|  |
| --- |
| An Evaluation of Fast Multi-Layer Perceptron Training Techniques for Games  David Robertson  Computer Games Applications Development, 2017 |

|  |
| --- |
| School of Arts, Media and Computer Games  Abertay Dundee |

Table of Contents

[List of Figures ii](#_Toc447628185)

[List of Tables iii](#_Toc447628186)

[Acknowledgements iv](#_Toc447628187)

[Abstract v](#_Toc447628188)

[Abbreviations, Symbols and Notation vi](#_Toc447628189)

[Introduction 1](#_Toc447628190)

[Introduction 1](#_Toc447628190)

[Appendices 2](#_Toc447628191)

[List of References 3](#_Toc447628192)

[Bibliography 4](#_Toc447628193)

# List of Figures

# List of Tables

# Acknowledgements

# Abstract

# Abbreviations, Symbols and Notation

Multi-layer perceptron - MLP

Error Back Propagation - EBP

Random-Minimum Bit Distance Gram-Schmidt - RMGS

Artificial Neural Network - ANN

Resilient propagation - RPROP

Artificial Intelligence - AI

# 1 Introduction

Artificial intelligence (AI) has been at the absolute core of video games since the beginning, as Alex Champandard (2004) states “since the days of Pong and Pac-man artificial Intelligence has played an undeniable role in computer games.” AI has been fundamental to keeping computer games engaging and enjoyable. Because of this, game AI has continued to develop and evolve over the years, slowly introducing more academic techniques into the field and adapting them to suit what will make the game the most fun. Games have easily been able to adopt the most basic of AI: from rule-based systems; in which the rules for how the AI is going to act depending on the situation are written out in full and the programmer must account for every possibility; to some of the more complex techniques such as genetic algorithms; which requires very little programming and allows the AI to evolve on its own to find the optimal solution to the problem. One type of academic AI has been an outlier to this trend; Artificial Neural Networks (ANN). ANNs have been around for a very long time; with the original “Logic Threshold Unit” being proposed by Warren McCulloch and Walter Pitts in 1943 (Stanford University 2000). However, due to their computational demands and long training times, they have never really found a place in games.

There have been a number of attempts to implement ANNs into video games, but nonetheless they have all done just what a far less complicated game AI technique could easily have taken care of. There are a number of drawbacks to using a neural network for controlling the game AI; including that if offline training is used, then once the network has been trained, it has been trained for good and can no longer learn on the fly. Online learning allows this kind of on the fly learning, but the majority of learning algorithms for neural networks are unsuitable for this and must be adapted for real-time dynamic processes (Charles and McGlinchey 2004).

The key problem with implementing an artificial neural network in a game is the training time. It takes hundreds of iterations to train the network, so if any adaptations have to be made or the training data was incorrect the entire process will have to be stopped and restarted with the updated training data. Thus there have been attempts at different methods of training a neural network in particular, the multi-layer perceptron (MLP) neural network, has had many different training methods proposed to speed up learning time. Methods such as Quick Propagation and Resilient Propagation reduce some of the issues with Error Back Propagation (EBP) and are "batch" methods (Champandard 2004) which inevitably speed up the process. However, they do not reduce the time significantly.

The algorithm that this project will mainly compare to error back propagation is the “Random-Minimum Bit Distance Gram-Schmidt” (RMGS) method (Verma 1997). The training time for this particular method is negligible as it trains the entire neural network in one iteration instead of hundreds. It is noted that this method is not as accurate as other methods. However, in a game scenario, it is actually beneficial in some cases for the AI not to be 100% accurate, otherwise the player would never be able to beat them. Since this method only takes one iteration to train the network, there is potential for MLPs to be able to be used and trained on the fly during a game, and if it is feasible and accurate enough, it may finally initiate an interest in the use of this mature technique in games. This project aims to prove that feasibility.

The relevance of this project is that it aims to discover if using a faster method of training an MLP will perform competitively to the standard error back propagation method. If this were to be successful then it would show that not only can MLP networks be trained more quickly, but also in particular, using the RMGS method, they have the potential to be retrained in real time rather than taking incredible amounts of time to do this.

## 1.1 Research Question

Can alternative training methods for multi-layer perceptron neural networks compete in performance with error back propagation in order to promote the use of MLPs in games?

