|  |
| --- |
| Swansea Metropolitain University |
| Final Project |
| Automated Agent Control Using Non-deterministic Artificial Intelligence |
|  |
| **Sion Williams** |
| **2008/2009** |

|  |
| --- |
| Project submitted as part of the requirements for the award B.Sc. (Hons) Computer Games Development |

## Declaration

I, ............................................(FULL NAME)………………………………… declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This has not previously been accepted as part of any other degree submission.

***Signed : .............................................***

***Date : .............................................***

## Form Of Consent

**I** ............................................ hereby consent that my Project, submitted in candidature for the B.Sc.(Hons) Computer Games Development degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

***Signed : .............................................***

***Date : .............................................***

## Acknowledgements

Dr Glenn L. Jenkins, supervisor and lecturer, for all his help and guidance throughout the project. So much has been learnt from the experience and it has produced a great enthusiasm for computer games that will hopefully carry on in the future.

I would also like to thank my family, friends and girlfriend for their support throughout the project. Thank you for your endless cups of coffee.

## Abstract

Neural networks are a type of non-deterministic artificial intelligence that has the ability to control. Understanding the mechanics behind a neural network is critical in determining the feasibility of such an approach. For this purpose a multi layered feed forward neural network has been developed and trained using a back propagation method. Results of an experimental study into the ability of a neural network to learn and generalize a problem are presented.

## Glossary of terms and abbreviations

A.I – Artificial intelligence

CPU – Central Processing Unit

Deterministic - Perfect knowledge of the next state

Non- deterministic – Random, unpredictable

Epoch – A training epoch is the duration to complete a full set of training data.

G.A – Genetic Algorithm

NN – Neural Network

PPU - Physics processing units

Table of Contents

[Declaration 2](#_Toc229497747)

[Form Of Consent 3](#_Toc229497748)

[Acknowledgements 4](#_Toc229497749)

[Abstract 5](#_Toc229497750)

[Glossary of terms and abbreviations 6](#_Toc229497751)

[Table of Tables 10](#_Toc229497752)

[Table Of Figures 12](#_Toc229497753)

[1. Introduction 13](#_Toc229497754)

[2. Literature Review 15](#_Toc229497755)

[2.1. Bayes Theorem 15](#_Toc229497756)

[2.2. Bayesian Network 16](#_Toc229497757)

[2.3. Genetic Algorithms 16](#_Toc229497758)

[2.4. Neural Networks 18](#_Toc229497759)

[2.4.1. Biological Neural Network 18](#_Toc229497760)

[2.4.2. Artificial Neuron Model 19](#_Toc229497761)

[2.4.2.1. Weights and Bias 20](#_Toc229497762)

[2.4.2.2. Summation Function 20](#_Toc229497763)

[2.4.2.3. Transfer/Activation Function 22](#_Toc229497764)

[2.4.3. Multi Layered Perceptron 24](#_Toc229497765)

[2.4.4. Unsupervised Learning 25](#_Toc229497766)

[2.4.5. Supervised Learning 25](#_Toc229497767)

[2.4.6. Mean Squared Error 26](#_Toc229497768)

[2.4.6.1. Back propagation Learning 26](#_Toc229497769)

[2.5. Learning Rate 27](#_Toc229497770)

[2.6. Converging 27](#_Toc229497771)

[2.7. Auto-associative and Hetero-associative 28](#_Toc229497772)

[2.8. Conclusion 28](#_Toc229497773)

[3. Methodology 30](#_Toc229497774)

[3.1. Introduction 30](#_Toc229497775)

[3.2. Research 30](#_Toc229497776)

[3.3. Neural Network Development 31](#_Toc229497777)

[3.3.1. Extreme Programming 32](#_Toc229497778)

[3.3.2. Unified Modelling Language Design 33](#_Toc229497779)

[3.4. Determining The Number of Hidden Layers and Hidden Nodes 34](#_Toc229497780)

[3.5. Dataset Generation 35](#_Toc229497781)

[3.6. Training 41](#_Toc229497782)

[3.7. Procedure 46](#_Toc229497783)

[3.8. Testing 47](#_Toc229497784)

[3.9. Analysis 48](#_Toc229497785)

[4. Results 49](#_Toc229497786)

[5. Discussion 59](#_Toc229497787)

[6. Future Improvements 62](#_Toc229497788)

[7. Bibliography 63](#_Toc229497789)

[Appendix A – Class Diagram 67](#_Toc229497790)

[Appendix B – Program Code 68](#_Toc229497791)

# Table of Tables

[Table 1 40](#_Toc229498008)

[Table 2 40](#_Toc229498009)

[Table 3 47](#_Toc229498010)

[Table 4 Test 1 Expected result 0.15 49](#_Toc229498011)

[Table 5 test 2 Expected result 0.25 49](#_Toc229498012)

[Table 6 Test 3 Expected result 0.35 49](#_Toc229498013)

[Table 7 Test 4 Expected result 0.45 50](#_Toc229498014)

[Table 8 Test 5 Expected Result 0.65 50](#_Toc229498015)

[Table 9 Test 6 Expected result 0.75 50](#_Toc229498016)

[Table 10 Test 7 Expected Result 0.85 50](#_Toc229498017)

[Table 11 Test 8 Expected Result 0.95 51](#_Toc229498018)

[Table 12 Test 1 Expected Result 0.15 52](#_Toc229498019)

[Table 13 Test 2 Expected Result 0.25 52](#_Toc229498020)

[Table 14 Test 3 Expected Result 0.35 52](#_Toc229498021)

[Table 15 Test 4 Expected Result 0.45 52](#_Toc229498022)

[Table 16 test 5 Expected Result 0.65 52](#_Toc229498023)

[Table 17 Test 6 Expected Result 0.75 53](#_Toc229498024)

[Table 18 Test 7 Expected Result 0.85 53](#_Toc229498025)

[Table 19 Test 8 Expected Result 0.95 53](#_Toc229498026)

[Table 20 54](#_Toc229498027)

[Table 21 54](#_Toc229498028)

[Table 22 55](#_Toc229498029)

[Table 23 55](#_Toc229498030)

[Table 24 56](#_Toc229498031)

[Table 25 56](#_Toc229498032)

[Table 26 57](#_Toc229498033)

[Table 27 3 Hidden Node Win Table 58](#_Toc229498034)

[Table 28 6 Hidden Node Win Table 58](#_Toc229498035)

[Table 29 Network Id's 59](#_Toc229498036)

[Table 30 Comparison of Networks 20 and 22 61](#_Toc229498037)

# Table Of Figures

[Figure 1 Crossover 17](#_Toc229498038)

[Figure 2 Random Mutations 17](#_Toc229498039)

[Figure 3 Biological neuron 19](#_Toc229498040)

[Figure 4 Single Input Neuron 20](#_Toc229498041)

[Figure 5 Multiple Input Neuron 21](#_Toc229498042)

[Figure 6 Hard-Limit Function 22](#_Toc229498043)

[Figure 7 Linear Function 23](#_Toc229498044)

[Figure 8 Log-Sigmoid Function 23](#_Toc229498045)

[Figure 9 Multi Layered Perceptron 25](#_Toc229498046)

[Figure 10 Global And Local Minimum 28](#_Toc229498047)

[Figure 11 Example of Feeler LOS 30](#_Toc229498048)

[Figure 12 3 Hidden Nodes/ 1 Hidden Layer 35](#_Toc229498049)

[Figure 13 6 Hidden nodes/ 1 Hidden Layer 35](#_Toc229498050)

[Figure 14 Intersecting The Segment AB Against A Plane 36](#_Toc229498051)

[Figure 15 Value of t 37](#_Toc229498052)

[Figure 16 Intersection Algorithm 38](#_Toc229498053)

[Figure 17 Linear Relationship Between Input and Output 39](#_Toc229498054)

[Figure 18 Neural Network Training Example 42](#_Toc229498055)

[Figure 20 Neural Update (Epoch's Completed) time speedup 62](#_Toc229498056)

## Introduction

Artificial intelligence techniques can be classified in two ways: deterministic and non-deterministic . Deterministic behaviour is specified and predictable but is fast and easy to implement, test, understand and debug. Non-deterministic behaviour on the other hand has a degree of uncertainty, and dependant on the programmer’s knowledge of the system can be unpredictable .

Non-deterministic methods can offer games unpredictable game play and the resource to learn, both of which can be desirable features of games. Yet, whilst non-deterministic methods offer developers the luxury of not having to explicitly code all behaviours in anticipation of all possible scenarios, developers remain wary of the unpredictable nature of non-deterministic methods.

One of the main challenges for A.I is to create intelligent agents that adapt, i.e. change their behaviour based on interactions with the environment and adapting to new situations as they occur. Research would suggest that non-deterministic A.I provides a solution to all of the above scenarios but have not advanced further than theoretical research. Game technology can provide a rich platform for validating and advancing this research [4].

To prove that a non-deterministic A.I approach could work effectively in a game, a literature review was conducted on current types of non-deterministic A.I. Once completed the findings of the research were evaluated and a type of non-deterministic A.I was selected for testing. Testing was done using an experimental design so that results could be analysed statistically. The long term aim of this project was to create an automated agent that was solely controlled using a non-deterministic A.I technique.

