions, `mutateNode`. This function performs the add node mutation, adding a new node between an existing connection. It chooses a random connection from the list, disables it, constructs a new hidden node and adds the appropriate connections.

```
    def mutateNode(self):
        if (not self.connections):
            return False

        node = NodeGene(NodeType.HIDDEN)
        # Add a node between an existing connection
        conn = random.choice(list(self.connections.values()))
        conn.enabled = False

        new_connections = (ConnectionGene(conn.source, node, weight = 1),
```

```
            ConnectionGene(node, conn.sink, weight=conn.weight))

        for connection in new_connections:
            self.connections[connection.connectionId] = connection

        self.nodes[node.nodeId] = node
```

The second mutation function, `mutateConnection`, adds a connection between two random nodes. It defines a list of nodes that are eligible to be sources and a list of nodes that are eligible to be sinks (input nodes and bias nodes are not eligible to be sinks, and if the `bias` argument is passed, only bias nodes can be sources), whereafter it creates the Cartesian product of these two lists using the built-in `itertools` module. (The Cartesian product $A \times B$ is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$). It chooses a random combination from this product where $a \neq b$ and where the connection does not already exist, and adds this connection to the list of connection genes.

```
    def mutateConnection(self, bias=False):
        if not bias:
            source_eligible = [node for node in self.nodes.values() if
                node.nodeType in (NodeType.SENSOR, NodeType.BIAS,
                NodeType.HIDDEN, NodeType.OUTPUT)]
        else:
            source_eligible = [node for node in self.nodes.values() if
                node.nodeType == NodeType.BIAS]
        sink_eligible = [node for node in self.nodes.values() if node.nodeType in
            (NodeType.HIDDEN, NodeType.OUTPUT)]

        existing_connections = {(conn.source, conn.sink):conn for conn in
            self.connections.values()}

        combinations = [combination for combination in
            itertools.product(source_eligible, sink_eligible) if combination[0]
            != combination[1] and combination not in existing_connections]

        if (not combinations):
            return False
        chosen = random.choice(combinations)

        self.addConnection(*chosen)
```

Some more convenience functions for copying the structure and fetching a specific node or connection are defined.

```
    def node(self, nodeId):
        return self.nodes.get(nodeId, False)

    def connection(self, connectionId):
        return self.connections.get(connectionId, False)

    def copy(self):
        return copy.deepcopy(self)
```

The `mutateWeights` function is defined, which either perturbs or resets connection weights.

```
    def mutateWeights(self):
        for connection in self.connections.values():
            if (random.random() < self.config["rates"]["perturbation"]):
                connection.weight += (random.gauss(0.0, 0.5))
            else:
                connection.weight = random.gauss(0.0, 1.0)
```

The `mutate` function is defined, which uses the $[0.0, 1.0)$ output from `random.random()` as a chance comparison to choose whether to perform the mutations provided by the functions above.

```
    def mutate(self):
        if (random.random() < self.config["rates"]["mutation"]):
            self.mutateWeights()

        n = self.config["rates"]["newnode"]
        while (random.random() < n):
            self.mutateNode()
            n -= 1

        n = self.config["rates"]["newlink"]
        while (random.random() < n):
            self.mutateConnection()
            n -= 1

        if (random.random() < self.config["rates"]["biasmut"]):
            self.mutateConnection(True)
```

The function used to determine whether two genomes belong to the same species, `sameSpecies`, is defined. Firstly, it fetches the constants c_1, c_2 and c_3 from the configuration. Then, it computes various values used in the equation $\delta = \frac{c_1 \cdot D}{N} + \frac{c_2 \cdot E}{N} + c_3 \cdot \overline{W}$. The excess threshold is determined by taking the minimum largest connection ID from the species, and is used to determine whether genes are disjoint or excess. The matching genes are determined using a list comprehension that checks whether a connection with the same ID exists in the other genome's connection list, in which case it adds both to the matching gene set. The computation of the other values is fairly self-explanatory. At the end of the function, a boolean value representing its membership in the species is returned.

```
    def sameSpecies(self, other):
        c_1 = self.config["compatibility"]["excess"]
        c_2 = self.config["compatibility"]["disjoint"]
        c_3 = self.config["compatibility"]["weights"]

        # d = c_1*E/N + c_2*D/N + c_3 * W
        nGenes = (len(self.connections), len(other.connections))

        N = 1

        # Gene connectionIds below this threshold are disjoint. Uniques higher
            than this threshold are excess.
        if (len(self.connections) == 0) or (len(other.connections) == 0):
            excessThresh = 0
        else:
            excessThresh = min(max(cId for cId in self.connections.keys()),
                max(cId for cId in other.connections.keys()))
```

```
16
17          matchingGenes = {gene: other.connection(gene.connectionId) for gene in
                self.connections.values() if other.connection(gene.connectionId)}
18
19          weightDiff = [abs(a.weight - b.weight) for a,b in matchingGenes.items()]
20          if (weightDiff):
21              avgweightDiff = mean(weightDiff)
22          else:
23              avgweightDiff = 0
24
25          otherGenes   = set(self.connections.values()) ^
                set(other.connections.values())
26
27          disjoints = {gene for gene in otherGenes if gene.connectionId >
                excessThresh}
28          excesses = otherGenes - disjoints
29
30          return ((c_1 * len(excesses) + c_2 * len(disjoints)) / N + c_3 *
                avgweightDiff) < self.config["speciation"]["species_difference"]
```

The `crossover` function, the final piece of the genotype puzzle, is defined last. It computes variables also used in the `sameSpecies` function, and so requires little explanation. The matching and other genes are selected, and are chosen based on the rules described in section 5.1.2.3. At the end, the resulting genome is returned.

```
1       def crossover(self, other):
2           global GENOME_ID
3           offspring = Genome()
4
5           if (len(self.connections) == 0) or (len(other.connections) == 0):
6               excessThresh = 0
7           else:
8               excessThresh = min(max(cId for cId in self.connections.keys()),
                    max(cId for cId in other.connections.keys()))
9
10          matchingGenes = [(gene, other.connection(gene.connectionId)) for gene in
                self.connections.values() if other.connection(gene.connectionId)]
11
12          otherGenes = set(self.connections.values()) ^
                set(other.connections.values())
13
14          matchingChoices = [random.choice(x) for x in matchingGenes]
15
16          if (self.fitness > other.fitness):
17              otherChoices = [choice for choice in otherGenes if choice in
                    self.connections.values()]
18          elif (other.fitness > self.fitness):
19              otherChoices = [choice for choice in otherGenes if choice in
                    other.connections.values()]
20          else:
21              otherChoices = list(otherGenes)
22
23          offspring.connections = {i.connectionId:i for i in (matchingChoices +
                otherChoices)}
24
```

```
        for connection in offspring.connections.values():
            if connection.enabled == False:
                connection.enabled == False if random.random() <
                    self.config["rates"]["disabled_inherit"] else True

        offspring.nodes = {i.nodeId:i for i in (set(self.nodes.values()) |
            set(other.nodes.values()))}

        GENOME_ID += 1
        offspring.id = GENOME_ID

        return offspring
```

5.3.2 The Phenotype Implementation, `network.py`

This file contains the classes that define the phenotype of a neural network: `Node` and `Network`. We can once again start with the imports:

```
from collections import defaultdict, namedtuple, OrderedDict
from .nodetype import NodeType
from pprint import pprint
import random
import sys
```

No definitions here, so we will continue to describe the `Node` class. In the initialisation function, the node type, ID and activation function are fetched from the passed node gene. The weights leading into this node are fetched from the passed connections, as are the node's dependencies.

```
class Node():
    def __init__(self, nodeGene, connections):
        self.nodeType = nodeGene.nodeType
        self.nodeId = nodeGene.nodeId
        self.weights = {conn.source.nodeId:conn.weight for conn in connections
                        if conn.enabled and conn.sink == self}
        self.activationFunction = nodeGene.activationFunction
        self.connections = connections
        self.dependencies = {conn.source.nodeId for conn in connections
                             if conn.sink == self and conn.enabled}
```

More magic methods describing equality are defined below, just like in the `ConnectionGene` and `NodeGene` classes.

```
    def __eq__(self, other):
        if (self.nodeId == other.nodeId) and (self.nodeType == other.nodeType):
            return True
        return False

    def __ne__(self, other):
        return not self.__eq__(other)

    def __hash__(self):
        return hash(self.nodeId)

    def __repr__(self):
```

```
        if self.nodeType in (NodeType.SENSOR, NodeType.BIAS):
            return "Node(#%r, %r, %r)" % (self.nodeId, self.nodeType,
                    self.weights)
        return "Node(#%r, %r, %s, %r)" % (self.nodeId, self.nodeType,
                self.activationFunction.__name__, self.weights)
```

Finally, the most important function, `__call__`, is defined. This function computes the output of the node based on the passed inputs. First, all of the inputs that the node depends on (from `self.dependencies`) are summed and multiplied by their weights. If the node is a bias node, the input is set to 1. If the node is an input node, the input is fetched directly, because input nodes have no dependencies or weights. If the node is a hidden or output node, the activation function is applied to the input. Finally, the output is added to the necessary lists and returned.

```
    def __call__(self, output, inputs=None, final_out=None):
        if (self.nodeType == NodeType.SENSOR):
            input = inputs.get(self.nodeId)
        else:
            input = 0
            for dependency in self.dependencies:
                val = inputs.get(dependency, 0.0)
                cW = self.weights.get(dependency, 0.0)
                if (val == None):
                    inputs[dependency] = 0.0
                    val = 0.0
                input += val * cW

        if (self.nodeType == NodeType.BIAS):
            input = 1.0

        if (self.nodeType in (NodeType.HIDDEN, NodeType.OUTPUT)):
            input = self.activationFunction(input)

        output[self.nodeId] = input

        if self.nodeType == NodeType.OUTPUT:
            final_out[self.nodeId] = input

        return output
```

The `Network` class is defined. This is the phenotype version of the `Genome` class defined in `genome.py`. In the initialisation function, the nodes are created from the passed genome, and an empty values hashmap is created which stores the output of the network for use with recurrent connections. Additionally, a convenience function is defined to select a node.

```
class Network():
    def __init__(self, genome):
        self.nodes = OrderedDict(sorted(self.createNodes(genome), key=lambda
            x:x[0]))
        self.values = defaultdict(float)

    def node(self, nodeId):
        return self.nodes[nodeId]
```

The `createNodes` function is a generator that yields a list of nodes from a given genome. For every node gene in a genome, it selects its corresponding connections, constructs a `Node` instance and yields a pair of `(nodeId, node)`.

```
    def createNodes(self, genome):
        for nGene in genome.nodes.values():
            connections = [c for c in genome.connections.values()
                   if c.sink == nGene or c.source == nGene]
            node = Node(nGene, list(connections))
            yield (node.nodeId, node)
```

Next, the function to run the network is defined. First, the input and bias nodes are selected and run. Then, the rest of the functions are run. As you may have spotted, the node hashmap has been defined as an `OrderedDict` in the initialisation function, and the nodes already have an ordering. This is used to our advantage when running the pool, because we can run the pool without caring about dependencies for the first frame (making recurrent connections possible), then save the values in `self.values`. These values are used in the computation of the outputs for the other frames, and in spite of the incorrect output for the first frame, still accurately runs the neural network. This is similar to code used in other implementations.[23] The final outputs are collected in `output_data`, which is returned, sorted on nodeId, as the output of the neural network.

```
    def runWith(self, inputdata=None):
        data_pool = defaultdict(float)
        output_data = defaultdict(float)

        # First, run the inputs and bias nodes
        inputs = [node for node in self.nodes.values() if node.nodeType ==
            NodeType.SENSOR or node.nodeType == NodeType.BIAS]
        inputs.sort(key=lambda node:node.nodeId)

        for ip in inputs:
            ip(self.values, dict(zip([node.nodeId for node in inputs],
                inputdata)), output_data)


        # Then the other nodes
        other_nodes = [node for node in self.nodes.values() if node.nodeType in
            (NodeType.OUTPUT, NodeType.HIDDEN)]

        for node in other_nodes:
            node(self.values, self.values, output_data)

        self.values.update(output_data)
        output_data = list(output_data.items())
        output_data.sort(key=lambda node: node[0])

        return [x[1] for x in output_data]
```

5.3.3 The Species class, `species.py`

This file contains a `Species` class, which contains a list of genomes, a species ID, and some fitness measurements (`topFitness`, `averageFitness`, `staleness`), and functions to

[23] CodeReclaimers (2008-2017).

compute average fitness and sort the genomes. The reason the genomes are sorted on fitness reverse is to make sure `genomes[0]` is the species with highest fitness, whereas a normal sort would sort the genomes lowest fitness to highest.

```
import random
from . import configuration

SPECIES_ID = 1

class Species():
    def __init__(self):
        global SPECIES_ID
        self.config = configuration.getGlobalConfig()
        self.id = SPECIES_ID
        self.genomes = []
        self.topFitness = 0.0
        self.averageFitness = 0.0
        self.staleness = 0
        SPECIES_ID += 1

    def calculateAverageFitness(self):
        total = 0

        for genome in self.genomes:
            total = total + genome.globalRank

        self.averageFitness = total / len(self.genomes)

    def sortGenomes(self):
        self.genomes.sort(key=lambda genome: genome.fitness, reverse=True)
```

5.3.4 The Network Pool, `pool.py`

The network pool, combined with the genotypic and phenotypic classes, forms the meat of the NEAT algorithm. A population of networks described by genomes is divided into multiple species, run using the `Network` class, sorted, and using the algorithms described below, turned into a new generation, and run again, until a genome is generated that is optimized for the problem that is being solved.