## 1.2 Aim

This project aims to evaluate the effectiveness of different multi-layer perceptron training methods by comparing their performance in controlling a vehicle in a top down racing game.

## 1.3 Structure

The sections of this paper will proceed in the following order: Section 2 will discuss the background of AI in games along with an explanation of multi-layer perceptrons and the training techniques. Section 3 will be an in depth discussion on how the RMGS training method works. Following this, section 4 will explain the methodology of the project. The results of this investigation will then be presented, and explained in sections 5 and 6. Finally, section 7 will cover the conclusions drawn and a discussion on future work that could develop from this project.

# 2 Literature Review

It has been proven that MLPs can control a car in a racing game, for example Colin McRae Dirt 2 utilises this for its game AI. However, training multi-layer perceptrons takes a lot of time and thus they are rarely used in games. This project aims to prove that alternative training methods for MLP networks can not only reduce the training time, but be as effective as the most commonly used "Error Back Propagation".

Racing games can be identified as excellent grounds for testing MLP networks as there are many potential inputs to process for driving a racing car around a track. For this kind of neural network, as the number of inputs increases, the harder the network has to work. This will test the Random Minimum Bit Distance Gram-Schmidt (RMGS) and resilient training methods thoroughly. This section will firstly describe an MLP network and how it works, followed by some games that use ANNs and finish with descriptions of each of the training methods that are going to be implemented in this project.

## 2.1 AI and Games

\*\*\*History lesson on Game AI\*\*\*\*\*

## 2.2 Artificial Neural Networks

\*\*\*General description of ANNs including SOM and ART\*\*\*\*

Artificial Neural Networks come in many different forms, with the most common being the Multi-layer Perceptron. Other examples of ANNs are the Self-Organising Map (SOM) and the Adaptive Resonance Theory Network (ART). Both of these are defined as unsupervised learning networks, which means that they are not given a target output when being trained, thus they train themselves. SOMs are a type of clustering algorithm in which they group similar patterns/inputs together. The design of this type of network is shown in the figure below:

\*\*\* SOM DIAGRAM \*\*\*

The weights are set to random values, the inputs are then passed to the network, this is done via the calculation:

\*\*\* SOM EQATION \*\*\* D(J) = E(wij – xi)^2;

With *x* representing the input vector, *w* representing thew weight vector and *J* being the index of the neuron. Once the *J* neuron with the lowest value is found, the neurons around the *Jth* neuron are updated with the following formula:

\*\*\* SOM LEARNING \*\*\* wij(new) = wij(old) + delta[xi – wk(old)];

Thus the surrounding neurons will be the correct neuron for vectors that are similar to the current input vector. This allows for the network to do as the name describes: self-organise.

ART networks are another clustering technique. There are multiple kinds of ART network with the most basic being ART1 which is used for clustering binary input vectors. The dseign of this network can be described with the following figure:

\*\*\* ART1 DIAGRAM \*\*\*

MLPs are defined as a supervised learning method. Supervised learning is when you have your input variables and desired output variables before training begins, this means that the network simply has to find a way to turn input the input variables into the matching output variables. The MLP network will be discussed in full in section 2.4.

## 2.3 Creatures

One of the only commercial games to have featured an artificial neural network at the heart of the game AI is Millennium Interactive’s "Creatures" (link to the past present and future of artificial neural networks in games). The entire game is based around interacting with 2D "living" creatures, each of which has its own heterogeneous neural network for a brain, which is built up of 1000 neurons grouped into nine lobes with around 5000 synapses connecting neurons. (link to creatures 1996). Figure 1 shows how the lobes of the creatures' brains were arranged and gives an idea of how they worked. The game is entirely built around the player interacting with the "Norns" to help them learn and evolve. This has been noted as one of the most influential games ever in regards to game AI, as it was one of the first popular games to apply machine learning to a simulation. (AI game dev top 10 influential games) Below is the structure of the ANN that was inside each of the “Norns’ heads”:



*Figure 1. "Creatures brain layout" creatures 1996 paper*

The brain of the “Norns” was nothing really like a common ANN it was more like an attempt at recreating a brain, using designated lobes to interact with certain stimuli. Although there are multiple types, nowadays when referring to ANNs it is commonly perceived as referring to Multi-Layer Perceptrons.