In Chapter 2, the literature review will study current non-deterministic models for implementing artificial intelligence. Chapter 3 will review the problem area, provide an overview of the approach to solve the problem, offer a description of the testing procedure and describe how the test data will be analysed. Chapter 4 will provide the results obtained from testing which will then progress to Chapter 5 which examines the test results. Chapter 6 will then conclude the project and discuss if this approach is suitable for use within computer games. Chapter 7 will then close the project offering suggestions that could possibly improve the use of non-deterministic artificial intelligence models.

## Literature Review

The previous section provided an overview of the area to be discussed throughout this study. This literature review will continue by looking at the academic research, in the form of articles, publications, interviews and journals surrounding current approaches to nondeterministic artificial intelligence.

Due to the abundance of information available, particularly related to artificial intelligence, material will be selective. The key area of interest for the literature review will be applications of non-deterministic A.I in games. The reason for excluding deterministic approaches is that they have already been well documented and tested and their inclusion would be too much for a single project.

### Bayes Theorem

Bayes Theorem, developed by the Rev. Thomas Bayes, an 18th century mathematician and theologian, was first published in 1763 . Mathematically it is expressed as:

* P(H|E) is the conditional probability of H, given E. It is also called the posterior probability as it’s derived from or dependent upon E.
* H is the hypothesis or theory of interest.
* E represents a new piece of evidence that either confirms or disconfirms the theory.
* P(H) is the prior probability or marginal probability of H.
* P(E|H) is the condition probability of E given H.
* P(E) is the prior or marginal probability of E, and acts as a normalizing or scaling factor.

The Bayes theorem offers a form of conditional probability by relating the conditional and marginal probabilities of events H and E, where E has a non-diminishing probability [5]. It is useful because there are many real world examples where the probability of one event is conditional on the probability of a previous one. For example the probability of grass being wet after rain is conditional on the basis that it rains in the first place.

### Bayesian Network

While the Bayes theorem can anticipate the factor of conditionality, in many cases calculations are non-deterministic polynomial-time hard [6]. This means although it may be possible to manage a scenario with 5 discrete random variables, monitoring a system with many more variables would be unmanageable [6]. This is where a Bayesian network is useful. A Bayesian network is a graphical model that encodes probabilistic relationships among variables of interest [7] and can be used to learn casual relationships, and hence can gain understanding about a problem domain making it a very useful control method in A.I.

### Genetic Algorithms

Genetic algorithms (GA) are search algorithms based on the mechanics of natural selection and genetics as observed in the biological world . Charles Darwin proposed the theory of evolution in his work titled “On the Origin of Species” in 1859. He proposed that those that are most able to survive in their environments are able to pass on their traits to the next generation. These individual traits are encoded in chromosomes. In the next generation, these chromosomes are combined in a process called crossover. Crossover is a recombination of the chromosomes in the offspring.

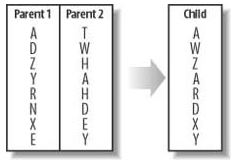


Figure Crossover

shows how each GA parent passes half of its genetic material on to the child. However, in the real world this crossover process might not be exact. Random mutations can also take place. Random mutations are nature's way of developing evolution. If a random mutation improves the species, it gets passed on to future generations. However, if not, the particular trait doesn't get passed on. shows how a random mutation may appear.

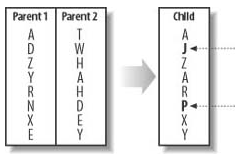


Figure Random Mutations

This constant recombination of chromosomes from the most successful members of the previous generation, combined with random mutations, creates future generations that are better adapted to survive and flourish in their environments.

GA use both “survival of the fittest” or crossover and randomisation to robustly explore a function. One example of their use in games is described by Nicholas Cole where they discuss their use for tuning parameters in a “Counterstrike” artificially controlled non-player character.

### Neural Networks

In 1943 Warren S. McCulloch and Walter Pitts released a paper called “A logical calculus of the ideas immanent in nervous activity”. This article introduced the first mathematical model of a biological neuron. Later, in 1958 the first practical application of the artificial neural network was presented by the AI community , the perceptron. The perceptron was created by Rosenblatt to model the human vision system [10].

Neural networks have since seen many applications from aerospace technology as high performance autopilots to truck brake diagnosis systems in the transportation industry [12]. Their uses are numerous but they have encountered many problems in video games. One of the biggest problems is the large number of different architectures and the time costly matter of debugging errors, research suggests that many developers still believe neural networks are too resource intensive [13].

### Biological Neural Network

Neural networks fall into two categories: artificial neural networks and biological neural networks . An artificial neural network is a computational structure inspired by the study of biological neural processing . The most familiar biological neural network is the brain. The brain contains approximately 1011 neurons with each neuron connecting to approximately 1000 other neurons .

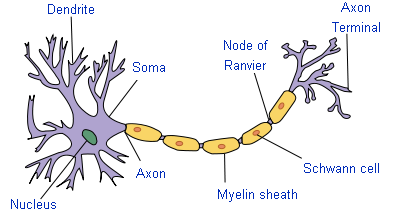


Figure Biological neuron

is a typical graphical representation of a biological neuron. The components of the neuron work as follows:

Dendrite(s) - responsible for collecting incoming signals.

Soma – responsible for the main processing and summation of signals.

Axon – responsible for transmitting signals to other dendrites.

### Artificial Neuron Model

Figure 4 shows an artificial single input neuron. In this figure the neuron has an input scalar *p,* a weight scalar *w* and a bias *b. p* and *w* are multiplied to form *wp* and are passed to the summation function depicted in the figure by sigma ∑. The other input 1 is multiplied by the bias *b* and then also passed to the summation function.

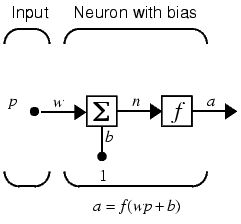


Figure Single Input Neuron

The output of the summation function *n* also known as the net input is now gets passed to the transfer or activation function *f*. The neuron output is calculated as:

The actual output depends entirely on the transfer function that is chosen. Here *f* the transfer function takes the argument *n* and produces the output *a*.

### Weights and Bias

Fausett defines weights as being a value associated with a connection path between two processing elements in a neural network. It is used to modify the strength of a transmitted signal in many networks. The weights contain fundamental information concerning the problem being solved by the net. In many nets the weights are modified during training using a learning algorithm.

The bias can be thought of as an additional weight that is always set to 1. To add additional learning power to a network the weight of the bias is also trained.

### Summation Function

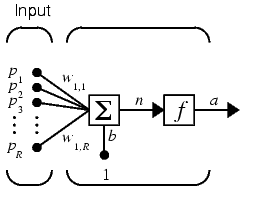


Figure Multiple Input Neuron

In a neural network a neuron can have any number of inputs . In the inputs are represented as:

P1,P2,P3…….PR

Their corresponding weights as:

W1,1,W1,2,W1,3……W1,R

The two are then multiplied together and passed to the summing function as:

W1,1P1+W1,2P2+W1,3+P3+……+W1,RPR

As the neuron also has a bias this is added in the summing function to produce:

W1,1P1+W1,2P2+W1,3+P3+……+W1,RPR+b

The following can be simplified to:

The equation shown above is the simplified version of the function commonly known as the summing or netting function. The product of this function is passed as an argument to the transfer function.

### Transfer/Activation Function

As explained previously the output of a neuron can be changed based on the transfer function. The transfer function can be a linear or non-linear function of *n* [16 pp. 2-3,2-6]*.* Some of the most commonly used activations functions are shown below.

*Hard-limit Transfer Function*

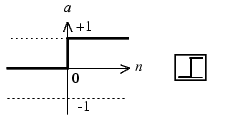


Figure Hard-Limit Function

The hard-limit function also known as step function shown in limits the output of the neuron to either 0, if the input argument *n* is less than 0, or 1 if *n* is greater than or equal to 0.

Single layer networks often use a hard-limit functions to convert the net input to an output unit that is a binary (1or 0) or bipolar (1 or -1) signal . This type of transfer function is often used in classification problems where there are two distinct categories .

*Linear Transfer Function*

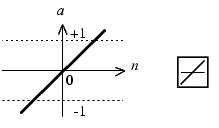


Figure Linear Function

For the input units this function is the identity function.

This type of function is typically used in the ADALINE network.

*Log-Sigmoid Transfer Function*

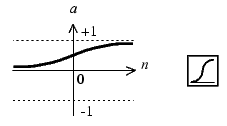


Figure Log-Sigmoid Function

This function is the most commonly used transfer function in feed forward neural networks that are trained using back propagation . The reason for this is that the simple relationship between the value of a function at a point and the value of the derivative at that point reduces the computational burden during training . The log-sigmoid function takes the input *n,* which can be between -∞ and +∞ and scales it to the range 0 to 1 [17].

𝜎 is the activation response. This increases or decreases how fast the neural network reaches its asymptotic value by changing the gradient of the function.

*Bipolar Log-Sigmoid Transfer Function*

The bipolar function is also used as an activation function for neural networks. It operates in the same way as the log-sigmoid but is scaled so that values are ranged between -1 and +1, the function is:

### Multi Layered Perceptron

The multi layered perceptron is a feed forward network . “Feed forward” means that the values only move from input to hidden to output layers; no values are fed back to earlier layers (a Recurrent Network allows values to be fed backward). In its simplest form a multi layer network will have an input layer (which takes information from a given example), a second or hidden layer (which receives its inputs from input/first layer), and an output layer (which takes outputs from the hidden layer to produce the networks overall results) this can be seen in greater detail in . The input layer is defined by the number of inputs provided by the example and the outputs are defined by the number values to be predicted. is a “fully connected” network meaning that the output from each input and hidden neuron is distributed to all of the neurons in the following layer.