The initialisation function starts by initialisating data that applies to the pool, such as the `fitnessFunction`, `maxFitness`, et cetera. If an initial genome is passed, it is used to create an initial generation. If an initial genome is not passed, such as during loading of a previously saved generation as discussed below, the generation has to be created separately. Additionally, a runtime directory is created wherein the genomes are saved.

```
import pprint
import random
import sys
import os
import pdb
import datetime
import json
import traceback
from .species import Species
from .network import Network
from . import configuration
```

```
12
13 class NetworkPool():
14     def __init__(self, initialGenome=None, fitnessFunction=None):
15         print("Constructing pool.")
16         self.fitnessFn = fitnessFunction
17         self.config = configuration.getGlobalConfig()
18         self.species = []
19         if initialGenome:
20             for i in range(self.config["pool"]["population"]):
21                 print("Constructing genome %d" % i)
22                 copy = initialGenome.copy()
23                 copy.mutate()
24                 self.addToSpecies(copy)
25
26         self.maxFitness = 0.0
27         self.maxPrevFitness = 0.0
28         self.maxGenome = None
29         self.evaluatedGenomes = 0
30         self.generation = 1
31
32         self.rtdir = "run-" + datetime.datetime.utcnow().strftime("%Y%m%d-%H%M%S")
33         os.mkdir(self.rtdir)
```

The `addToSpecies` function provides the functionality of adding a genome to a species, which can be either existing or newly created, based on the algorithm described in 5.1.2.4.

```
1     def addToSpecies(self, genome, speciesId=None, staleness=None):
2         for species in self.species:
3             if species.genomes[0].sameSpecies(genome):
4                 species.genomes.append(genome)
5                 return
6
7         newSpecies = Species()
8         newSpecies.genomes.append(genome)
9         newSpecies.staleness = staleness or 0
10        self.species.append(newSpecies)
```

The `rankGlobally` and `totalAverageFitness` functions concern the ranking of genomes and the species they are a member of, which is used to proportionally assign the number of offspring created by species.

```
1     def rankGlobally(self):
2         globalList = [genome for species in self.species for genome in
              species.genomes]
3
4         globalList.sort(key=lambda x: x.fitness)
5
6         for n, genome in enumerate(globalList):
7             genome.globalRank = n
8
9     def totalAverageFitness(self):
10        total = 0
11
12        for species in self.species:
13            total = total + species.averageFitness
14
```

```
15        return total
```

Constructing a new generation is handled by the `nextGeneration` function. First, the species are culled, removing the least fit half of their genomes. Then, the stale species (species that have stagnated for more than `staleness` generations) are removed, whereafter the species are ranked globally. The average fitness is calculated, and with that, the proportional offspring calculated. If the offspring is less than one genome, the species is deleted by the function `removeWeakSpecies`. The species generate their offspring, whereafter they are culled to consist of their single most fit organism, which is then copied verbatim into the next generation. If the number of genomes in the total population is less than the population size, the gap is filled with extra offspring through mutating the best genomes. Subsequently, the generation counter is updated.

```
1     def nextGeneration(self):
2         self.cullSpecies(cutToOne=False)
3         self.removeStaleSpecies()
4         self.rankGlobally()
5         for species in self.species: species.calculateAverageFitness()
6         self.removeWeakSpecies()
7
8         self.totalFitness = self.totalAverageFitness()
9
10        children = []
11
12        for species in self.species:
13            toBreed = (self.config["pool"]["population"] *
                  species.averageFitness) // self.totalFitness
14            toBreed -= 1
15
16            children.append(self.generateOffspring(species))
17
18        self.cullSpecies(cutToOne=True)
19
20        while (len(children) + len(self.species) <
              self.config["pool"]["population"]):
21            species = random.choice(self.species)
22            children.append(self.generateOffspring(species))
23
24        for child in children: self.addToSpecies(child)
25
26        self.generation += 1
```

The `generateOffspring` function constructs a new child from a species based on the crossover and interspecies mating rate described in the configuration file.

```
1     def generateOffspring(self, species):
2         if random.random() < self.config["rates"]["crossover"]:
3             if (random.random() < self.config["rates"]["interspecies"]):
4                 species2 = random.choice(self.species)
5                 genome = random.choice(species2.genomes)
6                 child = genome.crossover(random.choice(species.genomes))
7                 child.mutate()
8                 return child
9             elif len(species.genomes) >= 2:
10                g1, g2 = random.sample(species.genomes, 2)
```

```
            child = g1.crossover(g2)
            child.mutate()
            return child

    child = random.choice(species.genomes).copy()
    child.mutate()
    return child
```

The functions `cullSpecies`, `removeWeakSpecies` and `removeStaleSpecies` fulfill the functions described above in `nextGeneration`.

```
def cullSpecies(self, cutToOne = False):
    for species in self.species:
        species.sortGenomes()

        remaining = (len(species.genomes) // 2) + 1

        if (cutToOne):
            remaining = 1

        species.genomes = species.genomes[:remaining]

def removeWeakSpecies(self):
    survived = []

    self.totalFitness = self.totalAverageFitness()

    for species in self.species:
        toBreed = (self.config["pool"]["population"] *
            species.averageFitness) // self.totalFitness
        if toBreed >= 1:
            survived.append(species)
        else:
            print("Removing species %d (tB: %d, avF: %f, tF: %f)" %
                (species.id, toBreed, species.averageFitness,
                self.totalFitness))

    self.species = survived

def removeStaleSpecies(self):
    survived = []
    for species in self.species:
        species.sortGenomes()

        if (species.genomes[0].fitness > species.topFitness):
            species.topFitness = species.genomes[0].fitness
            species.staleness = 0
        else:
            species.staleness += 1

        if (species.staleness <
            self.config["speciation"]["stagnation_threshold"]) or
            (species.topFitness >= self.maxFitness):
            survived.append(species)
        else:
            print("Removing stale species %d." % (species.id))
```

```
41
42          self.species = survived
```

The genomes are saved to disk in the runtime directory initialised in the initialisation function, and the `saveGenome` function fulfils this task. The data format used is JSON (*JavaScript Object Notation*), which is a broadly used exchange format both for Python and in the general programming community.

```
1       def saveGenome(self, species, n, genome):
2           dataFile = open(os.path.join(self.rtdir, "gen-%d" % self.generation,
                "spc-%d-gnm-%d" % (species.id, n)), "w")
3           data = {"connections": sorted([(conn.connectionId, conn.source.nodeId,
                conn.sink.nodeId, conn.weight) for conn in genome.connections.values()
4                                   if conn.enabled], key=lambda x: x[0]),
5           "nodes": sorted([(node.nodeId, node.nodeType.value) for node in
                genome.nodes.values()]),
6           "maxFitness": self.maxFitness,
7           "generation": self.generation,
8           "genomeId": n,
9           "staleness": species.staleness,
10          "speciesId": species.id}
11
12          dataFile.write(json.dumps(data))
13          dataFile.close()
```

Finally, the `runOnce` function runs every genome in a single generation, running it on every input, updating its fitness and saving it. Additionally, this sets the maximum values (`maxFitness`, etc) for a specific generation. As you can see, there is a lot of debugging code in this function (from pretty-printing the species and their genome count to breaking into a debugger when it is impossible to get a correct fitness value), since it truly ties everything together.

```
1       def runOnce(self, inputsFn, sendGenome, sendOutput):
2           pprint.pprint([(sp.id, len(sp.genomes)) for sp in self.species])
3           print("Running generation %d" % (self.generation))
4           os.mkdir(os.path.join(self.rtdir, "gen-%d" % self.generation))
5           for species in self.species:
6               for n, genome in enumerate(species.genomes):
7                   genome.id = n
8                   print("Gen %d S %d G %d" % (self.generation, species.id, n+1))
9                   network = Network(genome)
10                  fitness = 0
11                  sendGenome(self, species, genome, n)
12                  nInputs = 0
13                  for input in inputsFn():
14                      nInputs += 1
15                      output = network.runWith(input)
16                      sendOutput(output)
17                      try:
18                          fitness = self.fitnessFn(input, output)
19                      except:
20                          traceback.print_exc()
21                          pprint.pprint(genome.nodes)
22                          pprint.pprint(genome.connections)
23                          pdb.set_trace()
```

```
                    sys.exit(1)
                genome.fitness = fitness

                self.saveGenome(species, n, genome)

                self.evaluatedGenomes += 1

        self.maxPrevFitness = self.maxFitness
        self.maxPrevGenome = self.maxGenome
        self.maxGenome = max(((genome,species) for species in self.species for
            genome in species.genomes), key=lambda g: g[0].fitness)
        self.maxFitness = self.maxGenome[0].fitness
        self.maxSpecies = self.maxGenome[1]
        self.maxGenome = self.maxGenome[0]
```

5.3.5 The Loader Class, `loader.py`

This file provides a loader for the files saved to disk by the NetworkPool described in the above section. It simply reconstructs the pool from the saved data.

```
# This file has proved instrumental a few times when we have had to reload
# generations from disk because of errors made both in the Python and Lua
# scripts, but was really fairly simple to write.

# in fact, it was done during a particularly boring chemistry lesson
# on a scrap piece of paper.

from .genome import NodeGene, ConnectionGene, Genome
from .pool import NetworkPool
from .nodetype import NodeType

import json, os

class Loader():
    def __init__(self, rundir, gen):
        self.gendir = os.path.join(rundir, "gen-%s" % gen)

    def getGenomes(self, fitnessFunction):
        pool = NetworkPool()

        for genome in os.listdir(self.gendir):
            data = json.load(open(os.path.join(self.gendir, genome)))
            g = Genome()

            for node in data["nodes"]:
                g.addNode(node[0], node[1])

            for connection in data["connections"]:
                g.addConnection(g.node(connection[1]), g.node(connection[2]),
                    connection[-1], connection[0])

            pool.maxFitness = data["maxFitness"]
            pool.generation = data["generation"]
            pool.addToSpecies(g, speciesId=data["speciesId"],
                staleness=data["staleness"])

```

```
35        return pool
```

5.3.6 `configuration.py` and `nodetype.py`

These are really more like auxiliary tiny files which have small convenience functions or classes that are used throughout the code. They are simple enough to understand without any further explanation.

```
import toml
config = None

def getGlobalConfig():
    return config

def loadConfig(filename):
    global config
    config = toml.load(filename)
    return config
```

```
import enum

class NodeType(enum.Enum):
   SENSOR = 1
   BIAS = 2
   HIDDEN = 3
   OUTPUT = 4
```

6 Experiment: The application of ANN based AI to SMB3

6.1 Data Allocation and RAM Map

6.1.1 Introduction and Overview

Before the NEAT algorithm can be applied to the learning problem, it needs to be given a usable set of inputs. For the learning process to be as fair as possible, the neural network needs to 'see' what a normal player would. This means everything displayed on screen must be to converted to a set of numerical data, which has been diminished in size and complexity. Conveniently, all the necessary data can be received directly from memory. For this, the freeware emulator BizHawk is used, as it has functionality to read RAM data, as well as support for lua scripting.

6.1.2 NES Memory Mapping

Super Mario Bros. 3 (SMB3) is a 1988 software title for the 8-bit console, the Nintendo Entertainment System (NES). Although this means the total amount of data is relatively small, the semi-arbitrary nature of the allocation of it still makes finding the right locations difficult and time consuming. For SMB3, however, there are some resources available online to simplify the process a bit ,[24] although no complete RAM map is currently available. This paper will present an incomplete RAM map as well, wherein all used data locations are given.

Data is stored within the memory in different memory locations, for specific use of a single variable, such as the location of an enemy, or the location of a tile sprite. The ROM (read-only memory) is fixed, and consists of all the game data. Because this does not (directly) correspond to what is displayed on screen, this won't be useful for the experiment. The RAM (random-access memory) contains the instructions for the display, and contain all the data necessary for defining the inputs. All locations have a size of a single byte and their location is defined in hexadecimal notation, for which the prefix `0x` is used. For example, data location number `1234` will be notated as `0x04D2`.

For early NES games, the maximum amount of memory that could be addressed was 32KB of program ROM, 8KB of 8x8 character graphics tiles and 2 KB of RAM.[25] In order to be able to increase the complexity and size of the games such as SMB3, later NES software developers designed creative solutions to bypass these limitations. Within the cartridge, a chip called a *mapper* was added, which provided access to extended memory. For SMB3, the mapper is known as MMC3 and changed a couple aspects of the RAM and ROM. For now, the only relevant modification is the extension of 8KB additional RAM at the location `0x6000` until `0x7FFF`. Because of this, SMB3 has access to a total of 10KB of RAM.

Similar to all NES games, level data in SMB3 is stored as an sequence of tiles, each 16x16 pixels in size. A single level is are often around 160 tiles wide and 27 tiles high. This means the tile data for a level consists of $160 \cdot 27 = 4320$ data locations. Because of the large amount of RAM, it was possible for the developers of SMB3 to load an entire level into memory. This was done in sections with a width of 16 horizontal tiles, where the top left tile corresponds to the first location (`0x0001` = 1) and the bottom right tile corresponds to the last location ($16 \cdot 27$ = `0x01B0` = 432). The byte directly after this accordingly corresponds to the top left tile of the next section. It turns out all the tile data is stored in sequential order in the additional RAM, starting from location `0x6000`. The value of each byte determines which tile will be present.

[24] Crystal (n.d.).
[25] Epoch (n.d.).

The remainder of the necessary data is located in the original RAM locations, `0x0000` until `0x07FF`. The data is roughly classified in corresponding groups and assigned in one of eight 256 byte long sections. However, this only results in the data being only a little more organized. A lot of inconsistencies are present, to a point where their layout might seems entirely random. Fortunately, bytes with the same or closely related function are often positioned right next to each other.