## 2.4 Multi-Layer Perceptron Network

The multi-layer perceptron (MLP) is one of the most well-known and used artificial neural networks. It is classified as a “feed-forward” ANN that has the ability to map sets of input data to output data. Figure 2 shows an example of how an MLP network looks and gives an insight into the process of its workings:



*Figure 2. An example of a Multi-Layer Perceptron Neural Network(Kawaguchi 2000)*

The general function of the network is best described as for each neuron in any layer other than the input layer: For each of these neurons every output of the previous layer is multiplied by a corresponding weight, summed together and then passed through an activation function. Figure 3 accurately demonstrates the process for each neuron:



*Figure 3. Symbolic Illustration of Linear Threshold Gate(Kawaguchi 2000)* [*http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html*](http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html)

Many activation functions can be used in an MLP network. They range from fairly simple ones such as the linear threshold function, as seen above, which only triggers if the value passes a certain threshold to more complex functions such as the “sigmoid” function”, which the following equation describes:

(Equation )

This allows for the neuron to fire at any time but with varying values rather than just a one or a zero.

\*\*\*\* ADD SIGMOID PICTURE \*\*\*\*

## 2.5 Spiking vs Multi-Layer Perceptron Neural Networks

In the paper "Spiking Neural Network vs multi-layer perceptron: who is the winner in the racing car computer game" the authors Urszulla Markowska-Kaczmar and Mateusz Koldowski created a top down racing game in which the competing networks would race against one another to aim for the best time around the track. Both of the networks are trained via genetic algorithms and given points based on their performance for further evolutions.

Spiking Neural Networks are the third generation of neural network (Markowska-Kaczmar and Koldowski 2015). They are modelled to most accurately represent a brain. Each neuron in a Spiking Neural network is given an activation voltage and the current voltage that the neuron has will be calculated by the input of the network. Once the activation voltage has been reached, a pulse will fire and the neuron will revert to the resting voltage. This means that the input of the network will not affect the size and shape of the "pulse" from the neuron; instead, it will determine when it fires. They are regarded as a "computationally powerful and biologically more plausible model of distributed computation" (Yee and Teo 2011).

This project is similar to this one, in that it is comparing the performance of two types of neural networks in a racing game; the difference is that the comparison in this project is between two training methods for multi-layer perceptron networks instead. The paper states that the car uses ray casted sensors to follow a racing line around the track.

\*\* methodology move below \*\*

In this project the car will also use sensors to find its position, however it will be searching for waypoints that will be placed on the track. The reason for this change is that when given a training line, it is far easier for the ANN to know where it needs to be, as the adjustments for driving are constant. With waypoints the ANN must try to line up with the perpendicular position before it reaches it thus requiring better training to do so. A similar evaluation technique will also be used in this project; as the time it takes for the car to get around the track is the best measure of the performance of the training methods in a racing game scenario.

\*\*\*\*\*\*\*\*

## 2.6 Colin McRae Dirt 2

Colin McRae Dirt 2 is another one of the few games that has been documented to use an Artificial Neural Network to control the AI in game. In an interview with the website "AI-Junkie" the programmer responsible for the game’s AI development, Jeff Hannan, evaluated his work and how the process worked. The neural network that he implemented was not extremely complex stating that "The outputs from the neural net were simply on/off flags for pressing buttons on the controller" (Buckland 2004), and that the neural net was only programmed to follow the racing line as "The AI in these other situations was simply rule based when it came to crashes and overtaking.” This shows that another, more simple AI technique could have been implemented in its place. It was also noted that in this interview Hannan had to constantly fiddle with the number of neurons and structure of the network to get it to train properly, this would suggest that he was forced not to use EBP as it takes so long to train. The training method that was implemented in this neural network was resilient back propagation (RPROP), as it has been proven to be faster than the classic EBP training technique (Champandard 2004).

\*\* expand on this area a bit\*\*

\*\* again move to methodology somehow \*\*

This project could be compared to Hannan’s work and is somewhat a spiritual continuation on the AI that was implemented. For example, the output of the neural network in this project will be flags for the buttons on the "controller" and the neural network will be a multi-layer perceptron network.