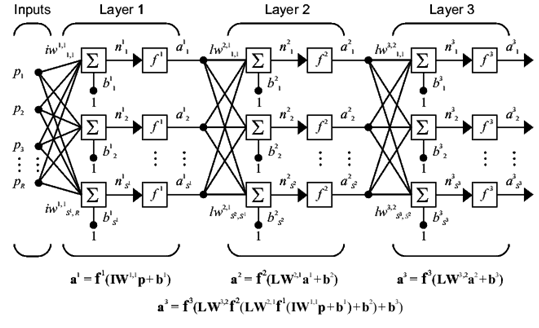


Figure Multi Layered Perceptron

### Unsupervised Learning

A means of modifying weights of a neural network without specifying the desired output for any input patterns. Unsupervised are also termed self-organizing. There is more interaction between neurons, typically with feedback and interlayer connections between neurons promoting self-organization.

### Supervised Learning

Each network undergoes training; the reason for this is to ensure that the correct response is obtained for a given input. Supervised learning is a learning paradigm for providing a network with a set of example data which provides input and expected outputs. These examples are known as training sets.

One form of supervised learning is known as the back propagation learning algorithm . The back propagation algorithm works by propagating the error back through the network, effectively changing the weights of the inputs and biases.

### Mean Squared Error

Sometimes used in place of ‘squared error’ or ‘total squared error’ in the derivation of the delta rule and back propagation training algorithms or in stopping conditions. Mean squared error is given by:

### Back propagation Learning

Back propagation learning implements a simple process of calculating an error in the output node and feeding the error back through the network

The algorithm begins with the assignment of randomly generated weights for the multi layer network. The following process is then repeated until the mean-squared error (MSE) of the output is sufficiently small:

Take a data set example *E* with the associated correct response *C.*

Compute the forward propagation of *E* through the network (compute the weighted sums of the network, and the activations, *ui*, for every node).

Starting at the output the error of a network is calculated using the following formulas:

For the output cell:

For all hidden cells:

*m* denotes all cells connected to the hidden node

*W* is the given weight vector

*u* is the activation

Finally, the weights within the network are updated as follows:

For weights connecting to the output:

For weights connecting to the hidden:

*p* represents the learning rate*.* This value is important as it limits the change that may occur during each step.

### Learning Rate

A parameter that controls the amount by which weights are changed during training. In some networks the learning rate may be constant (as in back propagation); in others it is reduced as training progresses to achieve stability ( for example, in Kohonen learning) [19].

### Converging

shows a simplified diagram of the search space of a neural network. An actual search space of a neural network would have closer to 200 dimensions. also shows how the neural network can converge to a minimum but this is not always the best solution, this scenario is known as converging to the local minimum.

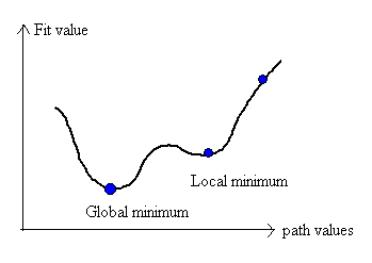


Figure Global And Local Minimum

### Auto-associative and Hetero-associative

Auto-association is the phenomenon of associating an input vector with itself as the output, whereas hetero-association is that of recalling a related vector given an input vector.

As described by Rao ; You have a fuzzy remembrance of a phone number. Luckily, you stored it in an auto-associative neural network. When you apply the fuzzy remembrance, you retrieve the actual phone number. This is a use of auto-association. Now if you want the individual’s name associated with a given phone number, that would require hetero-association.

### Conclusion

Throughout this section many different A.I techniques have been reviewed, some more complex than others and many inspired by human nature itself. Of all of the different techniques mentioned artificial neural networks have seen the most success in games with both Black & White and Colin McRae Rally 2 winning awards for innovative A.I.

Artificial Neural Networks (ANN's) are widely used and well-suited for generalization problems, although not as often discussed in literature, they make very good controllers for game agents. The reason behind this is that they form a type of function approximator; inputs to the network represent independent variables, while the outputs represent the dependant variables. In a game scenario the input variables could be information about the surrounding areas and the outputs could be the responses to these variables.

## Methodology

### Introduction

This section discusses the research methods utilized to study the ability of a neural network to learn and generalize given a particular scenario. It outlines decisions on methods to create learning data sets, training routines and test case development. Also discussed are features of the research environment and how they may affect testing and results.

The network chosen will be a multi layer perceptron; it will be tested with 3 hidden nodes and 6 hidden nodes. The network will be trained for a set number of epochs using the back propagation learning algorithm.

The neural network was designed to work as a controller for a game vehicle, the inputs to the network would be feeler like lines of sight and the output would control the rotation of the vehicle. The completed neural network should be able to navigate a vehicle around a track at a constant velocity without touching any side walls.

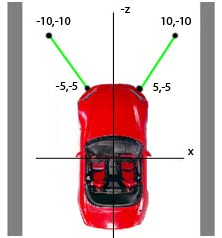


Figure Example of Feeler LOS

### Research

This project employed an experimental design. This form of research allowed the hypothesis to be proven mathematically using statistical analysis. Several reasons contributed to the use of a true experimental design, the first being that as results were of statistical nature the ability to discredit the results was removed. Another advantage was the ability to replicate the experiments and validate results at any time after the experiment was finished - this was made possible by guaranteeing the initial random weights for the network where the same each time the network was tested; this was achievable through loading initial weights from a .csv file. This type of design also meant it was much easier to manipulate one variable and see its effect on the overall output.

Unfortunately, problems do exist with this method. With this design all data should be recorded for analysis but due to the large number of discreet variables i.e. weights, activations etc. tracking these variables for each epoch was impossible.

### Neural Network Development

Rao explains that there are three aspects to the construction of a neural network:

1. Structure – the architecture and topology of the neural network
2. Encoding – the method of changing weights
3. Recall – the method and capacity to retrieve information

Structure relates to how many layers the network should contain, and what their functions are, such as for input, for output, or for feature extraction. Structure also encompasses how interconnections are made between neurons in the network, and what their functions are.

Encoding refers to the paradigm used for the determination of and changing of weights on the connections between neurons. In the case of a multilayer feed-forward neural network, the weights where determined randomly between (-0.5, 0.5). Subsequently, in the process of training, the back propagation algorithm was used. Once training was finished the weights no longer required updating.

Finally, recall is also an important aspect of a neural network. Recall refers to getting an expected output for a given input. If the same input as before is presented to the network, the same corresponding output as before should result.

The design requirements of the neural network were addressed using many techniques commonly found in software engineering. The first thing considered was a software development model. The most obvious model was the extreme programming cycle as the model welcomed changing requirements, even late in development where working programs where the primary measure of success. UML was employed as a method of visualizing the system before a single line of code was written.

### Extreme Programming

Extreme programming is a software engineering methodology (and a form of agile software development) that focus’s on short development cycles. This project adopted many of the stronger principles behind extreme programming:

*Test Driven Development (TDD)*

Test driven development involves implementing unit tests into code as it’s created as opposed to at the end of a coding period. The purpose of this is to test functionality of a program early on in development. Initially these tests fail as more code is often required but not yet implemented, once these tests pass its then fair to say the code is complete and working as expected. .

*Continuous Design*

Continuous design insures that the software stays in tune with what is required from it at the current stage in the development and not what the requirements where initially . On many projects external factors can cause milestones and deadlines to be missed; such as technology evolving too rapidly, publishers frequently change their minds or developers simply wanting to improve their work. Extreme programming isn’t bound by large design documentation; instead the client and the developer emphasize a close collaboration throughout, thus, allowing agreed changes to be implemented immediately without having to first update any design documentation.

*Energized Work*

Energized work is the practice of only working as many hours as you are being productive. Because programming is fundamentally about ideas, programmers can make lots of progress in a short amount of time if they can come up with the right ideas at the right time. .

Extreme programming has seen much success in game development due to its ability to adapt to changing requirements and well tested code . Although many of the principles described above are out-of-scope of this project, many of the underlying principles can be used.

At this point many of the decisions for choosing extreme programming should be clear. First of all, to make the most of time and resources it was important to organize the development process efficiently. Secondly, the project wanted a methodology that was able to cope with unstable requirements, programming A.I can be very frustrating and new or better ways of doing so may become more apparent over time. As with most modern methods (extreme programming or not), implementation was iterative and incremental.

### Unified Modelling Language Design

The unified modelling language is a graphical modelling syntax that’s aimed primarily at modeling software-based systems, particularly systems built using the object oriented (OO) style . Graphical modeling languages were used heavily in the 1980’s in the software industry; UML was born for the purpose of unifying many of these languages .

*Class Diagram*

Classes are the most important building block of any object oriented system . When projects are of large structure class diagrams can model the static design of a system without any code being implemented. A class diagram describes the types of objects in the system and the various kinds of static relationships that exist between them . Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. See Appendix A – Class Diagram

*State Machine Diagram*

State machine diagrams are a technique used to describe the behavior of a system . Unlike class diagrams they show the dynamic aspects of a system. In object oriented systems, state machine diagrams show behavior cycles of individual objects.