Mario's location is stored in a similar manner as the tiles. Because a single byte can only amount to 256 (2^8) different integers, it is not possible to store Mario's x and y coordinates, which must be able to over $256 \cdot 10 = 2560$, in a single byte. This is why both these values have a so called high and low byte. Because 256 pixels is equal to the width of 16 tiles, one byte, the high byte, determines the section Mario is in, while the other byte, the low byte, determines the pixel in the section on which Mario is located. Mario's x coordinate high byte is located in `0x0075`, and the low byte in `0x0090`. Mario's y coordinate high byte is located in `0x0087`, and the low byte in `0x00A2` .[26]

Enemy locations are a bit more complex. For the horizontal position, instead of an absolute coordinate, their location relative to the screen position is defined instead. Unfortunately, we were unable to find a corresponding high byte for their location, resulting in an overflow problem we were unable to resolve. The workaround for this problem will be described in later in this section. The vertical position is defined using an absolute coordinate. SMB3 has support for the presence of up to 5 enemies at once, which are sequentially stored in descending order. Additionally, because the values only get reset when a new enemy has taken its place in the data, it is necessary to make use of an additional byte, in which the display status of the enemy is defined. The corresponding values for different enemy states is available in the RAM map. The horizontal positions are defined in the locations `0x00AB-0x00B0`, the vertical positions in the locations `0x00A2-0x00A7`, and the status in the locations `0x0660-0x0665`. In order to transfer this data to the input data, the horizontal position of the screen is required as well, which is located at `0x00FD` and `0x0012`.

In the early levels, two types of projectiles are present, both of which two can be present on screen at any given time. The data follows the same rules as the enemies, where the horizontal byte is relative to the screen position, and the vertical is an absolute coordinate. Because the projectiles do not have an assigned status byte, the program computes its speed, by comparing previous frame data with the current frame data, in order to determine whether it should be considered an input. For the first projectile, a fireball, the horizontal position is located in `0x05CB` and `0x05CC`, and the vertical position in `0x05C1` and `0x05C2`. For the second projectile, a boomerang, the horizontal position is located in `0x05CD` and `0x05CE`, and the vertical position in `0x05C3` and `0x05C4`.

Lastly, we need to compute a fitness value. This will consist of Mario's x coordinate, located in `0x0075` and `0x0090`, the amount of time left, located in `0x05EE` to `0x05F0` ,[27] and the achieved score, located in `0x0715` to `0x0717`. The relative significance of these values will be discussed in the following section.

6.1.3 Converting the Data to NN Inputs

The retrieved data needs to be modified in order for the neural network to be able to successfully utilize it. The inputs need to be in accordance with the display and what a normal player would observe, as well as be limited in the amount of inputs. This is achieved by defining the inputs as an approximation of the space surrounding Mario, where each

[26] Crystal (n.d.).
[27] Crystal (n.d.).

input represent a single 16 by 16 pixel tile. Each tile can represent one of three value: a hitbox is present, which is represented by a white square and will output a positive value; an enemy or projectile is present, which is represented by a black square and will output a negative value, or nothing is present, which is represented by the absence of a colour and an output of zero. In accordance with the display, the inputs are 16 tiles wide. The height is 15 tiles. This leads to a total input size of $16 \cdot 15 = 240$, defined as a single array. Each frame, this array is updated to reflect the current status of the game. Before the updated data for the next frame is added to the array, the input array is cleared. This is achieved by setting all values to zero, by means of a simple for loop.

```
1       -- Refresh inputs
2       for c = 1, 16*16, 1 do
3           tiles[c] = 0
4       end
```

Mario's location can be determined directly from memory using the appropriate byte locations. When Mario goes off-screen at the top of the level, the vertical high byte goes from zero to $0-1 = 255$, resulting in misleading y coordinates. Although it is very unlikely this will occur during the experiment, this is circumvented anyway.

```
1       -- Get Mario Coordinates
2       marioX = memory.readbyte(0x00090) + memory.readbyte(0x00075) * 256 + 8
3       if memory.readbyte(0x00087) ~= 255 then
4           marioY = memory.readbyte(0x000A2) + memory.readbyte(0x00087) * 256
5       else
6           -- (The vertical Low byte is equal to 255 on the topmost row of the
                map)
7           marioY = 0
8       end
```

Every frame, the program tests the data location of the tiles around Mario for their values, and determines if a hitbox is present. To achieve this, the x and y coordinates for all 240 tiles are determined, after which those are converted to a tile location. Using the previously described relations, this location is converted to the location within the RAM. By reading this byte and comparing the value to a list of tiles that contain hitboxes, the presence of a hitbox is determined. Because of the limitations of lua, this results in an excessively large if statement, which will be omitted in the following code. The full list of values can be examined in either the included full version of the program, or the RAM map. For each tile, the array is updated to a value of 1 if the presence of a hitbox is detected.

```
1       -- Check all tiles around Mario for hitboxes and update inputs (hitboxes)
2       for i = 1, 241, 1 do
3           tileX = marioX - 128 + (16*(i-1))%256
4           tileY = marioY - 128 + 16*math.floor((i-1)/16)
5
6           tileHorizontalLow = math.floor(tileX/256)
7           tileHorizontalHigh = math.floor((tileX%256)/16)
8           if tileY >= 0 then
9               tileVertical = math.floor(tileY/16)
10          else
11              -- Dit is voor dezelfde redenen als bij marioY hierboven
12              tileVertical = 0
```

```
            end
            tileData = memory.readbyte(0x6000 + tileHorizontalLow*27*16 +
                tileHorizontalHigh + tileVertical*16)
            if tileData == 44 or tileData == 83 or tileData == 85 or tileData ==
                87 or tileData == 95 or tileData == 97 or tileData == 99 or
                tileData == 103 or
```

```
            then
                tiles[i] = 1
            end
        end
```

After this, the array is updated for the locations of all enemies. Since the enemies are located relative to the screen position, their absolute x coordinates can be determined by adding the horizontal position of the screen to their location. Because the coordinates are calculated from the top left of the actual hitbox of the enemy, half the length of their sprite (8) is added as well. The y position can be determined simply by adding 256 to the byte. Next, these two values are converted to a tile position relative to Mario. Note the enemies are calculated in in reverse order. This was done to better reflect their positions in the RAM, but it is not required.

The solution to the overflow problem is also implemented here. The inputs are not updated for enemies positions at the 3 columns at both ends of the screen. The overflow would cause the coordinates to be displayed on the wrong side of the inputs array, but does not continue to within these border, as the enemy despawns before that point is reached. Lastly, if the enemy status is of a correct value as well, the inputs are updated with an value of -1. The horizontal and vertical tile locations are converted to the appropriate location in the array, by multiplying the vertical value by the length of one row (16) and adding the horizontal value.

```
        -- Update frame-dependent Variables
        screenHorizontalPosition = memory.readbyte(0x00FD) +
            memory.readbyte(0x0012)*256

        -- Get enemy data
        for i = 1, 5, 1 do
            enemyAlive[i] = memory.readbyte(0x665 - i + 1)

            enemyX = screenHorizontalPosition + memory.readbyte(176 - i + 1) + 8
            enemyHorizontal[i] = math.floor((enemyX - marioX) / 16) - 7

            enemyY = memory.readbyte(167 - i + 1) + 256
            enemyVertical[i] = math.floor(enemyY / 16) + 9 - math.floor(marioY/16)
        end
        -- Circumvent enemy position overflow - No enemies render outside the two
            lines
        for i = 1, 5, 1 do
            if enemyHorizontal[i] < -12 or enemyHorizontal[i] > -3 then
                EnableEnemyRender[i] = 0
            else
                EnableEnemyRender[i] = 1
            end
        end
        -- Update inputs (enemies)
```

```
        for i = 1, 5, 1 do
            if EnableEnemyRender[i] == 1 then
                if enemyAlive[i] == 2 or enemyAlive[i] == 5 then
                        tiles[(enemyVertical[i])*16 + enemyHorizontal[i]] = -1
                end
            end
        end
```

The projectile locations are calculated differently. It is possible to calculate their position with only the available low byte. By making use of Mario's position and computing either the modulo of 16, for the horizontal position, or an division with 16, for the vertical position, and adding a certain integer, the relative position to Mario can be determined. In order for the programme to know when to stop rendering the projectile, it calculates when it has stopped moving. When active, projectiles are always moving, and only when they have disappeared from screen do their coordinates stop changing. Because the projectile does not always move every single frame, the data of the two previous frames is stored. When the program detects one of the projectiles is moving, it updates the appropriate location of the input array with a value of -1.

```
        -- Get Projectile data (fireball)
        fireballHorizontal1 =
            math.floor((memory.readbyte(0x05CB)+256-marioX)/16)%16 - 6
        fireballHorizontal2 =
            math.floor((memory.readbyte(0x05CC)+256-marioX)/16)%16 - 6
        fireballVertical1 = math.floor((memory.readbyte(0x05C1) + 256 -
            marioY)/16) + 9
        fireballVertical2 = math.floor((memory.readbyte(0x05C2) + 256 -
            marioY)/16) + 9

        fireballDirection1 = fireballSecondPrevious1 - memory.readbyte(0x05CB)
        fireballSecondPrevious1 = fireballPrevious1
        fireballPrevious1 = memory.readbyte(0x05CB)

        fireballDirection2 = fireballSecondPrevious2 - memory.readbyte(0x05CC)
        fireballSecondPrevious2 = fireballPrevious2
        fireballPrevious2 = memory.readbyte(0x05CC)
```

```
        -- Update inputs (projectiles)
        if fireballDirection1 ~= 0 then
            tiles[fireballVertical1*16 + fireballHorizontal1] = -1
        end
        if fireballDirection2 ~= 0 then
        tiles[fireballVertical2*16 + fireballHorizontal2] = -1
        end
```

As a final step, all this data is drawn on screen. First an empty box is drawn, after which the input tiles are drawn over this. Mario's location relative to the inputs is drawn as well.

```
        -- Draw Input Box - empty
        gui.drawBox(3, 11, 16*4 + 4, 15*4 + 12, 0xff000000 ,0x40000000)
        gui.drawLine(15, 12, 15, 71, 0xff888888)
        gui.drawLine(56, 12, 56, 71, 0xff888888)
```

```
        -- Draw inputs
        for layer = 1, 15, 1 do
            for i = 1, 16, 1 do
                sqX = 4 + (i-1) * 4
                sqY = 12 + (layer-1) * 4
                -- Enemies and projectiles
                if tiles[(layer-1)*16+i] == -1 then
                    gui.drawBox(sqX, sqY, sqX + 3, sqY + 3, 0xFF000000, 0xFF000000)
                else
                -- Hitboxes
                if tiles[(layer-1)*16+i] == 1 then
                    gui.drawBox(sqX, sqY, sqX + 3, sqY + 3, 0xffffffff, 0xffffffff)
                else
                -- Empty space
                    gui.drawBox(sqX, sqY, sqX + 3, sqY + 3, 0x000000, 0x000000)
                end
                end
            end
        end

        -- Draw position Mario
        gui.drawBox(37, 45, 38, 51, 0xFF740858, 0xFF740858)
        gui.drawLine(34, 51, 41, 51, 0xFF740858)
```

6.1.4 Data Interchange Between Lua and Python

When a new genome is run, Python sends its genome data to Lua. This is done in `mario.py`, using the following section of code. The data exchange is based on JSON, which corresponds with the JSON format used in genome saves detailed above. Initially, we wanted to use named pipes (also called FIFOs) for data exchange, but it turned out that this feature was extremely difficult to use with Microsoft Windows, so we turned to the low-tech version: regular text files.

The `FR` class is a wrapper around a file which simulates a blocking read such as the one you would have if the file were a socket and flushes after a write, which are the necessary semantics to make exchanging data using text files possible. The files `ptol` and `ltop` are used to exchange data from Python to Lua and Lua to Python, respectively. The other functions (`fitnessFunction`, `getInput`, `sendOutput`, `sendGenome`, `sendMaxFitness`) are used to send and receive data, the fitness and inputs are received from Lua and the genome and outputs are sent by Python.

When genomes got stuck, the Lua code sent a stuck value which indicated to the Python code to continue to the next genome, and if the genome would have a high enough fitness value the Lua code would signal completion.

```
class FR:
    def __init__(self, f):
        self.f = f

    def readline(self):
        i = None
        while not i:
            i = self.f.readline()
        return i

```

```
    def write(self, data):
        self.f.write(data)
        self.f.flush()

inFifo = FR(open("ltop", "w+"))
outFifo = FR(open("ptol", "w+"))

def fitnessFunction(input, output):
    return json.loads(inFifo.readline())["fitness"]

class EndRun(RuntimeError):
    pass

def getInput():
    while 1:
        ip = json.loads(inFifo.readline())
        if (ip.get("noInput", False)):
            raise StopIteration
        if (ip.get("complete", False)):
            raise EndRun
        yield ip["input"]

def sendOutput(output):
    data = json.dumps({"output": [*output[:2], 0, 0, *output[2:]]}) + '\n'
    outFifo.write(data)

def sendGenome(pool, species, genome, nthGenome):
    data = {"connections": sorted([(conn.source.nodeId, conn.sink.nodeId,
        conn.weight) for conn in genome.connections.values()
                                if conn.enabled], key=lambda x: x[0]),
            "nodes": sorted([node.nodeId for node in genome.nodes.values()]),
            "maxFitness": pool.maxFitness,
            "generation": pool.generation,
            "genpercent": str(100*(nthGenome + 1) / len(species.genomes))[:4],
            "genomeId": nthGenome + 1,
            "speciesId": species.id}
    asJson = json.dumps(data)
    outFifo.write((asJson + '\n'))

def sendMaxFitness(pool):
    data = json.dumps({"maxFitness": pool.maxFitness}) + '\n'
    outFifo.write(data)
```

```
        oldTime = time
        time = memory.readbyte(0x05EE)*100 + memory.readbyte(0x05EF)*10 +
            memory.readbyte(0x05F0)-300
        score = memory.readbyte(0x0717)*10 + memory.readbyte(0x0716)*2560

        if fitnessVertical == 0 and marioY <= 340 then
         fitnessVertical = 1
       end

        fitness = marioX + time + fitnessVertical*250

   if marioY >= 390 then
```

```
12      stuck = true
13   end
14
15   if (memory.readbyte(0x0303) == 3 or memory.readbyte(0x0303) == 17) and
         memory.readbyte(0x00ED) ~= 1 then
16      stuck = true
17   end
18
19   if (fitness > fitnessHighest) then
20      fitnessHighest = fitness
21   end
22
23   if (fitnessHighest > fitness) then
24      fitnessTimer2 = fitnessTimer2 + 1
25   end
26
27   if (fitnessTimer2 > 6) then
28      fitnessTimer2 = 0
29      stuck = true
30   end
31
32        if oldTime > time then
33            fitnessTimer = fitnessTimer + 1;
34        elseif (fitnessPrevious - time) < (fitness - time) then
35            fitnessTimer = 0;
36        end
37        if fitnessTimer > 3 then
38            stuck = true
39        end
40
41   if stuck == true then
42               writeLine({["noInput"] = 1})
43      stuck = false
44               break
45   end
```

```
1        writeLine({["input"] = tiles})
2
3        data = readLine()
4        outputs = data["output"]
5
6        writeLine({["fitness"] = fitness})
```

Every frame, the input array and the fitness value are sent to the current neural network. At first, the fitness function included the x value of Mario, the time left, and the score. Halfway during the experiment however, during debugging, the score was omitted. The reasons for this will be explained in section 6.3.