However, this project also has some differences including; using ray casts and waypoints around the track to steer correctly instead of following a racing line for the same reasons as discussed previously, and although resilient propagation will be implemented, the RMGS training method (which will be discussed in the next section) will be the focus of this project.

\*\*\*

## 2.7 Training Methods

### 2.7.1 Error Back Propagation

The error back-propagation method is the most common training method for multi-layer perceptron neural networks. The basics of the technique were first proposed in 1960 by Henry J. Kelley in terms of control theory, however it has been noted that “it’s importance was not fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton and Ronald Williams” (Nielsen 2017 <http://neuralnetworksanddeeplearning.com/chap2.html> ). Firstly, each of the weights in the MLP network are set to small random values. Then the first values of training data are passed through the MLP network to give an output. The network then calculates the error of the current output compared to the desired training data output using the “square error” function:

Equation

The error is then used to adjust the weights of the output layer's neurons. This process is repeated for all of the hidden layers, working backwards from the output layer until the entire network has been corrected. This entire process is repeated for the entire training data set multiple times until the calculated error reaches a minimum. (Bourg and Seemann 2004)

\*\*\*\* talk about the delta rule here and mention gradient decent \*\*\*\*\*

### 2.7.2 Resilient Propagation

First proposed by Mark Reidmiller and Heinrich Braun in 1993 Resilient Propagation (RPROP) aimed “To overcome the inherent disadvantages of pure gradient- descent” (Reidmiller Braun 1993 <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=298623> ). Reidmiller and Braun found that their training method out performed the classic EBP with ease and other training techniques such as “Quick Propagation” and “SuperSAB”. RPROP works somewhat similarly to Error Back Propagation in the sense that all the weights are updated depending on a calculated error. However, RPROP does not update the weights until all of the training data has been seen, therefore it is known as a “batch algorithm”. As the weights are not updated after each piece of training data an “error gradient” must be calculated for each neuron. This is done by passing all of the training data through the network and calculating a gradient for the error on the neuron. Once this has been completed, the weights are then adjusted accordingly in relation to the gradient of error that has been calculated. (Champandard 2004) Although the weights are adjusted in relation to the gradient, the gradient does not decide the size of the step used to update the weight. Thus eliminating any problems that involve a weight adjustment that is too steep. The general theory is very simple, as Champandard states “If the slope goes up, we adjust the weight downward. Conversely, the weight is adjusted upward if the gradient is negative.” And if neither of these are true, the algorithm has found a minimum and therefore no weight update is needed. The following equation demonstrates the process of identifying the step determination:

Equation

With being the step, being the update value and being the gradient of the error for all of the training samples as is the current epoch.

Champandard shows that the following equation can be used to calculate the new update value:

Equation

With and being constants with . This means that if the gradient is still going in the same direction, the step size is increased, and that if the gradient changes direction, the step size is decreased. If neither of these criteria match, the step size is left alone.

### 2.7.3 The Random-Minimum Bit Distance Gram-Schmidt Method

Many methods to speed up the training time of multi-layer perceptron neural networks have been proposed over the years. The training method that is going to be the main comparison to error back propagation in this project is the RMGS method. The interesting thing about this method is that it uses different techniques to train different layers of the network. However, it only needs one iteration to be trained. Hypothesized by Brijesh Verma (1997) the method makes use of supervised and unsupervised learning for training the output layer and the hidden layers respectively. As stated by Verma(1997) "The proposed solutions are much faster and without local minima because they use direct solution methods". This makes the implementation of the method far more complicated but, once completed, the training time is negligible compared to error back propagation. The Next section will go into more detail on how the RMGS training method works.

# 3 Random-Minimum Bit Distance Gram Schmidt Training Method

The RMGS method is by far the most complex of the training techniques used in this project, however this pays off with the low training times.

There are a number of equations used to train the network using this method, to make the explanation of the technique clearer, the individual equations will be explained and then referenced in the overall explanation of the RMGS method.

### 3.1 Minimum Bit Distance

The first of which is Minimum Bit Distance (MBD) as seen below (Verma 1997):

Equation

This is very simply a measurement in vector similarity. *X* is the input vector and *w* is the weight vector for the neuron, *n* is the number of neurons in the layer and *i* is the current neuron. This equation simply takes the magnitude of the vector created after the weight vector is taken away from the input vector. To make sure that similar vectors do not give the same output the value of the equation before taking the square root is multiplied by the current neuron divided by the total number of neurons in the layer.