### Determining The Number of Hidden Layers and Hidden Nodes

In most situations, there is no way to determine the best number of hidden units without training several networks and estimating the generalization error of each. If the network has too few hidden units, it will get a high training error and high generalization error due to under fitting and high statistical bias. If it has too many hidden units, it may get low training error but still have high generalization error due to over fitting and high variance.

For this project 2 topologies will be tested, the first with 3 hidden nodes and a single hidden layer (Figure 13) and the second with 6 hidden nodes and a single hidden layer (Figure 14).

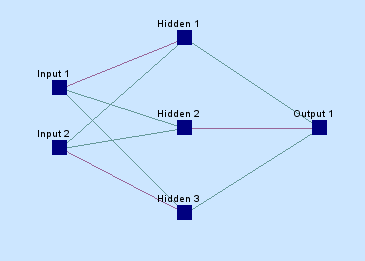


Figure 3 Hidden Nodes/ 1 Hidden Layer

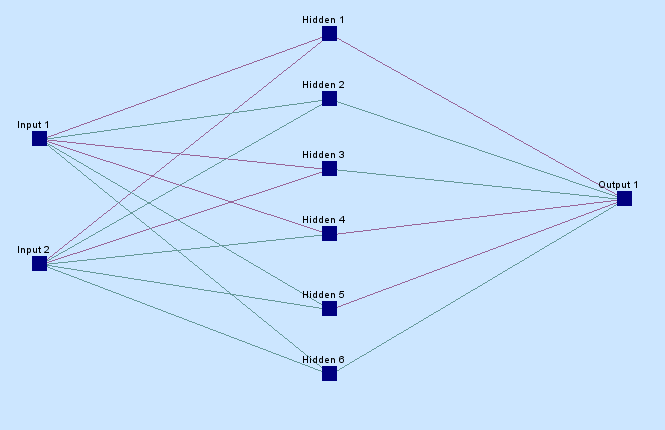


Figure 6 Hidden nodes/ 1 Hidden Layer

The hidden layer will be fed from 2 inputs and will feed into a single output.

### Dataset Generation

Dataset generation is the process of pre determining the correct input-outputs to train a neural network. In order for the neural network to work well after training it is critical that the training data covers all possible scenarios the neural network may find itself in.

*Inputs*

The vehicle will be operating with 2 feelers as its line of sight. This allows the vehicle to keep a track of all walls relative to the vehicle position.

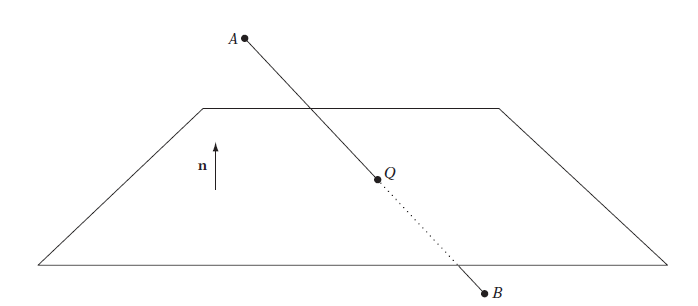


Figure Intersecting The Segment AB Against A Plane

Figure 15 shows how an intersection test between a feeler and wall may work.

The equation of a line is given by:

-   is any arbitrary point on the line.  In this case, is the point of intersection of the segment and the plane.

- is the starting point of the line

- is the end point of the line segment

- t is the parameter.  Values between 0 and 1 represent points on the line.

The equation of a plane is:

By substituting the equation of the segment for *X* in the plane equation:

This value of t can be used as an input as it is clamped between 0 and 1 and can be thought of as a percentage of intersection as shown in .

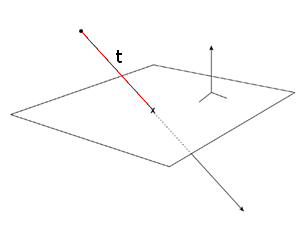


Figure Value of t

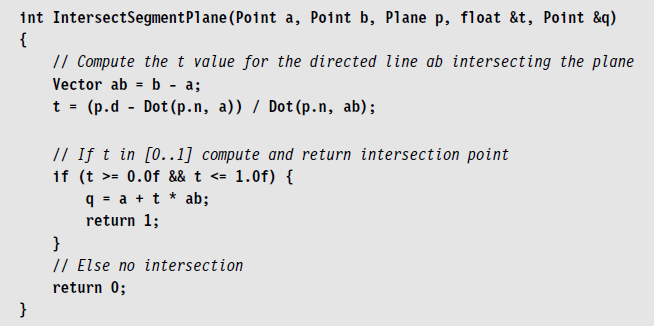


Figure Intersection Algorithm

is an algorithm from the Real Time Collision Book, this algorithm returns the point of intersection in 3d space, this is more detail than is required therefore the algorithm can be cut short and only the value for t needs to be returned.

*Outputs*

Given that the vehicle will be travelling at a constant velocity only one output will be required from the neural network. This output will be dependent on the inputs to the network. The first step in choosing an output involved choosing a range the network would output between. The log sigmoid activation function was a solution that produced an output between 0 and 1. Whilst this range required post processing to map it to a vehicle angle it provided an easy range for dataset generation.

Figure Linear Relationship Between Input and Output

shows how both feelers would map to the log sigmoid output. It was pre determined that an output between the ranges 0 – 0.5 would suggest an intersection on the left feeler and a clockwise rotation would be a result. An output >0.5 – 1.0 would be an intersection on the right feeler and an anti-clockwise rotation should be given.

Example:

Intersection on left feeler by 0.4 provides an output of 0.20, to convert this output to an angle it’s simply multiplied by 100 to give a rotation of 20°.

Intersection on right feeler by 0.4 provides an output of 0.70, as the output is greater than 0.5 an anticlockwise rotation is required, this is achieved by subtracting the output from the start of the range i.e. 0.50 – 0.70 = -0.2, again this is multiplied by 100 to represent an angle of -20°.

To achieve the best training dataset two tables were created, the first is a very simple map of all the possible scenarios the vehicle may find itself in (as shown in ). The second is a more in depth expansion on the first table () showing the correlation between an input and output.

Table

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Scenario | Left | Right | Resultant |  |  |  | | |  |
| 1 | O | O | No Rotation |  | Intersection | | I |
| 2 | I | O | Adjust Right | | No Intersection | | O |
| 3 | O | I | Adjust Left |  |  |  | | |  |
| 4 | I | I | Adjust Right | |  |  | | |  |

Table

|  |  |  |
| --- | --- | --- |
| Right | Left | Output |
| 0.00 | 0.00 | 0.00 |
| 0.00 | 0.20 | 0.10 |
| 0.00 | 0.40 | 0.20 |
| 0.00 | 0.60 | 0.30 |
| 0.00 | 0.80 | 0.40 |
| 0.00 | 1.00 | 0.50 |
| 0.20 | 0.00 | 0.60 |
| 0.40 | 0.00 | 0.70 |
| 0.60 | 0.00 | 0.80 |
| 0.80 | 0.00 | 0.90 |
| 1.00 | 0.00 | 1.00 |
| 0.20 | 0.20 | 0.10 |
| 0.40 | 0.40 | 0.20 |
| 0.60 | 0.60 | 0.30 |
| 0.80 | 0.80 | 0.40 |
| 1.00 | 1.00 | 0.50 |

### Training

In chapter 2, a method for training a neural network was researched. Using the same equations an example neural network given in will be solved to prove their ability to reduce an error.

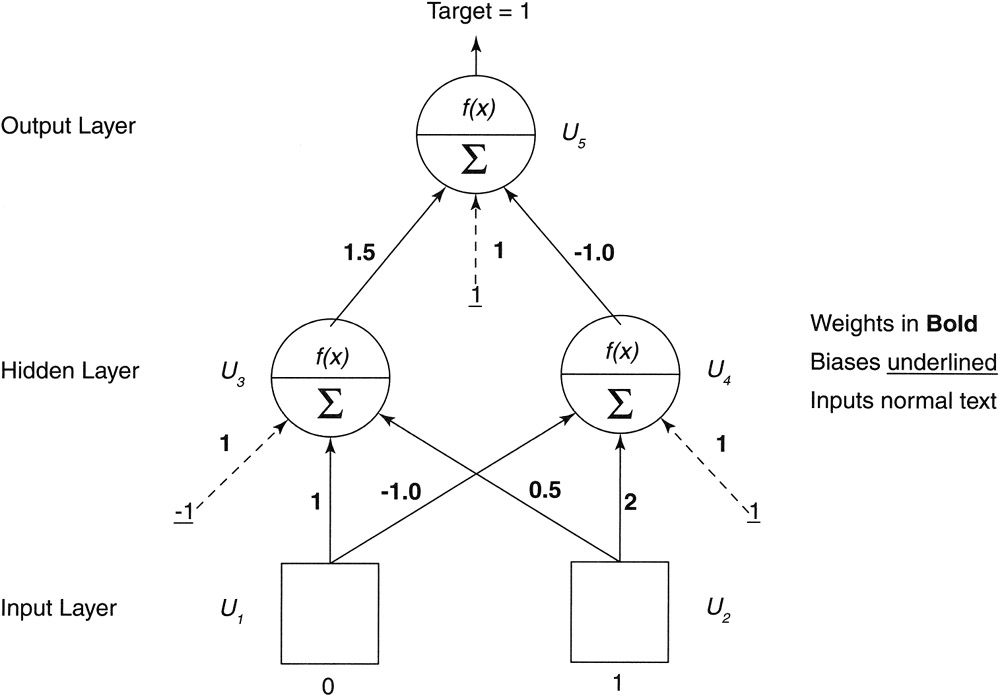


Figure Neural Network Training Example

Step 1: Initialization

Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range.