Python then returns the data of the output nodes to the lua script. The outputs are first converted to a boolean. With this the controller is updated. Bizhawk's `joypad.set()` function requires a specific combination of strings, which are first stored in the variable `controller`. The array `buttonNames` contains all the button names.

```
1    buttonNames = {
2        "A",
```

```
        "B",
        "Up",
        "Down",
        "Left",
        "Right"
    }
```

```
        for i = 1, 6 do
            if outputs[i] > 0.95 then
                outputsAbsolute[i] = true
            end
            if outputs[i] < 0.95 then
                outputsAbsolute[i] = false
            end
        end
```

```
        -- Refresh Outputs
        for i = 1, 6 do
            controller["P1 " .. buttonNames[i]] = false
        end

        for i = 1, 6 do
            controller["P1 " .. buttonNames[i]] = outputsAbsolute[i]
        end
        joypad.set(controller)

        emu.frameadvance();
    end
end
```

With the genome data and the inputs (to the neural network) the network can be visualized. If a node is active, its colour will change to white. Connections, visualized as lines, will change colour according to the value of the source node. White with a low opacity when it is equal to 0, red when it is equal to -1 and green when it is equal to 1. This is much easier said than done however. The programme will not be discussed in this paper however, but the full script is available as an attachment.

6.2 The Set-up of and Time Allocation for the Experiment

Using the lua and python scripts the experiment can be executed. Because the learning processes can take a very long time, we have allocated a maximum 24 hours for the experiment to run. The experiment is originally to run on 3 separate computers at once. Since the evolution of the network is essentially random, a comparison between the differences of the various evolutions could yield interesting results. Due to time limitations and debugging issues however, the experiment could only run for an extended period of time on a single computer. This process will be discussed in more detail in the next chapter. As mentioned before, the freeware emulator Bizhawk was used. For the NEAT algorithm, the variables values as discussed in section 5.3 were used.

The set-up for the experiment is done in `mario.py`.

```
    outFifo.write(data)

```

```
if len(sys.argv) > 1:
    loadPath = sys.argv[1]
    genNumber = sys.argv[2]
    pool = Loader(loadPath, genNumber).getGenomes(fitnessFunction)
    pool.fitnessFn = fitnessFunction
else:
    pool = NetworkPool(initialGenome, fitnessFunction)

try:
    while 1:
        pool.runOnce(getInput, sendGenome, sendOutput)
        pool.nextGeneration()
        print("Max fitness in generation %d (S %d G %d): %.20f | Staleness of
            species %d: %d" % (pool.generation, pool.maxSpecies.id,
            pool.maxGenome.id, pool.maxFitness, pool.maxSpecies.id,
            pool.maxSpecies.staleness))
except EndRun:
    print("Run ends!")
```

The configuration values used are defined in `Mario.toml`.

```
[compatibility]
excess = 2.0
disjoint = 2.0
weights = 0.8

[speciation]
species_difference = 1
stagnation_threshold = 15
champion_threshold = 5

[rates]
perturbation = 0.9
mutation = 0.5
randomset = 0.1
disabled_inherit = 0.75
crossover = 0.75
interspecies = 0.01
newnode = 0.5
newlink = 2.0
biasmut = 0.4

[pool]
population = 300
```

The `compatibility` section details the c_1, c_2 and c_3 values used in the experiment. The `speciation` section details the constants that concern speciation. `species_difference` is the δ_t value, `champion_threshold` is the threshold describing the number of genomes necessary for copying the best genome of a species to the next generation.

The `rates` section describes mutation rates used in the experiment. `perturbation` is the chance a connection is perturbed on the connection mutation. `mutation` is the chance a connection is mutated. `randomset` is the chance a connection weight is randomly set (1 - `perturbation`). `disabled_inherit` is the chance a disabled gene is disabled in its offspring. `newnode` and `newlink` are the add node and add connection rates. `biasmut` is the chance a new connection will connect to the bias node, a modification found to be

helpful in our testing.

The `pool` section describes the population size.

The experiment ran for roughly 24 hours before having to be cut short.

6.3 Results

6.3.1 Level Progress

Although many problems arose during large parts of the experiment, which required an extensive debugging phase, the experiment can be considered successful. The NEAT algorithm has managed to complete and learn a sizeable segment of the level. After running the experiment for 92 generations it completed 64% of level 1-1, up to the point indicated by the red line in figure 19. It reached a distance of 1792 pixels out of the total distance of 2816 pixels, of which the last 500 would not have presented any challenge to the network. This means the algorithms was incredibly close to finding beating the level, and had it had time to run longer, it could definitely have finished it. It should have noted that, because of the debugging problems, the 24 hours used to run the experiment used generations that had been manipulated in many different ways through different configuration options and bug fixes, creating networks that were certainly not the best networks that could have been generated in that time.

In figure 20 it can be seen that the experiment had was not able to pass the green pipes in the limited time available for learning. However, if given more time there would have been a chance that it would have made it past the green pipes because it had previously overcome similar obstacles. The same is true for the other obstacles that follow the green pipes in figure 21 where the experiment got stuck.

At the beginning of the experiment, NEAT did not make a lot of progress. Many genomes would not have initially mutated the structure to move right, and they would just stand there. Sometimes this did happen, and the genome would die walking into the first enemy. As hypothesis, this proved to be the first local optimum. The process of improvement was very incredibly slow. Even in the generations which have hight fitness scores, ± 90% of all genomes would not make it nearly as far. Even in later generations, where stale species would have had to be removed, these 'dumb' genomes would still be present. This can be because of either the nature of the algorithm, or a mistake in the implementation.

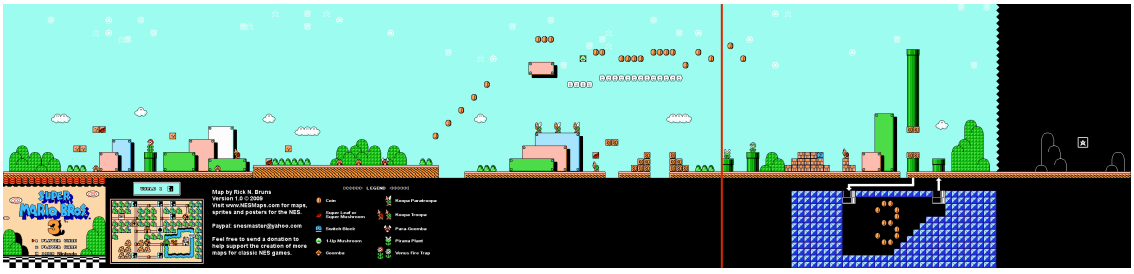

Figure 19: A zoomed-out view of level 1-1 with a red line indicating the maximum distance reached.

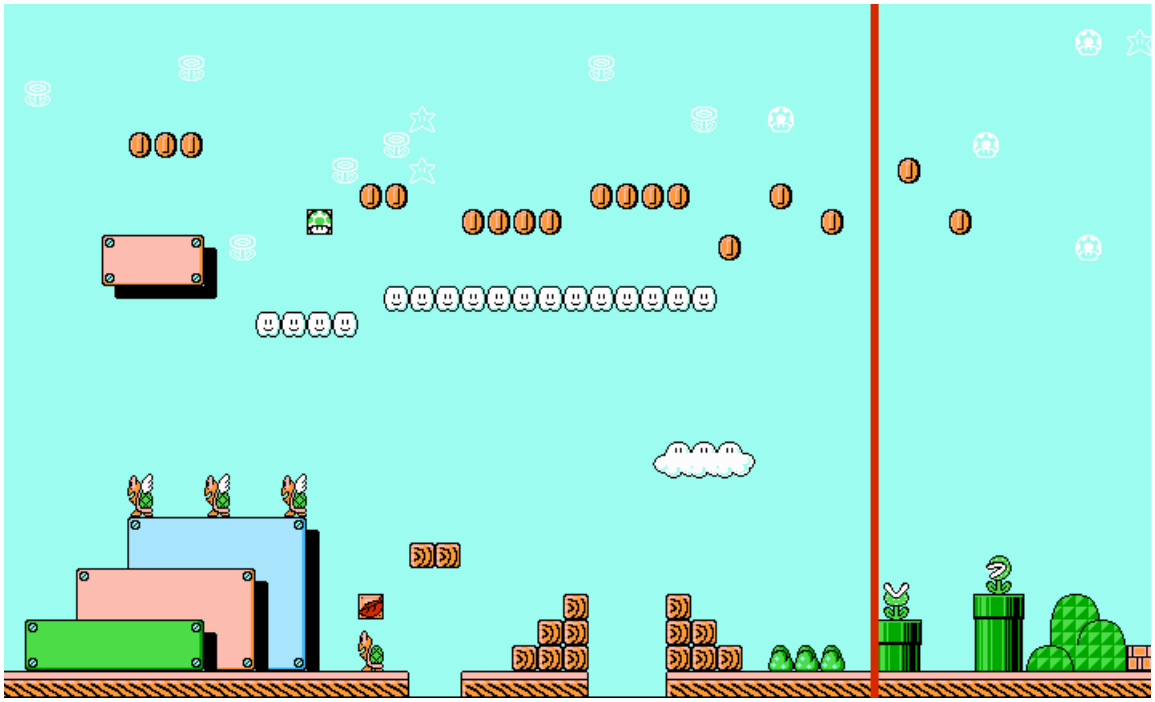

Figure 20: A zoomed-in view of level 1-1 with a red line indicating the maximum distance reached.

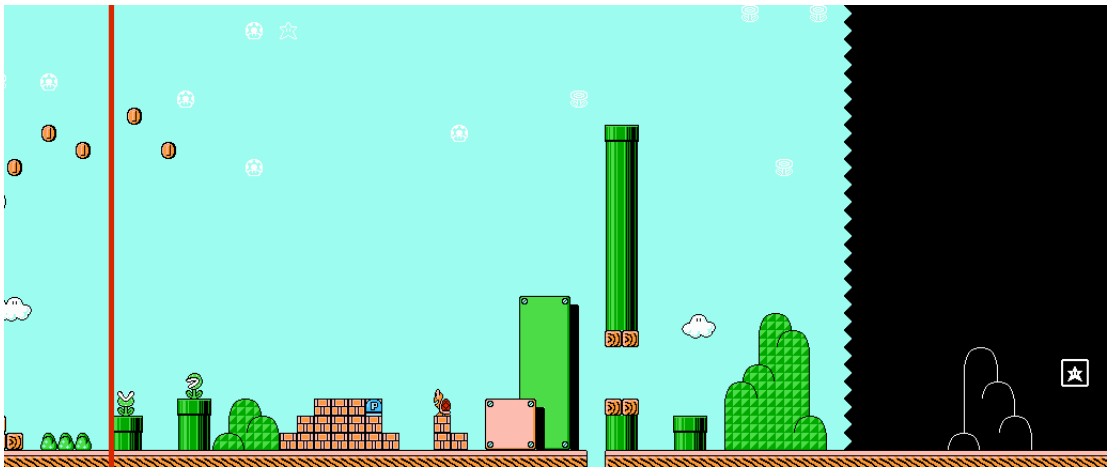

Figure 21: A zoomed-in view of level 1-1 with a red line indicating the maximum distance reached showing the final obstacles to be overcome.

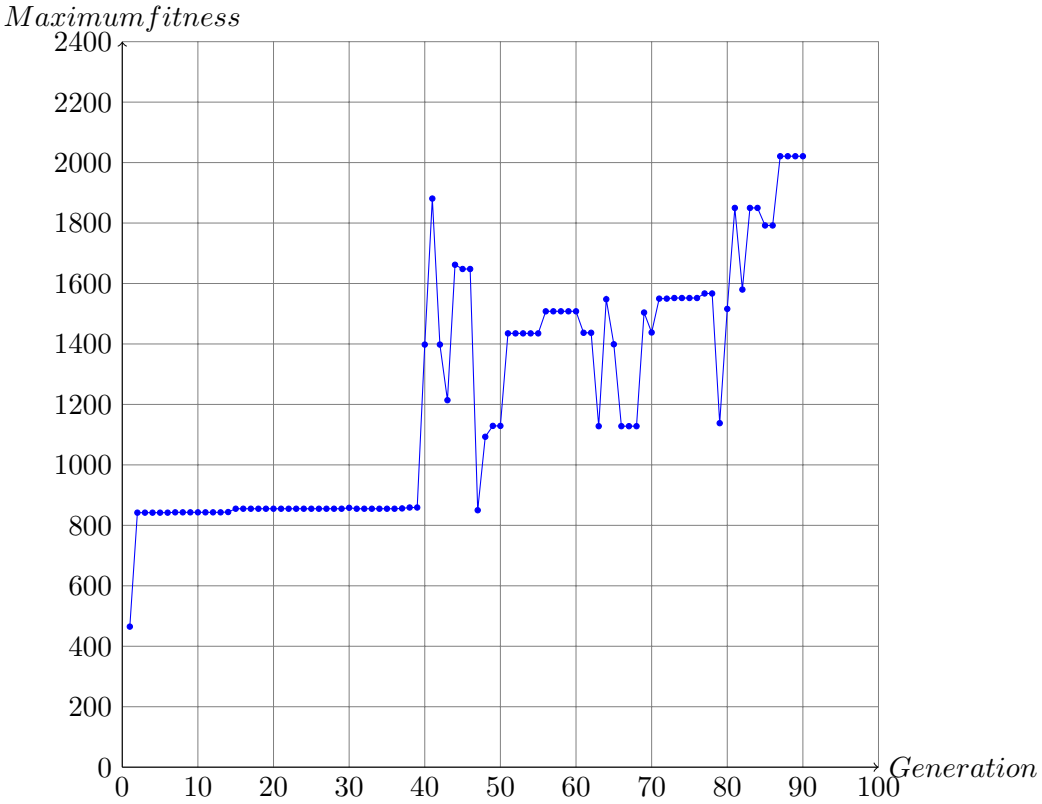

Figure 22: The maximum fitness of each generation.