This is important as described in the paper by Verma, some vectors with similar values in different positions may give the same output without this extra multiplication.

### 3.2 Modified Gram-Schmidt Process

The next process we need to cover is the Modified Gram-Schmidt method. This is used to get the *QR* decomposition of Matrix *X*, as the *Q* and *R* matrices can then be used to solve linear equations such as:

Equation

The pseudo code to calculate the *QR* decomposition from (<http://www.math.iit.edu/~fass/477577_Chapter_4.pdf> ) is as follows:

*for i = 1 : n*

*Vi = Xi*

*end*

*for i = 1 : n*

*Rii = ||Vi||2*

*Qi = Vi/Rii*

*for j = (i + 1) : n*

*Rij = Qi ∗ Vj*

*Vj = Vj – Rij\*Qi*

*end*

*end*

*\*\*\*\*\* will write this out in a Photoshop file and insert it as an image ^\*\*\*\*\**

\*\* edit some of the references below to all be “equation letters”\*\*

Firstly, the *X* matrix is copied into the *V* matrix for further calculations.

Next for each column in the matrix:

Initially, the position *ii* in the *R* matrix is set to the magnitude of vector and the column vector *i* in the *Q* matrix is set to divided by (the magnitude of ).

Then for each remaining column in the matrix: the position *ij* in matrix *R* is set to \* and then the value of  *\**  is taken away from .

Once this has been completed the output is the *Q* (orthonormal) matrix and the *R* (upper triangular) matrix are produced and can be used to solve linear equations.

In this project they are used to solve the over-determined system of equations:

Equation

The following must be done using the *Q* and *R* matrices from the Gram Schmidt method to solve this.

Firstly *y* must be calculated by multiplying the transpose of *Q* by *net*:

Equation

Then since the upper triangular matrix *R* has already been calculated, it can be used to find the weight vector by back substitution:

Equation

And the weight matrix has been solved.

The RMGS method makes use of all of these methods in the training of the network. It trains the network as follows:

\*\*\* ADD A DIAGRAM \*\*\*

1. The weights of the first hidden layer are set to small random values between -0.5 and 0.5.
2. All of the input training sets are passed through this layer and a matrix of the ouputs is created.
3. The weights of the second hidden layer are set to input vectors from the training data.
4. Using the MBD method the matrix is then passed through the second hidden layer column by column and another matrix is created with the outputs of the layer.
5. A linear system of equations is then created for the output layer using the desired outputs and the second hidden layer output matrix.
6. Using the modified Gram-Schmidt method the *QR* decomposition of the second layer output matrix is found.
7. For each neuron in the output layer
   1. Using the previously discussed method of solving the linear equations, the weight vector for the neuron is calculated
8. Steps 6 and 7 are then repeated on the second hidden layer to get the actual weights of the second hidden layer using the output from the first hidden layer as *X* and the second hidden layer as *net*.

Once the network has been trained, it can be used the same as a regular training method.

This training technique really stands out compared to the other two techniques discussed as most notably; it only takes one iteration to train the entire network. This provides a huge advantage to using the technique as it allows huge data sets to be used for training that will take minutes not hours to fully train the network. However with this comes a disadvantage in the fact that it is noted by Verma (Verma 1997) that in some scenarios the accuracy of the technique falls in comparison to its longer more rigorous competitors. This issue only really becomes a problem when dealing with something that has to be accurate for a serious reason, such as detection of a fault in an oil pipeline. In a game scenario, the accuracy of the network can be lower; this will provide unpredictability and most importantly, will allow the player to compete with the AI as it may make a mistake. The development of this project has been aimed at allowing the player the ability to actually compete with the AI as well as allowing the training techniques to be able to compete with each other. This will be discussed in full in the next section.