Step 2: Forward Pass

Feed forward the inputs through the network.

Using the sigmoid function discussed in Chapter 2:

Now for the output layer:

The target for the network is 1, the actual value calculated was 0.78139.

Step 3: The Error Backward Propagation Pass

Now the back propagation is applied, starting with the error of the output node and the hidden nodes.

Calculate the errors for both hidden nodes:

Step 4: Adjusting the connecting weights

Now that all error values have been calculated, using a learning rate (*p*) of 0.5 the relevant formula can be solved for each weight:

Now, update the output node bias:

In the case of , the weight was decreased where for the weight was increased. The bias was updated for greater excitation. Now for the adjustment of the hidden weights:

The final step is to update the node biases:

Proof:

Now that the new weights have been calculated it is possible to prove that the algorithm is reducing the error by calculating the mean squared error. Given by:

If the feed forward calculations are performed again using the new values, the following values are the result:

Compare this to the mean squared error before the change in weights:

This shows that in a single iteration of the back propagation algorithm the mean squared error was reduced by 0.001895.

### Procedure

Using the learning method described above the optimal solution can be found by using the following process for each cycle of the training set. Each cycle is called an *epoch*:

Forward Pass

1. Present an input from a data set and feed through the network using a provisional set of weights
2. Let hidden units process the data and feed to output units
3. Compute the difference between the predicted target value and the actual target value for this case

Backwards Pass

1. Propagate the error backward through the network and compute the change in error with respect to changes in weight values
2. Make adjustments to the weights to reduce the error

It was important not to generate a new set of weights for each test case scenario. Once one set of random weights where obtained these where exported then reused in further tests.

### Testing

Two different network topologies where tested in total, the first containing only 3 hidden nodes and the second containing 6. Both networks used a bias on each hidden and output node. Each test was conducted with random weights distributed between (-0.5, 0.5). A learning rate of (i) 0.05, (ii) 0.25, and (iii) 0.5 where selected for testing. For each learning rate, 1,000, 10,000, 25,000 and 50,000 epochs of training (using the same initial weights in each case) where performed.

Once the network finished training it was tested against a number of test cases. The test cases where derived from the chart mapping the inputs to the outputs (Figure 18) and are shown in Table 3.

Table

|  |  |  |  |
| --- | --- | --- | --- |
|  | Left | Right | Expected Result |
| Test 1 | 0 | 0.3 | 0.15 |
| Test 2 | 0 | 0.5 | 0.25 |
| Test 3 | 0 | 0.7 | 0.35 |
| Test 4 | 0 | 0.9 | 0.45 |
| Test 5 | 0.3 | 0 | 0.65 |
| Test 6 | 0.5 | 0 | 0.75 |
| Test 7 | 0.7 | 0 | 0.85 |
| Test 8 | 0.9 | 0 | 0.95 |

### Analysis

Excel was used to assemble descriptive charts to illustrate the training procedure. The error was sampled every ten epoch’s and exported to a .csv file for further analysis. At the end of every training cycle the results of each test case where also exported to a .csv.

## Results

The results given below are the product of 16 tests in total. Table 4 – 11 show the results produced by a network with 3 hidden nodes in a single hidden layer. Tables 12 – 19 show the results of a network with 6 hidden nodes and a single hidden layer.

The top row of each column indicates how many epoch of training a network undertaken before a result was obtained. The first column of each row indicates the learning rate associated with each network. For each test the network that produced the closest to expected output was highlighted.

*3 Hidden Nodes*

Table Test 1 Expected result 0.15

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.400707 | 0.227419 | 0.571729 | 0.50649 |
| 0.25 | 0.244051 | 0.531977 | 0.531822 | 0.53179 |
| 0.5 | 0.218382 | 0.147669 | 0.149751 | 0.152266 |

Table test 2 Expected result 0.25

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.438098 | 0.282189 | 0.574222 | 0.506493 |
| 0.25 | 0.295998 | 0.531979 | 0.531822 | 0.53179 |
| 0.5 | 0.283726 | 0.232385 | 0.241131 | 0.246324 |

Table Test 3 Expected result 0.35

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.486432 | 0.352228 | 0.576719 | 0.506497 |
| 0.25 | 0.363771 | 0.531982 | 0.531823 | 0.53179 |
| 0.5 | 0.367471 | 0.341206 | 0.349193 | 0.353724 |

Table Test 4 Expected result 0.45

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.537422 | 0.432988 | 0.579219 | 0.506502 |
| 0.25 | 0.44333 | 0.531985 | 0.531823 | 0.53179 |
| 0.5 | 0.461539 | 0.458604 | 0.45854 | 0.459572 |

Table Test 5 Expected Result 0.65

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.446982 | 0.73126 | 0.549408 | 0.506121 |
| 0.25 | 0.723003 | 0.531652 | 0.531728 | 0.531747 |
| 0.5 | 0.760462 | 0.752404 | 0.732316 | 0.723608 |

Table Test 6 Expected result 0.75

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.536127 | 0.786873 | 0.538248 | 0.506046 |
| 0.25 | 0.78461 | 0.531588 | 0.531711 | 0.531739 |
| 0.5 | 0.802525 | 0.848454 | 0.846983 | 0.844396 |

Table Test 7 Expected Result 0.85

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.610883 | 0.789828 | 0.52713 | 0.50597 |
| 0.25 | 0.788237 | 0.531523 | 0.531693 | 0.531731 |
| 0.5 | 0.803704 | 0.862459 | 0.871995 | 0.874951 |

Table Test 8 Expected Result 0.95

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.649439 | 0.78861 | 0.516017 | 0.505892 |
| 0.25 | 0.787402 | 0.531458 | 0.531675 | 0.531723 |
| 0.5 | 0.802133 | 0.864554 | 0.878273 | 0.884074 |

*6 Hidden Nodes*

Table Test 1 Expected Result 0.15

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.445784 | 0.163684 | 0.140391 | 0.132434 |
| 0.25 | 0.220622 | 0.153443 | 0.164447 | 0.152726 |
| 0.5 | 0.181838 | 0.13525 | 0.000001 | 0.161277 |

Table Test 2 Expected Result 0.25

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.419997 | 0.242961 | 0.234268 | 0.232012 |
| 0.25 | 0.279581 | 0.235349 | 0.227702 | 0.212279 |
| 0.5 | 0.27146 | 0.234138 | 0.000003 | 0.213891 |

Table Test 3 Expected Result 0.35

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.39555 | 0.355198 | 0.372039 | 0.376062 |
| 0.25 | 0.346592 | 0.345195 | 0.315516 | 0.296099 |
| 0.5 | 0.367537 | 0.352618 | 0.000007 | 0.28838 |

Table Test 4 Expected Result 0.45

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.372522 | 0.43315 | 0.449236 | 0.454014 |
| 0.25 | 0.403098 | 0.460781 | 0.42552 | 0.407468 |
| 0.5 | 0.437777 | 0.460703 | 0.000019 | 0.390175 |

Table test 5 Expected Result 0.65

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.533848 | 0.633835 | 0.635294 | 0.634135 |
| 0.25 | 0.672234 | 0.659606 | 0.673555 | 0.644994 |
| 0.5 | 0.717076 | 0.671772 | 1 | 0.641652 |

Table Test 6 Expected Result 0.75

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.570726 | 0.777543 | 0.757594 | 0.757132 |
| 0.25 | 0.824542 | 0.784103 | 0.800842 | 0.768754 |
| 0.5 | 0.855017 | 0.785675 | 1 | 0.762117 |

Table Test 7 Expected Result 0.85

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.603539 | 0.876966 | 0.869023 | 0.870177 |
| 0.25 | 0.868491 | 0.870083 | 0.870613 | 0.867679 |
| 0.5 | 0.882589 | 0.863618 | 1 | 0.867245 |

Table Test 8 Expected Result 0.95

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
|  |  |  |  |  |
| 0.05 | 0.628879 | 0.918887 | 0.925152 | 0.928411 |
| 0.25 | 0.883608 | 0.923355 | 0.916716 | 0.933524 |
| 0.5 | 0.891368 | 0.91346 | 1 | 0.939714 |

Table 20 – 25 shows the change in error for each network (0.05, 0.25, 0.5) for both 3 and 6 hidden node topologies. Each table shows every 10th epoch of the 50000 that where calculated. Tables for 1000, 10000 and 25000 epoch where omitted as this data can be derived from the given charts.

*3 Hidden Nodes*

Table

Table

Table

*6 Hidden Nodes*

Table

Table

Table

To summarize the results obtained through testing, Table 26 shows the error (between expected and actual output) for each tests best performing network. Both 3 and 6 hidden node topologies are shown.