6.3.2 Maximum Fitness

The maximum fitness of each generation is presented in figure 22. For the generations 2 until 39, NEAT did not improve. The obstacle present at this fitness value turned out to be the piranha plant in the first green pipe, which would come up just as Mario would try to pass the pipe. This obstacle, which was as hypothesised the second local optimum, proved to be too difficult for the algorithm to solve and although it was theoretically possible, it would have taken a considerable amount of time. It would require a genome to wait for a small period of time, until the enemy had disappeared into the pipe again, before continuing. Apparently, this is either incredibly difficult to learn, or we were just very unlucky. At generation 40, it was decided to remove the plant from the game, in order for Mario to be able to continue. Immediately afterwards, the fitness went up significantly in only 2 generations. The area behind this obstacle proved to be little challenge for the network. After this, the maximum fitness score started to fluctuates a lot for the remaining generations. It is not 100% clear why this happened, but it might have something to do with the over-complication of the networks, resulting in conflicting outputs, and little efficiency in increasing fitness. Sometimes, no score stays the same for a couple of generations.

6.3.3 The Visualisation of Genomes

The visualisation would display the currently active genomes at the top of the screen, making them easy observable. Over the generations, the genomes got gradually more complex, as more mutations were applied. In later generations, the genomes got so complex the visualisation became incredibly cluttered. It became impossible to learn a lot about its structure only by looking at it, communicating the complex and intricate topologies

NEAT is able to generate. Sadly and as had been hypothesized, many of the structures generated in this experiment were not completely functional.

6.3.4 Further Noteworthy Observations

Because the time left was included in the fitness value, running would have been very likely to evolve in the population. Interestingly, a lot of genomes never evolved this ability. This is odd, because not being able to run severely lowers their fitness score. Running might have a negative side as well, resulting in a lower fitness score nevertheless. But because of the random nature of the learning process, it is of course also possible this abnormality did not evolve only due to chance.

Another interesting occurrence is that, and this ties into the previous paragraph, there seemed to be a divergence in strategies different species employed. Some species would run through the level very fast, in a way which would be unnatural for most human players, and 'anticipate' upcoming obstacles before they appeared on screen. Since in the previous generations this strategy proved effective, it does not need to remember where and when enemies might appear and it it can just do what the structure tells it to do. But other species where on the complete other side of the spectrum. They evolved the ability to see enemies right in front of them, and they would circle around them before jumping on them. Normally, this topology would cause the genome to slow down and thus lose fitness when compared to the speed-running species, but somehow they managed to evolve side by side. This is because the fast running species had a higher chance of dying earlier in the level, where the described slower species would cleverly avoid the enemies. This means the fitness of the species, which is shared among all the genomes present within, would be very much alike, with the fast species having a much higher deviation.

During the learning process, the algorithm managed to evolve some structures which lead to very interesting consequences. Several genomes managed to find and exploit certain glitches we did not even know existed. For example, they found a possibility to jump while in the air, something that is normally not possible, when falling next to a wall. It is even more incredible, since this manoeuvre apparently requires frame perfect timing, and is really difficult for a human to perform.

7 Conclusion

The results show it is possible to make a computer learn itself to play video games, through use of the machine learning concepts of artificial neural networks and Neuro-evolution.

This experiment, however, only scratched the surface of the NEAT functionality. This system could be applied to different levels and even different games. With better computational speeds, the learning process can be sped up significantly and applied to much more challenging games, both in terms of player challenge and computational necessities. One day, NEAT, or a variation on the concepts of NEAT, could be used to develop more functional AI for applications in video games, but also outside of them. This same system could in theory be used to train a wide variety of AI, such as

For video games, there might even be more efficient learning methods available, which are not based on Neuro-evolution but instead on Deep Learning or non-neural network based forms of machine learning.

NEAT could also be used by game developers to test their game in terms of difficulty. This would be a lot less expensive than hiring play-testers.

Has our extensive research and experimentation answered our research question? This is rather difficult to conclude. For one, the level had to be slightly altered for, in some places, it proved to difficult for the ANN. There was also not enough time for the ANN to be run until the level, be it in altered state, could truly be finished. Because of these discrepancies, the validity of the hypothesis is still debatable. Thus further research, spread over a longer period of time ought to be done for this to be settled.

Answering the sub-questions has proven to be more successful. For throughout the report, all of the sub-questions received a satisfactory answer in their respective chapters.

7.1 Reflective statement

We have barely scratched the surface when it comes to this domain of research. We could try different levels of SMB3, a different game, different variables in NEAT or even an ANN for the variables in NEAT.

To explore the world of ML we wanted to set a goal for ourselves, something to prove that we understand what we have learnt and something to apply our newly acquired knowledge on. This meant that it needed to be complex enough to warrant the use of ML and at the same time still be feasible enough to make in a restricted time frame. It also needed to be interesting, maybe produce something which, in its restrictive field of operation, could perform similar to a human being. In the end though, it can be concluded that we bit off more than we could chew. This together with some regrettable setbacks resulted in an unsatisfactory report. With some final tweaks however, we believe it should still be sufficiently presentable.

8 References

Bayes, T. (1763). An essay towards solving a problem in the doctrine of chances. *Phil. Trans.*, *53*(0), 370-418. Retrieved from `http://www.stat.ucla.edu/history/essay.pdf` doi: 10.1098/rstl.1763.0053

Block, H.-D. (1962). The perceptron: A model for brain functioning. i. *Reviews of Modern Physics*, *34*(1), 123.

Braun, H., & Weisbrod, J. (1993). Evolving feedforward neural networks. In R. C. Albrecht R.F & N. Steele (Eds.), *Proceedings of annga93, international conference on artificial neural networks and genetic algorithms* (p. 25-32). Innsbruck: Springer-Verlag.

CodeReclaimers. (2008-2017). *Neat-python.* Retrieved from `https://github.com/CodeReclaimers/neat-python/blob/master/neat/nn/recurrent.py#L11`

Crystal, D. (n.d.). *Super mario bros. 3 ram map.* Retrieved from `http://datacrystal.romhacking.net/wiki/Super_Mario_Bros._3:RAM_map`

Dasgupta, D., & McGregor, D. (1992). Designing application-specific neural networks using the structured genetic algorithm. In D. Whitley & J. D. Schaffer (Eds.), *Proceedings of the international conference on combinations of genetic algorithms and neural networks* (p. 87-96). Piscataway, New Jersey: IEEE Press.

Epoch, S. (n.d.). *sm3mix disassembly.* Retrieved from `http://sonicepoch.com/sm3mix/disassembly.html`

Minsky, M., & Papert, S. (1969). *Perceptrons: An introduction to computational geometry.* Cambridge, MA, USA: MIT Press.

Mitchell, T. M. (1997). *Machine learning* (1st ed.). New York, NY, USA: McGraw-Hill, Inc.

Ng, A. (n.d.). *Cs229 lecture notes - supervised learning.* Retrieved from `http://cs229.stanford.edu/materials.html`

Pujol, J. C. F., & Poli, R. (1998). Evolving the topology and the weights of neural networks using a dual representation. *Special Issue on Evolutionary Learning of the Applied Intelligence Journal*, *8*(1), 73-84.

Russell, S., & Norvig, P. (2009). *Artificial intelligence: A modern approach* (3rd ed.). Upper Saddle River, NJ, USA: Prentice Hall Press.

Silver, D., & D. Hassibis, G. D. (2016, jan). *Alphago: Mastering the ancient game of go with machine learning.* Retrieved from `https://research.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html`

Skaggs, W. E. (2013, aug). *Nervous system.* Retrieved from `http://www.scholarpedia.org/article/Nervous_system`

Stanley, K., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99-127.

Stigler, S. M. (1982). Thomas Bayes' Bayesian inference. *Journal of the Royal Statistical Society*, *Series A*(145), 250-258. Retrieved from `https://bit.ly/stiglerbayes`

Sykes, C. (2011). *Show and tell: My neural network machine.* interview. Retrieved from `http://www.webofstories.com/play/marvin.minsky/137`

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*(49), 433-460. Retrieved from `https://www.csee.umbc.edu/courses/471/papers/turing.pdf`

UFLDL, S. (2013, april). *Gradient checking and advanced optimization.* Retrieved from `http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization`

Unsupervised feature learning and deep learning. (2013, mar). Retrieved from `http://ufldl.stanford.edu/wiki/index.php/Main_Page`

9 Appendices

LinearRegressionFull.m Non-Vectorized

```
%Inport Training Examples
x0 = 1;
x1 = load('x.dat');
y = load('y.dat');

%Set variables
m = length(y);
alpha = 0.07;
max_iterations = 1500;
theta0 = 0;
theta1 = 0;

%Linear Regression
for i = 0:max_iterations
  for i = 1:m
    h[i] = theta0 * x0 + theta1 * x1[i];
  end

  J0 = 0;
  J1 = 0;
  for i = 1:m
    J0 = J0 + (h[i] - y[i]) * x0;
    J1 = J1 + (h[i] - y[i]) * x1[i];
  end

  theta0 = theta0 - alpha * 1/m * J0;
  theta1 = theta1 - alpha * 1/m * J1;
end
```

LinearRegressionFull.m Vectorized

```
%Import Training Examples
x = load('x.dat');
y = load('y.dat');

%Set variables vectorized
m = length(y);
x = [ones(m, 1), x];
alpha = 0.07;
max_iterations = 1500;
theta = zeros(size(x(1,:)))';

%Linear regression vectorized and shortened implementation
for i = 1:max_iterations
  theta = theta - alpha * ((1/m) * x' * ((x * theta) - y));
end
```

NeuralNetworkFull.m

```
%The data set
```

```
x1 = [0; 0; 1; 1];
x2 = [0; 1; 0; 1];
y =  [0; 0; 0; 1];

%Initialisation weights
W1 = stdnormal_rnd(2,3);
W2 = stdnormal_rnd(3,1);
b1 = stdnormal_rnd(1,3);
b2 = stdnormal_rnd(1,1);

%Initialisation variables
alpha = 1;
labda = 0.01;
max_iterations = 5000;
m = rows(x1);

%sigmoid function
function s = f(z)
  s = 1./(1 + exp(-z));
end


for i = 1:maxi

    %Forward propagation
    %Layer 1 (input)
    a1 = [exx1, exx2];

    %Layer 2
    z2 = a1 * W1 + b1;
    a2 = f(z2);

    %Layer 3 (output)
    z3 = a2 * W2 + b2;
    h = f(z3);
    a3 = h;

    %Backward propagation
    % Delta values
    delta3 = -(exy - h)   .* (a3 .* (1 - a3));
    delta2 = (delta3 * W2') .* (a2 .* (1 - a2));

    W2 = W2 - alpha * (a2' * delta3 * 1/m); % + lambda * W2);
    W1 = W1 - alpha * (a1' * delta2 * 1/m); % + lambda * W1);
    b2 = b2 - alpha * (sum(delta3) * 1/m);
    b1 = b1 - alpha * (sum(delta2) * 1/m);

end
```