# 4 Methodology

The majority of the practical work completed for this project was aimed towards accurately showing the effectiveness of each of the MLP training techniques described in the previous section. The following tasks had to be completed in order to achieve this objective:

* Develop a game to serve as a testing ground for the training techniques.
* Implement a basic polymorphic MLP framework to accommodate any training technique to be implemented.
* Implement the EBP training method.
* Implement the RPROP training method.
* Implement the RMGS training method.
* Develop a script to generate training data for the neural network.
* Train each method.
* Test each training method.
  + Performance Testing (Quantitative tests).
  + Player Testing (Qualitative tests).

## 4.1 The Game Application

The focus during the designing of the game was to make sure that it would be able to test the effectiveness of each MLP training technique fully. After reading of Jeff Hannan’s implementation of an MLP to control the driving in Colin Mcrae Dirt 2 (2005, it was apparent that a top down racing game would provide the perfect environment for such a test, as the more inputs and outputs that the network contains; the harder it will be for the training technique to train the network to a low error rate.

The game was developed using the Games Education framework (Grant Clarke) with the Box2D physics engine (Box2D) to provide accurate physics for the game environment. These were chosen as it would allow for the use of the C++ programming language, and allow for a good implementation of the MLP network from scratch.

The application was designed to allow the track to be generated using a text file, this was put in place to allow for the easy creation of many tracks. It works by firstly reading the entire file, which should have a grid of 40x40 using ‘#’ to define a generic barrier a ‘-‘ to define an open space and letters or numbers to define waypoints through the track. The waypoints work by defining a small letter to mark one side of the waypoint and a capital letter to define the other side, this allows for the use of the same small letter in the creation of multiple waypoints as shown here:

\*\*\* Picture of txt file \*\*\*

Following the grid, the positions of all of the capital letters and the designated “capital numbers” must be noted below the grid in the text file in the order that they will be on the track, this allows for the ordering of the waypoints. Once all of the data has been read in from the text file, the track is generated and all the barriers and waypoints are made and ordered.

\*\* image of the track in game \*\*

The design of the game allows the player to choose which training technique they wish to race against, or allows them to spectate a time trail of the training technique. The time trail option was implemented to allow for easy quantitative testing which will be discussed later.

\*\*SCREEN SHOT OF TITLE MENU WITH SELECTIONS IN PLACE\*\*

Once the player has chosen their preferred opponent, they will be put into game with a countdown before the race starts. Once the countdown completes they will be able to control their car using the WASD keys, W to go forward, S to go backwards with A and D controlling the wheels to steer left or right respectively. If nether A or D is held down the wheels will automatically straighten up with the car allowing the player a much smoother driving experience rather than having to control the exact position of their wheels at all times. In the top right hand corner of the screen, the player will be able to see their current lap time, if they are in first or second position and how many laps they have left in the race. (mention maybe the ability to change laps, currently not implemented)

To avoid any “cheating” the track will have waypoints throughout, that the player and AI must pass before being able to complete the lap. Although the player will have this information, the only inputs that the car will have are as follows: its current angle in comparison to the next waypoint, the current angle of its tires, the distance it is away from each side of the track and its current speed. All of these variables must be condensed into a range of 0 – 1 in order to be passed into the neural network. This is done in a variety of ways for each of the inputs.

For the Angle in comparison to the waypoint, firstly the current angle of the car is compared to the current angle of the waypoint to get the difference in radians. This value is checked by using the modulo of 2 pi to calculate the angle within 1 turn as box2d continues to add to the angle instead of resetting to 0 if it goes above 2 pi. Finally, the value is then divided by 2 pi to get a value between 0 and 1.

The current angle of the tires is far more simple, they are simply divided by pi/2 and 0.5 is added. This is because the tires are limited to pi/4 in either direction, therefore when divided by pi/2 they will give a value between -0.5 and 0.5. Thusly adding 0.5 will bring that value between 0 and 1.

The current speed input is calculated by dividing the car’s current speed by its maximum speed, giving a value between 0 and 1.

The distance to the side of the track variable is calculated using raycasts. As shown in figure \*\* this is done by casting a ray out for each side of the car along the angle of the current waypoint.

\*\*\* insert Image \*\*

This these rays are then added together to get a distance from one side of the track to the other. The length of the left raycast is then divided by this value, thus giving a value between 0 and 1 representing the horizontal positioning of the car with 0.5 being in the centre, 1 being all the way to the right and 0 being all the way to the left.