Table

Table 27 and Table 28 show which networks won the most amount of tests, where win is defined by having the closest output to the expected output or least error.

Table 3 Hidden Node Win Table

|  |  |  |
| --- | --- | --- |
|  |  | Win's |
|  | 1k |  |
| 0.05 | 10k |  |
|  | 25k |  |
|  | 50k |  |
|  |  |  |
|  | 1k | II |
| 0.25 | 10k |  |
|  | 25k |  |
|  | 50k |  |
|  |  |  |
|  | 1k |  |
| 0.5 | 10k | I |
|  | 25k | III |
|  | 50k | III |

Table 6 Hidden Node Win Table

|  |  |  |
| --- | --- | --- |
|  |  | Win's |
|  | 1k |  |
| 0.05 | 10k | I |
|  | 25k | I |
|  | 50k | I |
|  |  |  |
|  | 1k |  |
| 0.25 | 10k |  |
|  | 25k |  |
|  | 50k | II |
|  |  |  |
|  | 1k |  |
| 0.5 | 10k | II |
|  | 25k |  |
|  | 50k | I |

## Discussion

Table Network Id's

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1000 | 10000 | 25000 | 50000 |
| Network 3 |  |  |  |  |
| 0.05 | 1 | 2 | 3 | 4 |
| 0.25 | 5 | 6 | 7 | 8 |
| 0.5 | 9 | 10 | 11 | 12 |
|  |  |  |  |  |
| Network 6 |  |  |  |  |
| 0.05 | 13 | 14 | 15 | 16 |
| 0.25 | 17 | 18 | 19 | 20 |
| 0.5 | 21 | 22 | 23 | 24 |

For the purpose of this discussion the neural networks that where tested will be labelled according to Table 29 Network Id's.

The artificial neural network has resulted in a non-deterministic model that could be implemented into a game agent to be used as a controller. Results from experiments confirm that a neural network can generalize and provide acceptable results provided it has been carefully trained.

Table 26 shows that at the end of each test the most successful network’s error were never greater than 0.1 (between the expected output and the actual output). Table 26 also shows that with exception to test 5 (shown in Table 8 and Table 16) the neural networks with 3 hidden nodes performed better than the 6 hidden node networks; smaller error’s where recorded on every test.

Whilst it can be seen from the data that each test produced a successful network, no single network won every test. Table 27 and Table 28 show that the 3 hidden node networks that were trained with a learning rate of 0.5 achieved the most success with network 11 winning 3 tests and network 12 winning another 3 tests.

On average the 6 hidden node networks achieved more success across the entire range with only network 13 and network 23 failing to converge, Table 23 and Table 25 shows why this has happened. Table 23 shows the error vs. epoch for networks 13 - 16. At 1k epoch (100 on the table), the error is still reducing and has not yet reach its global minimum; this means that the network has not yet converged thus resulting in actual outputs of no resemblance to expected. Table 25 however shows a different reason for not converging; it shows that the error in the output was caused by a plateau between 21k and 27k.

Back propagation has provided an effective and widely used architecture for the training of the artificial neural networks. However, statistical analysis showed that back propagation has problems—particular examples being Table 20, Table 21 and Table 26 where evidence of a plateau can be seen. In future work this could be avoided by using a stochastic descent method [30].

The advantages for neural networks are not clear-cut. For instance, there are types of AI systems that can readily exhibit forms of control. What has been confirmed through experiment is that neural networks can generalise well, meaning that once implemented in a game it should cope in any scenario.

As mentioned previously the most successful networks through training where network 11 and 12. Unfortunately both networks where only successful up to test 5, at this point the error for each test thereafter increased. This was not suitable for the purpose in which the neural network was required. As networks 11 and 12 where the best of the 3 hidden node networks the only option left was to further examine the 6 hidden node networks.

Of the 6 hidden node networks the most successful where networks 20 and 22. Further analysis of the test results shown that when comparing both networks ability to get closest to the actual value the result was a tie.

Table Comparison of Networks 20 and 22

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Test 6 | Test 7 | Test 8 |
| Network 20 | 0.152726 | 0.212279 | 0.296099 | 0.407468 | 0.644994 | 0.768754 | 0.867679 | 0.933524 |
| Network 22 | 0.13525 | 0.234138 | 0.352618 | 0.460703 | 0.671772 | 0.785675 | 0.863618 | 0.91346 |

To conclude this research the assumption can be made that either network 20 or 22 could be used as suitable controllers for a game agent. Both networks have the capacity to generalize and actual results remained consistent throughout testing.

## Future Improvements

Intel® have worked with neural networks in the past and have developed different methods for optimizing A.I algorithms to take advantage ofan Intel® Pentium® 4 Processor with Hyper-Threading Technology. During their testing Intel® recorded a 20% performance increase against an original unmodified source (as seen in Figure 20).

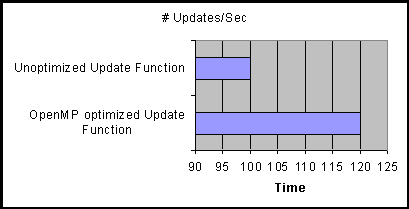


Figure Neural Update (Epoch's Completed) time speedup

Advances in physics processing units (PPU) may also play a part in improving the performance of neural networks in the future. By offloading some of the processing to a PPU means that the central processing unit (CPU) doesn’t have to work so hard in turn reducing the time taken for each process to execute.

# Bibliography

1. **David M. Bourg, Glenn Seeman.** *AI for Game Developers.* s.l. : O'Reilly, July 2004. Vol. O'Reilly. 0-596-00555-5.

2. **Schwab, Brian.** *AI Game Engine Programming.* s.l. : Jenifer Niles, 2004. 1-58450-344-0.

3. *Acquiring Visibly Intelligent Behavior with Example-Guided Neuroevolution.* **Bobby D. Bryant, Risto Miikkulainen.** s.l. : American Association for Artificial Intelligence, 2007.

4. *Gamebots: A 3D virtual world test-bed for multi-agent research.* **Adobbati, R, Marshall, A, Scholer A, Tejada, S, Kaminka, G.** New York : ACM Press, 2001.

5. **J. Stutz, P. Cheeseman.** *A Short Exposition on Bayesian Inference and Probability.* s.l. : National Aeronautic and Space Administration Ames Research Centre: Computational Sciences Division, Data Learning Group, 1994.

6. **Friedman, N. & M. Goldszmidt.** Learning Bayesian Network from Data. [Online] 1998. [Cited: 11 2008, 18.] http://www.erg.sri.com/people/moises/tutorial/index.htm.

7. **Heckerman, David.** *A tutorial on learning with Bayesian networks.* Redmond : Microsoft, 1996. MSR-TR-95-06.

8. **Luger, George F.** *Artificial Intelligence, Structures and Strategies for Complex Problem Solving.* Harlow, England : Addison- Wesley, 2002. 978-0321545893.

9. *Using a genetic algorithm to tune first-person shooter bots.* **N. Cole, S. Louis, and C. Miles.** pages 139 to 145, Oregon : IEEE Press, June 2004, Vol. Proceedings of the 2004 IEEE Congress on Evolutionary Computatio.

10. *The perceptron: a probabilistic model for information storageand organization.* **Rosenblatt, F.** s.l. : Psychological Review, 1958, Vol. 65.

11. **NASA.** NASA Neural Network Project Passes Milestone. *NASA.* [Online] 9 2, 2003. [Cited: 11 19, 2008.] http://www.nasa.gov/centers/dryden/news/NewsReleases/2003/03-49.html.

12. **Fausett, Laurene.** *Fundamentals of Neural Networks - Architectures, Algorithms and Applications.* s.l. : Pearson Education, Inc, 2006. 978-81-317-0053-2.

13. **Dewdney, A. K.** *Yes, We Have No Neutrons: An Eye-Opening Tour through the Twists and Turns of Bad Science.* s.l. : Wiley, 1997. 978-0471295860.

14. **Ojvind Bernander, M.S., Ph.D.** Neural Network. *Encarta.* [Online] [Cited: 07 20, 2008.] http://encarta.msn.com .

15. **Jones, M. Tim.** *AI Application Programming.* Massachusetts : David Pallai, 2003. 1-58450-278-9.

16. **Martin T. Hagan, Howard B. Demuth, Mark Beale.** *Neural Network Design.* s.l. : PWS Publishing, 1996. 81-315-0395-X.

17. **Rao, Valluru B.** *C++ Neural Networks and Fuzzy Logic .* s.l. : IDG Books Worldwide, Inc., 1995. 1558515526 .

18. *Learning internal representation by error.* **D. E. Rumelhart, G. E. Hinton, and R. J. Williams.** Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Cambridge, MA : MIT Press, 1986, Vol. 1.

19. **Farr, J Aaron.** Neural Networks in Computer and Cognative Science . *Cubicle Muses.* [Online] 10 10, 2003. [Cited: 12 03, 2008.] http://www.cubiclemuses.com/cm/blog/archives/000015.html/.

20. **Hannan, Jeff.** s.l. : Generation 5, 04 24, 2001.

21. **Mark Saunders, Philip Lewis, Adrian Thornhill.** *Research Methods for Business Students.* Essex : Pearson Education Limited, 2007. 978-0-273-70148-4.

22. **Swetnam, Derek.** *Writing your dissertation.* Oxford : How To Books Ltd, 2004. 978 1 85703 662 6.