genome.py

```
import random
import math
import copy
from statistics import mean
```

```
5  from . import configuration
6  from collections import defaultdict
7  from .nodetype import NodeType
8  import itertools
9  from pprint import pprint
10
11 GLOBAL_INNOVATION = 1
12 GLOBAL_NODES = 248
13
14 GENOME_ID = 1
15
16 def modSigmoid(x):
17     return (2 / (1 + math.exp(-4.9*x)))-1
18
19 class NodeGene():
20     def __init__(self, nodeType, nodeId=None, activationFunction = modSigmoid):
21         global GLOBAL_NODES
22         global GLOBAL_INNOVATION
23         self.nodeId = nodeId or GLOBAL_NODES
24         if not nodeId:
25             GLOBAL_NODES += 1
26         self.nodeType = nodeType
27         self.activationFunction = activationFunction
28
29     def __eq__(self, other):
30         try:
31             return self.nodeId == other.nodeId
32         except:
33             return False
34
35     def __ne__(self, other):
36         return not self.__eq__(other)
37
38     def __hash__(self):
39         return hash(self.nodeId)
40
41     def __repr__(self):
42         if self.nodeType in (NodeType.SENSOR, NodeType.BIAS):
43             return "Node(#%r, %r)" % (self.nodeId, self.nodeType)
44         return "Node(#%r, %r, %s)" % (self.nodeId, self.nodeType,
               self.activationFunction.__name__)
45
46 class ConnectionGene():
47     def __init__(self, source, sink, weight = None, connectionId = None, enabled
           = None):
48         global GLOBAL_INNOVATION
49         self.source = source
50         self.sink = sink
51         self.weight = weight or random.gauss(0.0, 1.0)
52         self.connectionId = connectionId or GLOBAL_INNOVATION
53         if not connectionId:
54             GLOBAL_INNOVATION += 1
55         self.enabled = enabled or True
56
57     def __repr__(self):
58         return "Connection(#%d, Node %r => Node %r, Weight %f, %s)" \
```

```
59            % (self.connectionId, self.source.nodeId, self.sink.nodeId,
                  self.weight, "Enabled" if self.enabled else "Disabled")
60
61    def __eq__(self, other):
62        return self.connectionId == other.connectionId
63
64    def __ne__(self, other):
65        return not self.__eq__(other)
66
67    def __hash__(self):
68        return hash(self.connectionId)
69
70class Genome():
71    def __init__(self, nodes=None, connections=None):
72        global GENOME_ID
73        self.config = configuration.getGlobalConfig()
74        self.nodes = {node.nodeId:node for node in (nodes or [])}
75        self.connections = {connection.connectionId:connection for connection in
              (connections or [])}
76        self.fitness = 0
77        self.id = GENOME_ID
78        GENOME_ID += 1
79
80        self.globalRank = 0
81
82    def addNode(self, nodeId, nodeType):
83        node = NodeGene(NodeType(nodeType), nodeId)
84        self.nodes[node.nodeId] = node
85
86    def addConnection(self, source, sink, weight=None, connectionId=None,
          enabled=None):
87        connection = ConnectionGene(source, sink, weight, connectionId, enabled)
88        self.connections[connection.connectionId] = connection
89
90    def mutateNode(self):
91        if (not self.connections):
92            return False
93
94        node = NodeGene(NodeType.HIDDEN)
95        # Add a node between an existing connection
96        conn = random.choice(list(self.connections.values()))
97        conn.enabled = False
98
99        new_connections = (ConnectionGene(conn.source, node, weight = 1),
              ConnectionGene(node, conn.sink, weight=conn.weight))
100
101       for connection in new_connections:
102           self.connections[connection.connectionId] = connection
103
104       self.nodes[node.nodeId] = node
105
106   def mutateConnection(self, bias=False):
107       if not bias:
108           source_eligible = [node for node in self.nodes.values() if
                  node.nodeType in (NodeType.SENSOR, NodeType.BIAS,
                  NodeType.HIDDEN, NodeType.OUTPUT)]
```

```
109         else:
110             source_eligible = [node for node in self.nodes.values() if
                    node.nodeType == NodeType.BIAS]
111         sink_eligible = [node for node in self.nodes.values() if node.nodeType in
                (NodeType.HIDDEN, NodeType.OUTPUT)]
112 
113         existing_connections = {(conn.source, conn.sink):conn for conn in
                self.connections.values()}
114 
115         combinations = [combination for combination in
                itertools.product(source_eligible, sink_eligible) if combination[0]
                != combination[1] and combination not in existing_connections]
116 
117         if (not combinations):
118             return False
119         chosen = random.choice(combinations)
120 
121         self.addConnection(*chosen)
122 
123         #pprint({k:v for k,v in self.connections.items() if v.enabled})
124         #print()
125 
126     def node(self, nodeId):
127         return self.nodes.get(nodeId, False)
128 
129     def connection(self, connectionId):
130         return self.connections.get(connectionId, False)
131 
132     def copy(self):
133         return copy.deepcopy(self)
134 
135     def mutateWeights(self):
136         for connection in self.connections.values():
137             if (random.random() < self.config["rates"]["perturbation"]):
138                 connection.weight += (random.gauss(0.0, 0.5))
139             else:
140                 connection.weight = random.gauss(0.0, 1.0)
141 
142     def mutate(self):
143         if (random.random() < self.config["rates"]["mutation"]):
144             self.mutateWeights()
145 
146         n = self.config["rates"]["newnode"]
147         while (random.random() < n):
148             self.mutateNode()
149             n -= 1
150 
151         n = self.config["rates"]["newlink"]
152         while (random.random() < n):
153             self.mutateConnection()
154             n -= 1
155 
156         if (random.random() < self.config["rates"]["biasmut"]):
157             self.mutateConnection(True)
158 
159     def sameSpecies(self, other):
```

```
160         c_1 = self.config["compatibility"]["excess"]
161         c_2 = self.config["compatibility"]["disjoint"]
162         c_3 = self.config["compatibility"]["weights"]
163 
164         # d = c_1*E/N + c_2*D/N + c_3 * W
165         nGenes = (len(self.connections), len(other.connections))
166 
167         N = 1
168 
169         # Gene connectionIds below this threshold are disjoint. Uniques higher
                than this threshold are excess.
170         if (len(self.connections) == 0) or (len(other.connections) == 0):
171             excessThresh = 0
172         else:
173             excessThresh = min(max(cId for cId in self.connections.keys()),
                    max(cId for cId in other.connections.keys()))
174 
175         matchingGenes = {gene: other.connection(gene.connectionId) for gene in
                self.connections.values() if other.connection(gene.connectionId)}
176 
177         weightDiff = [abs(a.weight - b.weight) for a,b in matchingGenes.items()]
178         if (weightDiff):
179             avgweightDiff = mean(weightDiff)
180         else:
181             avgweightDiff = 0
182 
183         otherGenes   = set(self.connections.values()) ^
                set(other.connections.values())
184 
185         disjoints = {gene for gene in otherGenes if gene.connectionId >
                excessThresh}
186         excesses = otherGenes - disjoints
187 
188         return ((c_1 * len(excesses) + c_2 * len(disjoints)) / N + c_3 *
                avgweightDiff) < self.config["speciation"]["species_difference"]
189 
190     def crossover(self, other):
191         global GENOME_ID
192         offspring = Genome()
193 
194         if (len(self.connections) == 0) or (len(other.connections) == 0):
195             excessThresh = 0
196         else:
197             excessThresh = min(max(cId for cId in self.connections.keys()),
                    max(cId for cId in other.connections.keys()))
198 
199         matchingGenes = [(gene, other.connection(gene.connectionId)) for gene in
                self.connections.values() if other.connection(gene.connectionId)]
200 
201         otherGenes = set(self.connections.values()) ^
                set(other.connections.values())
202 
203         matchingChoices = [random.choice(x) for x in matchingGenes]
204 
205         if (self.fitness > other.fitness):
```

```
206             otherChoices = [choice for choice in otherGenes if choice in
                    self.connections.values()]
207         elif (other.fitness > self.fitness):
208             otherChoices = [choice for choice in otherGenes if choice in
                    other.connections.values()]
209         else:
210             otherChoices = list(otherGenes)
211 
212         offspring.connections = {i.connectionId:i for i in (matchingChoices +
                otherChoices)}
213 
214         for connection in offspring.connections.values():
215             if connection.enabled == False:
216                 connection.enabled == False if random.random() <
                        self.config["rates"]["disabled_inherit"] else True
217 
218         offspring.nodes = {i.nodeId:i for i in (set(self.nodes.values()) |
                set(other.nodes.values()))}
219 
220         GENOME_ID += 1
221         offspring.id = GENOME_ID
222 
223         return offspring
```

network.py

```
1  from collections import defaultdict, namedtuple, OrderedDict
2  from .nodetype import NodeType
3  from pprint import pprint
4  import random
5  import sys
6 
7  class Node():
8      def __init__(self, nodeGene, connections):
9          self.nodeType = nodeGene.nodeType
10         self.nodeId = nodeGene.nodeId
11         self.weights = {conn.source.nodeId:conn.weight for conn in connections
12                         if conn.enabled and conn.sink == self}
13         self.activationFunction = nodeGene.activationFunction
14         self.connections = connections
15         self.dependencies = {conn.source.nodeId for conn in connections
16                              if conn.sink == self and conn.enabled}
17 
18     def __eq__(self, other):
19         if (self.nodeId == other.nodeId) and (self.nodeType == other.nodeType):
20             return True
21         return False
22 
23     def __ne__(self, other):
24         return not self.__eq__(other)
25 
26     def __hash__(self):
27         return hash(self.nodeId)
28 
29     def __repr__(self):
```

```
30          if self.nodeType in (NodeType.SENSOR, NodeType.BIAS):
31              return "Node(#%r, %r, %r)" % (self.nodeId, self.nodeType,
32                      self.weights)
33          return "Node(#%r, %r, %s, %r)" % (self.nodeId, self.nodeType,
34                  self.activationFunction.__name__, self.weights)
35
36      def __call__(self, output, inputs=None, final_out=None):
37          if (self.nodeType == NodeType.SENSOR):
38              input = inputs.get(self.nodeId)
39          else:
40              input = 0
41              for dependency in self.dependencies:
42                  val = inputs.get(dependency, 0.0)
43                  cW = self.weights.get(dependency, 0.0)
44                  if (val == None):
45                      inputs[dependency] = 0.0
46                      val = 0.0
47                  input += val * cW
48
49          if (self.nodeType == NodeType.BIAS):
50              input = 1.0
51
52          if (self.nodeType in (NodeType.HIDDEN, NodeType.OUTPUT)):
53              input = self.activationFunction(input)
54
55          output[self.nodeId] = input
56
57          if self.nodeType == NodeType.OUTPUT:
58              final_out[self.nodeId] = input
59
60          return output
61
62
63  class Network():
64      def __init__(self, genome):
65          self.nodes = OrderedDict(sorted(self.createNodes(genome), key=lambda
                x:x[0]))
66          self.values = defaultdict(float)
67
68      def node(self, nodeId):
69          return self.nodes[nodeId]
70
71      def createNodes(self, genome):
72          for nGene in genome.nodes.values():
73              connections = [c for c in genome.connections.values()
74                      if c.sink == nGene or c.source == nGene]
75              node = Node(nGene, list(connections))
76              yield (node.nodeId, node)
77
78      def runWith(self, inputdata=None):
79          data_pool = defaultdict(float)
80          output_data = defaultdict(float)
81
82          # First, run the inputs and bias nodes
83          inputs = [node for node in self.nodes.values() if node.nodeType ==
                NodeType.SENSOR or node.nodeType == NodeType.BIAS]
```

```
84          inputs.sort(key=lambda node:node.nodeId)
85
86          for ip in inputs:
87              ip(self.values, dict(zip([node.nodeId for node in inputs],
                    inputdata)), output_data)
88
89
90          # Then the other nodes
91          other_nodes = [node for node in self.nodes.values() if node.nodeType in
                (NodeType.OUTPUT, NodeType.HIDDEN)]
92
93          for node in other_nodes:
94              node(self.values, self.values, output_data)
95
96          self.values.update(output_data)
97          output_data = list(output_data.items())
98          output_data.sort(key=lambda node: node[0])
99
100         return [x[1] for x in output_data]
```

species.py

```
1   import random
2   from . import configuration
3
4   SPECIES_ID = 1
5
6   class Species():
7       def __init__(self):
8           global SPECIES_ID
9           self.config = configuration.getGlobalConfig()
10          self.id = SPECIES_ID
11          self.genomes = []
12          self.topFitness = 0.0
13          self.averageFitness = 0.0
14          self.staleness = 0
15          SPECIES_ID += 1
16
17      def calculateAverageFitness(self):
18          total = 0
19
20          for genome in self.genomes:
21              total = total + genome.globalRank
22
23          self.averageFitness = total / len(self.genomes)
24
25      def sortGenomes(self):
26          self.genomes.sort(key=lambda genome: genome.fitness, reverse=True)
```