Once all of these values have been calculated they can be passed onto the MLP. As shown below, the outputs of the network are flags to the directional keys for controlling the car, above 0.5 means that the button is pressed down and below 0.5 means it is not.

All of the training of the MLP occurs before the actual game starts, this means the player will not have to wait around while the network is being trained. To achieve this once a suitable weight arrangement has been discovered through training the network, the weights are saved to a text file and when the game is loaded to play properly the weights of the network will be loaded from the text file.

The network can be retrained by selecting the train option in the menu, this will overwrite the weights that are saved though.

## 4.2 MLP Building and Training

To implement the training techniques, firstly a base neural network class was created in order to keep the application more structurally sound.

Each training technique then builds off this parent class to create the implementation of their training methods.

### 4.2.1 Multi-Layer Perceptron Implementation

To implement each of these training techniques, a base MLP class was created along with two structs to represent a layer and a neuron.

The neuron struct simply contained:

* A set of weights
* An output value
* An error value
* A set of previous weights
* A set for saved weights
* A bias

The set of weights is initialised with a number of weights corresponding to the number of neurons in the previous layer, if the layer is the input layer, the weights are set to null. The set of previous weights are used for gradient decent using the delta rule and the saved weights are used to make sure the best weights are saved if the network over trains and begins to produce a higher error rate. The output value is used to pass the output of the neuron to the next layer. The error value is used to calculate the adjustment of the weights for the next iteration of the training process. Finally, it has been noted that a bias can be used to help train the MLP better, this is used in the RPROP training technique, and is not used in either of the EBP or the RMGS methods.

\*\*\* when EBP is finalised say whether or not it will use bias \*\*\*

The Layer struct contained:

* An integer containing the number of neurons in the layer
* A set of neurons

This laid out the groundwork for the MLP class which was constructed of:

* An integer storing the number of layers
* A set of layers
* Multiple doubles for adjusting the weights and calculating the error of the network
* Multiple Virtual functions including:
  + A Training Function
  + A Testing Function
  + A function for propagating the signal through the network
  + A function to compute the error of the network
  + A function to adjust the weights of the network
  + A function to set the weights in the network to random values
* A functions to set the input signal and receive the output signal of the network
* Multiple helper functions for setting up and dealing with vectors and matrices. Along with functions for dealing with randomness.

This allowed for the MLP class to be a perfect parent class for any of the training techniques. In doing so, this allows for the actual use of the networks once they have been trained to all be identical, therefore minimal changes are needed in running the network after training.

Although they all follow a similar structure, each training technique has several of its own functions and a unique “train” function, these will be discussed in full in the next sections.

### 4.2.2 Error Back Propagation

EBP is the most basic training technique for MLP networks; this implementation was designed to use EBP in its most basic of forms.

The EBP class has a couple of unique functions for instance, a function for computing the error of the network and a function for adjusting the weights of the network, these are called from the training function.

The “train” function for EBP works as follows:

The network is initialised with all of the weight values set to small random values between -0.5 and 0.5. All of the data in the dataset is then loaded into a 2d array with each row containing all of the input and target output data for that particular training pair. The input data is then passed through the network using the methods discussed in section 2.7.1. The output of the network is then compared to the desired output of the training pair and the error of this comparison is calculated. The error of the network is then used to calculate the error is the propagated backwards to calculate the error of each neuron in each layer. This error value is then used to adjust the weights of each neuron with the following calculation:

\*\*\* add weight adaptation calculation \*\*

This process is repeated for every training pair in the dataset multiple times. Each iteration the current error of the network is compared to the lowest error of the network. If the lowest error is greater than the current error, the current error becomes the new lowest error and the current weights of the network are saved.

If the current error is calculated to be over 1.3 times the lowest error, the training program exits and the weights of the network are restored to the values of the saved weights.

### 4.2.3 Resilient Propagation

The RPROP training method is also programmed using the base neural network class as a parent however has some differences to the EBP training method. Firstly, this training method uses many more 2D arrays than any of the other techniques. This is because rather than editing the base neuron to store more values including gradual weight changes and accumulative errors, these are instead stored in separate 2D arrays for ease of use and to require less wasted storage while using the other techniques.

RPROP’s unique functions include, helper functions to reset the values in an array or 2D array to zero and a function to calculate the error of the network.