23. "Extreme Programming" (lecture paper). *University of San Fransisco.* [Online] [Cited: 12 08, 2008.] http://www.cs.usfca.edu/~parrt/course/601/lectures/xp.html.

24. **Beck, Kent.** *Test-Driven Development by Example.* Boston : Addison-Wesley, 2003. 0321146530.

25. *Continuous Design.* **Shore, Jim.** 1, s.l. : IEEE Software, 2004, Vol. 21. http://www.martinfowler.com/ieeeSoftware/continuousDesign.pdf.

26. **Beck, Kent.** *Extreme Programming Explained, Embrace Change.* Boston : Addison-Wesley, 2004.

27. **Fristrom, James.** Treyarch's Spider-Man. *Game Developer Magazine.* August, 2002.

28. **Fowler, Martin.** *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* s.l. : Addison Wesley, 2003. 0-321-19368-7.

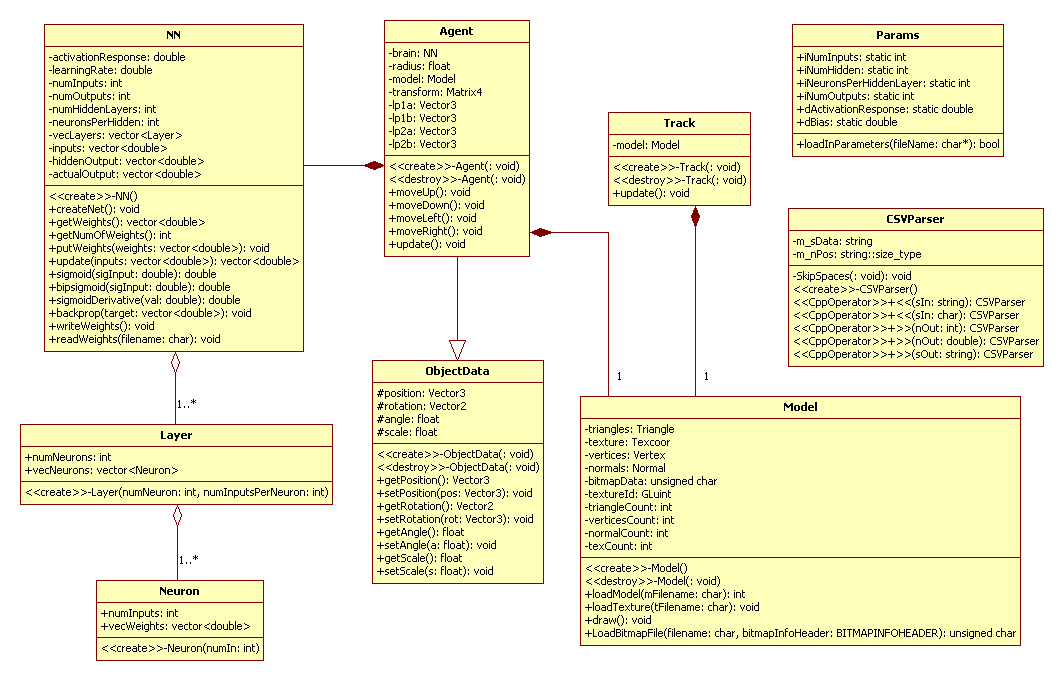
29. **Grady Booch, James Rumbaugh, Ivar Jacobson.** *The Unified Modeling Language User Guide .* s.l. : Addison Wesley Professional, 2005. 0-321-26797-4.

30. **Kessler, Laurie Williams and Robert.** *Pair Programming Illuminated.* Boston : Addison-Wesley, 2003. 0201745763.

31. **Bourg, David M.** *Physics for Game Developers.* California : O'Reilly, 2002. 0-596-00006-5.

32. **Ericson, Christer.** *Real Time Collision Detection.* s.l. : Morgan Kaufmann. 1-55860-732-3.

## Appendix A – Class Diagram



## Appendix B – Program Code

Layer.cpp

#include "Layer.h"

//Create a layer of neurons of the required size by calling the Neuron constructor

//the required number of times

Layer::Layer(int numNeuron, int numInputsPerNeuron)

{

numNeurons = numNeuron;

for (int i = 0; i < numNeurons; i++)

{

vecNeurons.push\_back(Neuron(numInputsPerNeuron));

}

}

Neuron.cpp

#include "Neuron.h"

#include <time.h>

#include <stdlib.h>

Neuron::Neuron(int numIn):numInputs(numIn +1)

{

//srand((unsigned int)time(0));

//for(int i=0; i<numIn + 1; i++)

for(int i=0; i<numIn; i++)

{

//set up weights with initial random value

vecWeights.push\_back(randFloat());

}

vecWeights.push\_back(Params::dBias);

}

NN.cpp

//Create a NN based on the values set below

NN::NN()

{

numInputs = Params::iNumInputs;

numOutputs = Params::iNumOutputs;

numHiddenLayers = Params::iNumHidden;

neuronsPerHidden = Params::iNeuronsPerHiddenLayer;

activationResponse = Params::dActivationResponse;

learningRate = Params::dLearningRate;

createNet();

}

//Creates a NN with weights initially set between -1 and 1

void NN::createNet()

{

//create the layers of the network

if(numHiddenLayers > 0)

{

//Push first hidden layer

vecLayers.push\_back(Layer(neuronsPerHidden, numInputs));

//Push every additional layer

//If only one hidden this will be missed

for( int i = 0; i < numHiddenLayers - 1; i++)

{

//Layer(int numNeuron, int numInputsPerNeuron)

vecLayers.push\_back(Layer(neuronsPerHidden, neuronsPerHidden));

}

//Create ouput layer

vecLayers.push\_back(Layer(numOutputs, neuronsPerHidden));

}

else

{

//If no hidden layers, create output layer

vecLayers.push\_back(Layer(numOutputs, neuronsPerHidden));

}

}

vector<double> NN::getWeights() const

{

//temp store for weights

vector<double> weights;

//for each layer

for (int i = 0; i < numHiddenLayers + 1; ++i)

{

//for each neuron

for (int j = 0; j < vecLayers[i].numNeurons; ++j)

{

//for each weight

for (int k = 0; k < vecLayers[i].vecNeurons[j].numInputs; ++k)

{

weights.push\_back(vecLayers[i].vecNeurons[j].vecWeights[k]);

}

}

}

return weights;

}

//Given a vector of doubles this function replaces the weights in the NN

//with the new values

void NN::putWeights(vector<double> &weights)

{

int weight = 0;

//for each layer

for (int i = 0; i < numHiddenLayers + 1; ++i)

{

//for each neuron

for (int j = 0; j < vecLayers[i].numNeurons; ++j)

{

//for each weight

for (int k = 0; k < vecLayers[i].vecNeurons[j].numInputs; ++k)

{

vecLayers[i].vecNeurons[j].vecWeights[k] = weights[weight++];

}

}

}

return;

}

//Returns the total number of weights needed for the net

int NN::getNumOfWeights()

{

int weights = 0;

//for each layer

for (int i = 0; i < numHiddenLayers + 1; ++i)

{

//for each neuron

for (int j = 0; j < vecLayers[i].numNeurons; ++j)

{

//for each weight

for (int k = 0; k < vecLayers[i].vecNeurons[j].numInputs; ++k)

{

weights++;

}

}

}

return weights;

}

double NN::sigmoid(double sigInput)

{

//Returns a value between 0..1 this curve is always centred around 0.5.

//Negative activation values produce a result less than 0.5, positive activation values produce a result greater than 0.5.

return( 1.0 / ( 1.0 + exp(-sigInput)));

}

double NN::sigmoidDerivative(double sigInput)

{

return( sigInput \* (1.0 - sigInput) );

}

//Given an input vector this function calculates the output vector

vector<double> NN::update(vector<double> &i)

{

actualOutput.clear();

hiddenOutput.clear();

inputs.clear();

for( int z = 0; z < numInputs; z++)

{

inputs.push\_back(i[z]);

}

//Get the weights

vector<double> weights;

weights = getWeights();

double sum;

int indx = 0;

//Calculate the input to hidden layer

for(int hid = 0; hid < neuronsPerHidden; hid++)

{

sum = 0.0;

for(int inp = 0; inp < numInputs; inp++)

{

sum += inputs[inp] \* weights[indx];

indx ++;

}

//Add in the Bias

sum += weights[indx];

indx ++;

hiddenOutput.push\_back(sigmoid(sum));

}

//Calculate the hidden to output layer

for(int out = 0; out < numOutputs; out++)

{

sum = 0.0;

for( int hid = 0; hid < neuronsPerHidden; hid++)

{

sum += hiddenOutput[hid] \* weights[indx];

indx++;

}

//Add in bias

sum += weights[indx];

indx++;

actualOutput.push\_back(sigmoid(sum));

}

return actualOutput;

}

float NN::backprop(vector<double> &target)

{

errorOutput.clear();

///////////////////////////////////////////////////

//Calculate output layer error

///////////////////////////////////////////////////

for(int out = 0; out < numOutputs; out++)

{

errorOutput.push\_back((target[out] - actualOutput[out]) \* sigmoidDerivative(actualOutput[out]));