pool.py

```
1   #
2   # NetworkPool
3   #
```

```
4
5  import pprint
6  import random
7  import sys
8  import os
9  import pdb
10 import datetime
11 import json
12 import traceback
13 from .species import Species
14 from .network import Network
15 from . import configuration
16
17 class NetworkPool():
18     def __init__(self, initialGenome=None, fitnessFunction=None):
19         print("Constructing pool.")
20         self.fitnessFn = fitnessFunction
21         self.config = configuration.getGlobalConfig()
22         self.species = []
23         if initialGenome:
24             for i in range(self.config["pool"]["population"]):
25                 print("Constructing genome %d" % i)
26                 copy = initialGenome.copy()
27                 copy.mutate()
28                 self.addToSpecies(copy)
29
30         self.maxFitness = 0.0
31         self.maxPrevFitness = 0.0
32         self.maxGenome = None
33         self.evaluatedGenomes = 0
34         self.generation = 1
35
36         self.rtdir = "run-" + datetime.datetime.utcnow().strftime("%Y%m%d-%H%M%S")
37         os.mkdir(self.rtdir)
38
39     def addToSpecies(self, genome, speciesId=None, staleness=None):
40         for species in self.species:
41             if species.genomes[0].sameSpecies(genome):
42                 species.genomes.append(genome)
43                 return
44
45         newSpecies = Species()
46         newSpecies.genomes.append(genome)
47         newSpecies.staleness = staleness or 0
48         self.species.append(newSpecies)
49
50     def rankGlobally(self):
51         globalList = [genome for species in self.species for genome in
               species.genomes]
52
53         globalList.sort(key=lambda x: x.fitness)
54
55         for n, genome in enumerate(globalList):
56             genome.globalRank = n
57
58     def totalAverageFitness(self):
```

```
59          total = 0
60
61          for species in self.species:
62              total = total + species.averageFitness
63
64          return total
65
66      def nextGeneration(self):
67          self.cullSpecies(cutToOne=False)
68          self.removeStaleSpecies()
69          self.rankGlobally()
70          for species in self.species: species.calculateAverageFitness()
71          self.removeWeakSpecies()
72
73          self.totalFitness = self.totalAverageFitness()
74
75          children = []
76
77          for species in self.species:
78              toBreed = (self.config["pool"]["population"] *
                    species.averageFitness) // self.totalFitness
79              toBreed -= 1
80
81              children.append(self.generateOffspring(species))
82
83          self.cullSpecies(cutToOne=True)
84
85          while (len(children) + len(self.species) <
                self.config["pool"]["population"]):
86              species = random.choice(self.species)
87              children.append(self.generateOffspring(species))
88
89          for child in children: self.addToSpecies(child)
90
91          self.generation += 1
92
93      def generateOffspring(self, species):
94          if random.random() < self.config["rates"]["crossover"]:
95              if (random.random() < self.config["rates"]["interspecies"]):
96                  species2 = random.choice(self.species)
97                  genome = random.choice(species2.genomes)
98                  child = genome.crossover(random.choice(species.genomes))
99                  child.mutate()
100                 return child
101             elif len(species.genomes) >= 2:
102                 g1, g2 = random.sample(species.genomes, 2)
103                 child = g1.crossover(g2)
104                 child.mutate()
105                 return child
106
107         child = random.choice(species.genomes).copy()
108         child.mutate()
109         return child
110
111
112     def cullSpecies(self, cutToOne = False):
```

```
113         for species in self.species:
114             species.sortGenomes()
115 
116             remaining = (len(species.genomes) // 2) + 1
117 
118             if (cutToOne):
119                 remaining = 1
120 
121             species.genomes = species.genomes[:remaining]
122 
123     def removeWeakSpecies(self):
124         survived = []
125 
126         self.totalFitness = self.totalAverageFitness()
127 
128         for species in self.species:
129             toBreed = (self.config["pool"]["population"] *
                    species.averageFitness) // self.totalFitness
130             if toBreed >= 1:
131                 survived.append(species)
132             else:
133                 print("Removing species %d (tB: %d, avF: %f, tF: %f)" %
                        (species.id, toBreed, species.averageFitness,
                        self.totalFitness))
134 
135         self.species = survived
136 
137     def removeStaleSpecies(self):
138         survived = []
139         for species in self.species:
140             species.sortGenomes()
141 
142             if (species.genomes[0].fitness > species.topFitness):
143                 species.topFitness = species.genomes[0].fitness
144                 species.staleness = 0
145             else:
146                 species.staleness += 1
147 
148             if (species.staleness <
                    self.config["speciation"]["stagnation_threshold"]) or
                    (species.topFitness >= self.maxFitness):
149                 survived.append(species)
150             else:
151                 print("Removing stale species %d." % (species.id))
152 
153         self.species = survived
154 
155     def saveGenome(self, species, n, genome):
156         dataFile = open(os.path.join(self.rtdir, "gen-%d" % self.generation,
                "spc-%d-gnm-%d" % (species.id, n)), "w")
157         data = {"connections": sorted([(conn.connectionId, conn.source.nodeId,
                conn.sink.nodeId, conn.weight) for conn in genome.connections.values()
158                                 if conn.enabled], key=lambda x: x[0]),
159         "nodes": sorted([(node.nodeId, node.nodeType.value) for node in
                genome.nodes.values()]),
160         "maxFitness": self.maxFitness,
```

```
        "generation": self.generation,
        "genomeId": n,
        "staleness": species.staleness,
        "speciesId": species.id}

        dataFile.write(json.dumps(data))
        dataFile.close()

    def runOnce(self, inputsFn, sendGenome, sendOutput):
        pprint.pprint([(sp.id, len(sp.genomes)) for sp in self.species])
        print("Running generation %d" % (self.generation))
        os.mkdir(os.path.join(self.rtdir, "gen-%d" % self.generation))
        for species in self.species:
            for n, genome in enumerate(species.genomes):
                genome.id = n
                print("Gen %d S %d G %d" % (self.generation, species.id, n+1))
                network = Network(genome)
                fitness = 0
                sendGenome(self, species, genome, n)
                nInputs = 0
                for input in inputsFn():
                    nInputs += 1
                    output = network.runWith(input)
                    sendOutput(output)
                    try:
                        fitness = self.fitnessFn(input, output)
                    except:
                        traceback.print_exc()
                        pprint.pprint(genome.nodes)
                        pprint.pprint(genome.connections)
                        pdb.set_trace()
                        sys.exit(1)
                genome.fitness = fitness

                self.saveGenome(species, n, genome)

                self.evaluatedGenomes += 1

        self.maxPrevFitness = self.maxFitness
        self.maxPrevGenome = self.maxGenome
        self.maxGenome = max(((genome,species) for species in self.species for
            genome in species.genomes), key=lambda g: g[0].fitness)
        self.maxFitness = self.maxGenome[0].fitness
        self.maxSpecies = self.maxGenome[1]
        self.maxGenome = self.maxGenome[0]
```

loader.py

```
# This file has proved instrumental a few times when we have had to reload
# generations from disk because of errors made both in the Python and Lua
# scripts, but was really fairly simple to write.

# in fact, it was done during a particularly boring chemistry lesson
# on a scrap piece of paper.

```

```
from .genome import NodeGene, ConnectionGene, Genome
from .pool import NetworkPool
from .nodetype import NodeType

import json, os

class Loader():
    def __init__(self, rundir, gen):
        self.gendir = os.path.join(rundir, "gen-%s" % gen)

    def getGenomes(self, fitnessFunction):
        pool = NetworkPool()

        for genome in os.listdir(self.gendir):
            data = json.load(open(os.path.join(self.gendir, genome)))
            g = Genome()

            for node in data["nodes"]:
                g.addNode(node[0], node[1])

            for connection in data["connections"]:
                g.addConnection(g.node(connection[1]), g.node(connection[2]),
                    connection[-1], connection[0])

            pool.maxFitness = data["maxFitness"]
            pool.generation = data["generation"]
            pool.addToSpecies(g, speciesId=data["speciesId"],
                staleness=data["staleness"])

        return pool
```

configuration.py

```
import toml
config = None

def getGlobalConfig():
    return config

def loadConfig(filename):
    global config
    config = toml.load(filename)
    return config
```

nodetype.py

```
import enum

class NodeType(enum.Enum):
   SENSOR = 1
   BIAS = 2
   HIDDEN = 3
   OUTPUT = 4
```

mario.py

```
from neat.pool import NetworkPool
from neat.genome import Genome, NodeGene, ConnectionGene, modSigmoid
from neat.nodetype import NodeType
from neat.loader import Loader

from neat import configuration

import json
import pprint
import os
import sys

configuration.loadConfig("Mario.toml")

initialGenome = Genome([*(NodeGene(NodeType.SENSOR, i+1) for i in range(240)),
    NodeGene(NodeType.BIAS, 241), *(NodeGene(NodeType.OUTPUT, 242+i) for i in
    range(4))])

class FR:
    def __init__(self, f):
        self.f = f

    def readline(self):
        i = None
        while not i:
            i = self.f.readline()
        return i

    def write(self, data):
        self.f.write(data)
        self.f.flush()

inFifo = FR(open("ltop", "w+"))
outFifo = FR(open("ptol", "w+"))

def fitnessFunction(input, output):
    return json.loads(inFifo.readline())["fitness"]

class EndRun(RuntimeError):
    pass

def getInput():
    while 1:
        ip = json.loads(inFifo.readline())
        if (ip.get("noInput", False)):
            raise StopIteration
        if (ip.get("complete", False)):
            raise EndRun
        yield ip["input"]

def sendOutput(output):
    data = json.dumps({"output": [*output[:2], 0, 0, *output[2:]]}) + '\n'
    outFifo.write(data)

```

```
def sendGenome(pool, species, genome, nthGenome):
    data = {"connections": sorted([(conn.source.nodeId, conn.sink.nodeId,
        conn.weight) for conn in genome.connections.values()
                                   if conn.enabled], key=lambda x: x[0]),
            "nodes": sorted([node.nodeId for node in genome.nodes.values()]),
            "maxFitness": pool.maxFitness,
            "generation": pool.generation,
            "genpercent": str(100*(nthGenome + 1) / len(species.genomes))[:4],
            "genomeId": nthGenome + 1,
            "speciesId": species.id}
    asJson = json.dumps(data)
    outFifo.write((asJson + '\n'))

def sendMaxFitness(pool):
    data = json.dumps({"maxFitness": pool.maxFitness}) + '\n'
    outFifo.write(data)

if len(sys.argv) > 1:
    loadPath = sys.argv[1]
    genNumber = sys.argv[2]
    pool = Loader(loadPath, genNumber).getGenomes(fitnessFunction)
    pool.fitnessFn = fitnessFunction
else:
    pool = NetworkPool(initialGenome, fitnessFunction)

try:
    while 1:
        pool.runOnce(getInput, sendGenome, sendOutput)
        pool.nextGeneration()
        print("Max fitness in generation %d (S %d G %d): %.20f | Staleness of
            species %d: %d" % (pool.generation, pool.maxSpecies.id,
            pool.maxGenome.id, pool.maxFitness, pool.maxSpecies.id,
            pool.maxSpecies.staleness))
except EndRun:
    print("Run ends!")
```