The “train” function of the RPROP training method works as follows:

Firstly, the output of the networks is calculated in a similar way to EBP however a bias is added to the output of each neuron this is shown in (equation number here).

After the output of the network has been calculated firstly the gradient term of each layer must be calculated using the following code :

\*\* code snippet \*\*

This shows that by using the derivative of the sigmoid function, the gradient term of the output layer can be calculated easily, and the gradient of the hidden layer can be done by calculating the derivative multiplied by the summed output layer gradient terms.

The hidden layer’s ouput is then multiplied by these terms and accumulated, the layer’s bias is also calculated this way however instead of using the layer’s output a dummy value of 1.0 is used instead. The code snippet below shows this clearly:

This is then repeated for the input to hidden weights.

Once all of these gradients have been calculated they must be used to adjust the weights and biases of each neuron.

For adjustments of the weights the following pseudo code demonstrates how to decide which way to adjust the weights.

\*\*\* insert pseudo code on RPROP here \*\*\*

This process can be repeated for each neuron’s bias too.

Once all of the weights and biases have been updated the process is repeated until the number of iterations reaches the number defined.

The function also saves the lowest weights of the network similarly to the EBP implementation.

### 4.2.4 Random Minimum-bit distance Gram-Schmidt

* MBD
* Gram Schmidt
* Linear Equations

There are two extra calculation functions in the RMGS training method; these are the Minimum Bit Distance function and the Modified Gram-Schmidt function.

The minimum bit distance function takes in three 2D arrays and an integer, the training data, the output of the first hidden layer and an empty 2D array to be filled with the output values of the minimum bit distance calculation. The integer simply contains the number of training pairs in the dataset. The function then sets up the weights as described in section 3, it gets the weights by diving the number of training pairs by the number of hidden neurons and takes every input pair that is a multiple of that number. This is shown clearly in figure \*\*\*\*

The function then feeds the outputs of this function into the empty array and exits.

The Modified Gram-Schmidt function takes two 2D arrays and two integers, the integers define the number of training pairs and the current layer that the gram-schmidt method is being performed on. The first array contains the outputs of the current layer and the second array defines the expected outputs of the next layer. Figure \*\* demonstrates this clearly.

Once the function has performed the modified Gram-Schmidt method on the input array, the function also solves the linear equations and sets the weights of the next layer. This process is described in section 3.

The Training function of the RMGS method works as follows:

Firstly since the network is trained in 1 iteration there is no need to have multiple loops of the training process. It does work in kind of a batch process, as all of the training data is passed through the network at the same time before the weights are changed.

In this implementation the flowing layout is used:

A 3-layer MLP network with one input layer, one hidden layer and one output layer. The 4-layer method was tried however; it would not provide adequate values after training.

The hidden layer’s weights are all set to small random values between 0.5 and -0.5. The training data is then loaded into a 2D array as in the previous training techniques. All of the input data is then passed through the first hidden layer using equations (add equation numbers here) and put into a new 2D array. Since there is no second hidden layer, there is no need for the minimum bit distance calculation, thus the outputs from the first hidden layer are used in the modified Gram-Schmidt function. Once this function has been run, the training process is complete, the weights of the network are then saved to a designated text file for loading when the game is going to be played.

### 4.2.4 Training Data Creation

Training an MLP network can take huge amounts of data, thus to create quality training data in an efficient way, a python script was developed in order to generate mass amounts of correct data for the application.  
This consisted of a number of loops and if conditions to generate a text file stating desired outputs of the network depending on the place in the loops. During this generation of desired output values, after many iterations of training data, some “noise” was added to the inputs as it has been stated by (insert citation to noise in mlp network here) that adding noise to the training data increases the MLP’s ability to generalise to data that it may receive from the application.

## 4.3 Data Collection

Training Times

Computational Performance

Time trials

Survey collection method

# 5 Results

Training Times

Time trial results

Human Participant Survey Results

Computational Performance

# 6 Discussion

Discussion of Time Trial Results

Discussion of Human Racing Results

Discussion of Computational Performance Results

Final Comparison of Training Techniques

Critical Evaluation of the Solution and Application

# 7 Conclusion

Possible Future Research

# Appendices

# List of References

# Bibliography

If required