}

///////////////////////////////////////////////////

//Calculate the hidden layer error

///////////////////////////////////////////////////

vector<double> weights;

weights = getWeights();

int indx = (numInputs +1) \* neuronsPerHidden;

for( int hid = 0; hid < neuronsPerHidden; hid++)

{

errorHidden.push\_back(0.0);

for( int out = 0; out < numOutputs; out++)

{

errorHidden[hid] += errorOutput[out] \* weights[indx + hid];

}

errorHidden[hid] \*= sigmoidDerivative(hiddenOutput[hid]);

}

///////////////////////////////////////////////////

//Update the hidden to output weights

///////////////////////////////////////////////////

indx = (numInputs +1) \* neuronsPerHidden;

for(int out = 0; out < numOutputs; out++)

{

for( int hid = 0; hid < neuronsPerHidden; hid++)

{

weights[indx + hid] += (learningRate \* errorOutput[out] \* errorHidden[hid]);

}

//Update the bias

int i = weights.size() - 1;

weights[i] += (learningRate \* errorOutput[out]);

}

///////////////////////////////////////////////////

//Update the input to hidden weights

///////////////////////////////////////////////////

indx = 0;

int b = 2;

for( int hid = 0; hid < neuronsPerHidden; hid++)

{

for(int inp = 0; inp < numInputs; inp++)

{

indx = inp;

weights[indx + hid] += (learningRate \* errorHidden[hid] \* inputs[inp]);

}

indx ++;

//Update the input Bias

weights[b] += (learningRate \* errorHidden[hid]);

b += 3;

}

//Update the network with the newly calculated weights

putWeights(weights);

return target[0] - actualOutput[0];

}

void NN::writeWeights()

{

FILE\* fWeights;

fWeights = fopen("Weights.csv", "w");

//temp store for weights

vector<double> weights;

weights = getWeights();

for(int i = 0; i < getNumOfWeights(); i++)

{

fprintf(fWeights,"%f", weights[i]);

fprintf(fWeights,"%s\n", "");

}

}

CSVParser.cpp

#include <iostream>

#include <cstdlib>

#include "csvparser.h"

using namespace std;

CSVParser::CSVParser()

{

m\_sData = "";

m\_nPos = 0;

}

void CSVParser::SkipSpaces(void)

{

while (m\_nPos < m\_sData.length() && m\_sData[m\_nPos] == ' ')

m\_nPos++;

}

const CSVParser & CSVParser::operator <<(const string & sIn)

{

this->m\_sData = sIn;

this->m\_nPos = 0;

return \*this;

}

const CSVParser & CSVParser::operator <<(const char \*sIn)

{

this->m\_sData = sIn;

this->m\_nPos = 0;

return \*this;

}

CSVParser & CSVParser::operator >>(int & nOut)

{

string sTmp = "";

SkipSpaces();

while (m\_nPos < m\_sData.length() && m\_sData[m\_nPos] != ',')

sTmp += m\_sData[m\_nPos++];

m\_nPos++; // skip past comma

nOut = atoi(sTmp.c\_str());

return \*this;

}

CSVParser & CSVParser::operator >>(double & nOut)

{

string sTmp = "";

SkipSpaces();

while (m\_nPos < m\_sData.length() && m\_sData[m\_nPos] != ',')

sTmp += m\_sData[m\_nPos++];

m\_nPos++; // skip past comma

nOut = atof(sTmp.c\_str());

return \*this;

}

CSVParser & CSVParser::operator >>(string & sOut)

{

bool bQuotes = false;

sOut = "";

SkipSpaces();

// Jump past first " if necessary

if (m\_nPos < m\_sData.length() && m\_sData[m\_nPos] == '"') {

bQuotes = true;

m\_nPos++;

}

while (m\_nPos < m\_sData.length()) {

if (!bQuotes && m\_sData[m\_nPos] == ',')

break;

if (bQuotes && m\_sData[m\_nPos] == '"') {

if (m\_nPos + 1 >= m\_sData.length() - 1)

break;

if (m\_sData[m\_nPos+1] == ',')

break;

}

sOut += m\_sData[m\_nPos++];

}

// Jump past last " if necessary

if (bQuotes && m\_nPos < m\_sData.length() && m\_sData[m\_nPos] == '"')

m\_nPos++;

// Jump past , if necessary

if (m\_nPos < m\_sData.length() && m\_sData[m\_nPos] == ',')

m\_nPos++;

return \*this;

}

Main.cpp

#include "NN.h"

#include "csvparser.h"

#include <fstream>

#include <string>

#include <iostream>

using namespace std;

#define sqr(x) ((x)\*(x))

#define TEST 1000

Params parameters;

NN theBrain;

void main(void)

{

//Inputs

vector<double> input;

//Outputs

vector<double> actual;

vector<double> expected;

/////////////////////////////////Load Training Data/////////////////

ifstream infile("training.csv");

string sLine;

double dCol1, dCol2, dCol3;

CSVParser parser;

while (!infile.eof())

{

getline(infile, sLine); // Get a line

if (sLine == "")

continue;

parser << sLine; // Feed the line to the parser

parser >> dCol1 >> dCol2 >> dCol3;

input.push\_back(dCol1); //Left feeler

input.push\_back(dCol2); //Right feeler

expected.push\_back(dCol3); //Expected output

}

infile.close();

/////////////////////////////////Load Test weights/////////////////

//Store for test weights

vector<double> tWeights;

//"TestingWeights3" for 3 hidden nodes, "TestingWeights6" for 6 hidden

ifstream ifile("TestingWeights6.csv");

while (!ifile.eof())

{

getline(ifile, sLine); // Get a line

if (sLine == "")

continue;

parser << sLine; // Feed the line to the parser

parser >> dCol1;

tWeights.push\_back(dCol1); //Left feeler

}

ifile.close();

theBrain.putWeights(tWeights);

tWeights.clear();

tWeights = theBrain.getWeights();

//////////////////Run the training data throught the NN ////////////

//How many training pairs

int sExpect = expected.size();

//Test output file

FILE\* fError;

fError = fopen("Error.csv", "w");

float totError; //Stores accumilative error

float output; //Stores MSE

int count = 0; //Counter for test output

for(int epoch = 0; epoch < TEST; epoch++)

{

totError = 0.0f;

for( int i = 0; i < sExpect; i++)

{

//Feed forward

actual = theBrain.update(input);

totError += sqr(theBrain.backprop(expected));

//Move to next training set

input.push\_back(input[0]);

input.erase(input.begin()); //Remove left input

input.push\_back(input[0]);

input.erase(input.begin()); //Remove right input

expected.push\_back(expected[0]);

expected.erase(expected.begin()); //Remove output

}

count++;

//Print every 10th output to file

if(count == 10)

{

output = 0.5 \* totError;

fprintf(fError,"%f", output);

fprintf(fError,"%s\n", "");

count = 0;

}

}

//////////////////Test the NN with test inputs ////////////

FILE\* fTests;

fTests = fopen("Tests.csv", "w");

//Test 1

input.clear();

input.push\_back(0);

input.push\_back(0.3);

actual = theBrain.update(input);

cout << actual[0] << endl;

fprintf(fTests,"%f", actual[0]);

fprintf(fTests,"%s\n", "");

input.erase(input.begin());

input.erase(input.begin());

//Test 2

input.push\_back(0);

input.push\_back(0.5);

actual = theBrain.update(input);

cout << actual[0] << endl;

fprintf(fTests,"%f", actual[0]);

fprintf(fTests,"%s\n", "");

input.erase(input.begin());

input.erase(input.begin());

//Test 3

input.push\_back(0);

input.push\_back(0.7);

actual = theBrain.update(input);

cout << actual[0] << endl;

fprintf(fTests,"%f", actual[0]);

fprintf(fTests,"%s\n", "");

input.erase(input.begin());

input.erase(input.begin());

//Test 4

input.push\_back(0);

input.push\_back(0.9);

actual = theBrain.update(input);

cout << actual[0] << endl;

fprintf(fTests,"%f", actual[0]);

fprintf(fTests,"%s\n", "");

input.erase(input.begin());

input.erase(input.begin());

//Test 5

input.push\_back(0.3);

input.push\_back(0);

actual = theBrain.update(input);

cout << actual[0] << endl;

fprintf(fTests,"%f", actual[0]);

fprintf(fTests,"%s\n", "");

input.erase(input.begin());

input.erase(input.begin());

//Test 6

input.push\_back(0.5);

input.push\_back(0);

actual = theBrain.update(input);

cout << actual[0] << endl;

fprintf(fTests,"%f", actual[0]);

fprintf(fTests,"%s\n", "");

input.erase(input.begin());

input.erase(input.begin());

//Test 7

input.push\_back(0.7);

input.push\_back(0);

actual = theBrain.update(input);

cout << actual[0] << endl;

fprintf(fTests,"%f", actual[0]);

fprintf(fTests,"%s\n", "");

input.erase(input.begin()); //Remove left input

input.erase(input.begin()); //Remove right input

//Test 8

input.push\_back(0.9);

input.push\_back(0);

actual = theBrain.update(input);

cout << actual[0] << endl;

fprintf(fTests,"%f", actual[0]);

fprintf(fTests,"%s\n", "");

}

Params.ini

iNumInputs 2

iNumHidden 1

iNeuronsPerHiddenLayer 6

iNumOutputs 1

dActivationResponse 1

dLearningRate 0.5

dBias 1