Mario3LuaScript.lua

```
json = require "json"



-- GET FROM FIFO
ptol = io.open("ptol", "r")
ltop = io.open("ltop", "w")

function readLine()
    local line = ptol:read("*l")
    while line == nil do
        line = ptol:read("*l")
    end
    return json.decode(line)
end

function writeLine(data)
```

```
    ltop:write(json.encode(data) .. "\n")
    ltop:flush()
end

completion = false

-- BLOCK 2: INPUTS
while not completion do
    savestate.loadslot(1)
    -- BLOCK 1: VARIABLE INITIATION

    -- Initiation of the input array (tiles)
    tiles = {}
    for i = 1, 240, 1 do
        tiles[#tiles+1] = 0
    end

    -- Initiation array enemy for-loops
    enemyAlive = {}
    enemyVertical = {}
    enemyHorizontal = {}
    EnableEnemyRender = {}

    -- Variables to retain previous frame data
    fireballPrevious1 = 0
    fireballPrevious2 = 0
    fireballSecondPrevious1 = 0
    fireballSecondPrevious2 = 0
    boomarangPrevious1 = 0
    boomarangPrevious2 = 0
    boomarangSecondPrevious1 = 0
    boomarangSecondPrevious2 = 0

    -- Fitness checking
    fitnessPrevious = 0
    fitnessTimer = 0
    fitnessTimer2 = 0

    -- Finish block Y position
    memory.writebyte(0x00A9, 128);

    -- Controller data
    controller = {}
    outputs = {}
    outputsAbsolute = {}
    buttonNames = {
        "A",
        "B",
        "Up",
        "Down",
        "Left",
        "Right"
    }
```

```
74      data = readLine()
75      generation = data["generation"]
76      species = data["speciesId"]
77      GenProgress = data["genpercent"]
78      maxFitness = data["maxFitness"]
79      genomeNumber = data["genomeId"]
80      nodes = data["nodes"]
81      connections = data["connections"]
82
83      fitnessHighest = 0
84      oldTime = 0
85      time = 0
86      fitnessVertical = 0
87      removePlant = 0
88
89
90      while true do
91
92              -- Temporary, development only
93              -- memory.writebyte(0x000ED, 05) -- Infinite Tanooki power-up
94              -- memory.writebyte(0x05F1, 1) -- Disable timer
95
96
97          -- Draw Input Box - empty
98          gui.drawBox(3, 11, 16*4 + 4, 15*4 + 12, 0xff000000 ,0x40000000)
99          gui.drawLine(15, 12, 15, 71, 0xff888888)
100         gui.drawLine(56, 12, 56, 71, 0xff888888)
101
102         -- Refresh inputs
103         for c = 1, 16*16, 1 do
104             tiles[c] = 0
105         end
106
107         -- Get Mario Coordinates
108         marioX = memory.readbyte(0x00090) + memory.readbyte(0x00075) * 256 + 8
109         if memory.readbyte(0x00087) ~= 255 then
110             marioY = memory.readbyte(0x000A2) + memory.readbyte(0x00087) * 256
111         else
112             -- (The vertical Low byte is equal to 255 on the topmost row of the
                    map)
113             marioY = 0
114         end
115
116         -- Check all tiles around Mario for hitboxes and update inputs (hitboxes)
117         for i = 1, 241, 1 do
118             tileX = marioX - 128 + (16*(i-1))%256
119             tileY = marioY - 128 + 16*math.floor((i-1)/16)
120
121             tileHorizontalLow = math.floor(tileX/256)
122             tileHorizontalHigh = math.floor((tileX%256)/16)
123             if tileY >= 0 then
124                 tileVertical = math.floor(tileY/16)
125             else
126                 -- Dit is voor dezelfde redenen als bij marioY hierboven
127                 tileVertical = 0
128             end
```

```
129             tileData = memory.readbyte(0x6000 + tileHorizontalLow*27*16 +
                    tileHorizontalHigh + tileVertical*16)
130             if tileData == 44 or tileData == 83 or tileData == 85 or tileData ==
                    87 or tileData == 95 or tileData == 97 or tileData == 99 or
                    tileData == 103 or
131                 tileData == 110 or tileData == 112 or tileData == 121 or tileData
                        == 173 or tileData == 174 or tileData == 177 or
132                 tileData == 178 or tileData == 186 or tileData == 187 or tileData
                        == 37 or tileData == 38 or tileData == 39 or tileData == 80 or
133                 tileData == 81 or tileData == 82 or tileData == 160 or tileData
                        == 161 or tileData == 162 or tileData == 226 or tileData ==
                        227 or
134                 tileData == 228 or tileData == 46 or tileData == 49 or tileData
                        == 105 or tileData == 107 or tileData == 109 or tileData ==
                        113 or tileData == 114 or
135                 tileData == 116 or tileData == 154 or tileData == 155 or tileData
                        == 156 or tileData == 157 or tileData == 158 or tileData ==
                        167 or tileData == 168 or
136                 tileData == 169 or tileData == 170 or tileData == 171 or tileData
                        == 182 or tileData == 183 or tileData == 184 or tileData == 185
137             then
138                 tiles[i] = 1
139             end
140         end
141
142         -- Update frame-dependent Variables
143         screenHorizontalPosition = memory.readbyte(0x00FD) +
                memory.readbyte(0x0012)*256
144
145         -- Get enemy data
146         for i = 1, 5, 1 do
147             enemyAlive[i] = memory.readbyte(0x665 - i + 1)
148
149             enemyX = screenHorizontalPosition + memory.readbyte(176 - i + 1) + 8
150             enemyHorizontal[i] = math.floor((enemyX - marioX) / 16) - 7
151
152             enemyY = memory.readbyte(167 - i + 1) + 256
153             enemyVertical[i] = math.floor(enemyY / 16) + 9 - math.floor(marioY/16)
154         end
155         -- Circumvent enemy position overflow - No enemies render outside the two
                lines
156         for i = 1, 5, 1 do
157             if enemyHorizontal[i] < -12 or enemyHorizontal[i] > -3 then
158                 EnableEnemyRender[i] = 0
159             else
160                 EnableEnemyRender[i] = 1
161             end
162         end
163         -- Update inputs (enemies)
164         for i = 1, 5, 1 do
165             if EnableEnemyRender[i] == 1 then
166                 if enemyAlive[i] == 2 or enemyAlive[i] == 5 then
167                         tiles[(enemyVertical[i])*16 + enemyHorizontal[i]] = -1
168                 end
169             end
170         end
```

```
171
172         -- Get Projectile data (fireball)
173         fireballHorizontal1 =
                math.floor((memory.readbyte(0x05CB)+256-marioX)/16)%16 - 6
174         fireballHorizontal2 =
                math.floor((memory.readbyte(0x05CC)+256-marioX)/16)%16 - 6
175         fireballVertical1 = math.floor((memory.readbyte(0x05C1) + 256 -
                marioY)/16) + 9
176         fireballVertical2 = math.floor((memory.readbyte(0x05C2) + 256 -
                marioY)/16) + 9
177
178         fireballDirection1 = fireballSecondPrevious1 - memory.readbyte(0x05CB)
179         fireballSecondPrevious1 = fireballPrevious1
180         fireballPrevious1 = memory.readbyte(0x05CB)
181
182         fireballDirection2 = fireballSecondPrevious2 - memory.readbyte(0x05CC)
183         fireballSecondPrevious2 = fireballPrevious2
184         fireballPrevious2 = memory.readbyte(0x05CC)
185
186         -- Get Projectile data (boomarang)
187         boomarangHorizontal1 =
                math.floor((memory.readbyte(0x05CD)+256-marioX)/16)%16 - 6
188         boomarangHorizontal2 =
                math.floor((memory.readbyte(0x05CE)+256-marioX)/16)%16 - 6
189         boomarangVertical1 = math.floor((memory.readbyte(0x05C3) + 256 -
                marioY)/16) + 9
190         boomarangVertical2 = math.floor((memory.readbyte(0x05C4) + 256 -
                marioY)/16) + 9
191
192         boomarangDirection1 = boomarangSecondPrevious1 - memory.readbyte(0x05CD)
193         boomarangSecondPrevious1 = boomarangPrevious1
194         boomarangPrevious1 = memory.readbyte(0x05CD)
195
196         boomarangDirection2 = boomarangSecondPrevious2 - memory.readbyte(0x05CE)
197         boomarangSecondPrevious2 = boomarangPrevious2
198         boomarangPrevious2 = memory.readbyte(0x05CE)
199
200         -- Update inputs (projectiles)
201         if fireballDirection1 ~= 0 then
202             tiles[fireballVertical1*16 + fireballHorizontal1] = -1
203         end
204         if fireballDirection2 ~= 0 then
205         tiles[fireballVertical2*16 + fireballHorizontal2] = -1
206         end
207         if boomarangDirection1 ~= 0 then
208             tiles[boomarangVertical1*16 + boomarangHorizontal1] = -1
209         end
210         if boomarangDirection2 ~= 0 then
211         tiles[boomarangVertical2*16 + boomarangHorizontal2] = -1
212         end
213
214
215         -- Draw inputs
216         for layer = 1, 15, 1 do
217             for i = 1, 16, 1 do
218                 sqX = 4 + (i-1) * 4
```

```
219             sqY = 12 + (layer-1) * 4
220             -- Enemies and projectiles
221             if tiles[(layer-1)*16+i] == -1 then
222                 gui.drawBox(sqX, sqY, sqX + 3, sqY + 3, 0xFF000000, 0xFF000000)
223             else
224             -- Hitboxes
225             if tiles[(layer-1)*16+i] == 1 then
226                 gui.drawBox(sqX, sqY, sqX + 3, sqY + 3, 0xffffffff, 0xffffffff)
227             else
228             -- Empty space
229                 gui.drawBox(sqX, sqY, sqX + 3, sqY + 3, 0x000000, 0x000000)
230             end
231             end
232         end
233     end
234
235     -- Draw position Mario
236     gui.drawBox(37, 45, 38, 51, 0xFF740858, 0xFF740858)
237     gui.drawLine(34, 51, 41, 51, 0xFF740858)
238
239
240  -- BLOCK 3: GENOME DATA AND FITNESS
241
242    --Remove plant
243    if  memory.readbyte(0x00075) == 1 and removePlant == 0 then
244       memory.writebyte(0x665,0)
245       memory.writebyte(0x664,0)
246       removePlant = 1
247    end
248    if memory.readbyte(0x0075) == 6 or memory.readbyte(0x0075) == 7 then
249       memory.writebyte(0x664,0)
250    end
251
252
253     oldTime = time
254     time = memory.readbyte(0x05EE)*100 + memory.readbyte(0x05EF)*10 +
            memory.readbyte(0x05F0)-300
255     score = memory.readbyte(0x0717)*10 + memory.readbyte(0x0716)*2560
256
257     if fitnessVertical == 0 and marioY <= 340 then
258      fitnessVertical = 1
259    end
260
261     fitness = marioX + time + fitnessVertical*250
262
263 if marioY >= 390 then
264    stuck = true
265 end
266
267 if (memory.readbyte(0x0303) == 3 or memory.readbyte(0x0303) == 17) and
        memory.readbyte(0x00ED) ~= 1 then
268    stuck = true
269 end
270
271 if (fitness > fitnessHighest) then
272    fitnessHighest = fitness
```

```
273   end
274
275   if (fitnessHighest > fitness) then
276      fitnessTimer2 = fitnessTimer2 + 1
277   end
278
279   if (fitnessTimer2 > 6) then
280      fitnessTimer2 = 0
281      stuck = true
282   end
283
284        if oldTime > time then
285            fitnessTimer = fitnessTimer + 1;
286        elseif (fitnessPrevious - time) < (fitness - time) then
287            fitnessTimer = 0;
288        end
289        if fitnessTimer > 3 then
290            stuck = true
291        end
292
293   if stuck == true then
294               writeLine({["noInput"] = 1})
295      stuck = false
296               break
297   end
298
299        fitnessPrevious = fitness;
300
301
302        if marioX > screenHorizontalPosition + 192 then
303            completion = true
304            writeLine({["complete"] = 1})
305        else
306            completion = false
307        end
308
309    -- SEND TO FIFO
310        writeLine({["input"] = tiles})
311
312        data = readLine()
313        outputs = data["output"]
314
315        writeLine({["fitness"] = fitness})
316
317
318        for i = 1, 6 do
319            if outputs[i] > 0.95 then
320                outputsAbsolute[i] = true
321            end
322            if outputs[i] < 0.95 then
323                outputsAbsolute[i] = false
324            end
325        end
326
327
328        gui.drawBox(0,210,255,255, 0x8fffffff, 0xDfffffff)
```

```
329
330
331         gui.drawText(10, 220, "Gen: " .. generation, 0xff000000, 0x00000000, 10)
332         gui.drawText(65, 220, "Species: " .. species, 0xff000000, 0x00000000, 10)
333         gui.drawText(140, 220, "Genome: " .. genomeNumber, 0xff000000,
                0x00000000, 10)
334         gui.drawText(210, 220, "(" .. GenProgress .. "%)", 0xff000000,
                0x00000000, 10)
335
336         gui.drawText(10, 210, "Fitness: " .. fitness, 0xff000000, 0x00000000, 10)
337         gui.drawText(100, 210, "Max Fitness: " .. maxFitness, 0xff000000,
                0x00000000, 10)
338
339
340
341     -- BLOCK 4: OUTPUTS AND NEURAL NETWORK VISUALISATION
342
343         -- Refresh Outputs
344         for i = 1, 6 do
345             controller["P1 " .. buttonNames[i]] = false
346         end
347
348         for i = 1, 6 do
349             controller["P1 " .. buttonNames[i]] = outputsAbsolute[i]
350         end
351         joypad.set(controller)
352
353         emu.frameadvance();
354     end
355 end
```

NetworkVisualisation.lua

```
1  nodeX = {}
2  nodeY = {}
3  nodeDrawn = {}
4  nodeOutput = {}
5  nodeID = {}
6
7  while true do
8     for i = 1, #nodeX do
9        nodeX[i] = 0
10       nodeY[i] = 0
11       nodeDrawn[i] = 0
12       nodeOutput[i] = 0
13    end
14
15
16
17
18    -- Draw Buttons
19    for i = 1, 6 do
20       if i == 1 or i == 2 then
21          if outputsAbsolute[i] == false then
```

```
22            gui.drawText(220, 4 + 8*i, buttonNames[i], 0xff000000, 0x00000000, 10
                  )
23            gui.drawBox(215, 8 + 8*i, 218, 11 + 8*i, 0xff888888, 0x40000000)
24         end
25         if outputsAbsolute[i] == true then
26            gui.drawText(220, 4 + 8*i, buttonNames[i], 0xff276d1c, 0x00000000, 10
                  )
27            gui.drawBox(215, 8 + 8*i, 218, 11 + 8*i, 0xffffffff, 0xffffffff)
28         end
29      end
30      if i == 3 or i == 4 then
31         if outputsAbsolute[i+2] == false then
32            gui.drawText(220, 4 + 8*i, buttonNames[i+2], 0xff000000, 0x00000000,
                  10 )
33            gui.drawBox(215, 8 + 8*i, 218, 11 + 8*i, 0xff888888, 0x40000000)
34         end
35         if outputsAbsolute[i+2] == true then
36            gui.drawText(220, 4 + 8*i, buttonNames[i+2], 0xff276d1c, 0x00000000,
                  10 )
37            gui.drawBox(215, 8 + 8*i, 218, 11 + 8*i, 0xffffffff, 0xffffffff)
38         end
39      end
40   end
41
42
43
44   for i = 1, #nodes do
45      nodeID[i] = nodes[i]
46   end
47
48   for i = 1, 245, 1 do
49      if i <= 240 then
50         nodeX[i] = 4 + 4 * ((i-1)%16)
51         nodeY[i] = 12 + 4 * (math.floor((i-1)/16))
52      end
53      if i == 241 then
54         nodeX[i] = 16*4
55         nodeY[i] = 79
56      end
57      if i >= 242 and i <= 247 then
58         nodeX[i] = 215
59         nodeY[i] = 8 + 8*(i-241)
60      end
61   end
62
63   for i = 1, 245, 1 do
64      nodeDrawn[i] = true
65   end
66
67   for i = 246, #nodes do
68      nodeDrawn[i] = false
69   end
70
71   for i = 246, #nodes do
72
73      if nodeDrawn[i] == false then
```

```
74          nodeX[i] = 85 + math.floor((i-245)/5)*20 + (i%2)*6 + (i-245)*3
75          nodeY[i] = 2 + (((i-245)%5)+1)*10
76          nodeDrawn[i] = true
77       end
78    end
79
80    for i = 1, 240 do
81       if tiles[i] == 1 then
82          nodeOutput[i] = 1
83       end
84       if tiles[i] == -1 then
85          nodeOutput[i] = -1
86       end
87       if tiles[i] == 0 then
88          nodeOutput[i] = 0
89       end
90    end
91    for i = 242, 245 do
92       if outputsAbsolute[i-241] then
93          nodeOutput[i] = 1
94       else
95          nodeOutput[i] = 0
96       end
97    end
98
99
100   nodeOutput[241] = 1
101
102
103   for i = 246, #nodes do
104      nodeOutput[i] = 0
105   end
106
107
108
109   for i = 1, #connections do
110      source = connections[i][1]
111      sink = connections[i][2]
112      weight = connections[i][3]
113
114      for i = 1, #nodes do
115         if nodeID[i] == source then
116            sourceNumber = i
117         end
118         if nodeID[i] == sink then
119            sinkNumber = i
120         end
121      end
122
123
124      if nodeOutput[sourceNumber] == 1 then
125         nodeOutput[sinkNumber] = 1
126      end
127      if nodeOutput[sourceNumber] == 0 then
128         nodeOutput[sinkNumber] = 0
129      end
```

```
130       if nodeOutput[sourceNumber] == -1 then
131          nodeOutput[sinkNumber] = -1
132       end
133
134       if nodeDrawn[sourceNumber] and nodeDrawn[sinkNumber] then
135          if nodeOutput[sourceNumber] == 0 then
136             if weight >= 0 then
137                gui.drawLine(nodeX[sourceNumber]+1, nodeY[sourceNumber]+1,
                       nodeX[sinkNumber]+1, nodeY[sinkNumber]+1, 0x4F00ff00)
138             end
139             if weight < 0 then
140                gui.drawLine(nodeX[sourceNumber]+1, nodeY[sourceNumber]+1,
                       nodeX[sinkNumber]+1, nodeY[sinkNumber]+1, 0x4Fff0000)
141             end
142          else
143             if weight >= 0 then
144                gui.drawLine(nodeX[sourceNumber]+1, nodeY[sourceNumber]+1,
                       nodeX[sinkNumber]+1, nodeY[sinkNumber]+1, 0xFF00ff00)
145             end
146             if weight < 0 then
147                gui.drawLine(nodeX[sourceNumber]+1, nodeY[sourceNumber]+1,
                       nodeX[sinkNumber]+1, nodeY[sinkNumber]+1, 0xFFff0000)
148             end
149          end
150       end
151    end
152
153
154    gui.drawBox(nodeX[241], nodeY[241], nodeX[241]+3, nodeY[241]+3, 0xffffffff,
           0xffffffff)
155
156    for i = 246, #nodes do
157       if nodeDrawn[i] == true then
158          if nodeOutput[i] == 0 then
159             gui.drawBox(nodeX[i], nodeY[i], nodeX[i]+3, nodeY[i]+3, 0xff888888,
                    0x40000000)
160          end
161          if nodeOutput[i] == 1 then
162             gui.drawBox(nodeX[i], nodeY[i], nodeX[i]+3, nodeY[i]+3, 0xffffffff,
                    0xffffffff)
163          end
164          if nodeOutput[i] == -1 then
165             gui.drawBox(nodeX[i], nodeY[i], nodeX[i]+3, nodeY[i]+3, 0xff000000,
                    0xff000000)
166          end
167       end
168    end
169
170    emu.frameadvance();
171 end
```