|  |
| --- |
| 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

# Introduction

Artificial intelligence 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 such as rule based systems; in which rules of 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 the programmer codes very little of and allows for 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 take 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 start the rise in 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 a MLP neural network will still be as accurate as using 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, MLP networks have the potential to be retrained in real time rather than taking incredible amounts of time to do this.

## Research Question

Can alternative training methods for multi-layer perceptron neural networks be as effective as error back propagation in order to promote their use in games?

## 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.

## Structure

The structure of this document will be as follows: A review of previous implementations in controlling a racing car in a game using an MLP network, and a review of different MLP training techniques. Following that will be a full explanation on how the project was carried out and how it was evaluated. Then the results of the evaluation will be described and presented with a discussion on what this shows to close.

# Literature Review

It has been proven that MLP neural networks 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.

## AI and Games

## Artificial Neural Networks

## 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 the one of the first popular game to apply machine learning to a simulation. (AI game dev top 10 influential games)



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

## 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; 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)

An example of an activation function is the “sigmoid” function, which the following equation describes:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Expand on this area!!!!!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## 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 particular paper as 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. 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 taking 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.

## Colin McRae Dirt 2

Colin McRae Dirt 2 is another one of the only 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. The training method that was implemented in this neural network was resilient back propagation (RPROP).

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.

## Training Methods

### 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. Following this, the error is used to adjust the weights of the output layer's neurons; this sequence is then repeated for 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)

### 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 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 like involving too steep of a weight adjustment. 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:

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:

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.

### 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 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 as it is important to understand this.

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

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.

The next equation 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:

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

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 Vi and the column vector i in the Q matrix is set to Vi divided by Rii (the magnitude of Vi).

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

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:

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:

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

And the weight matrix has been solved.

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

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 regularly trained method.

# 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 for any training technique to be implemented on top of.
* 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).

## The Game

Game Application

The game application that was created for this project was a 2D top down racing game.

The tools used to create this game were Microsoft Visual Studio 2015, Box2D and the Games Education Framework, the entire application was written in C++.

To fully test the training methods, a game application was constructed

## Training

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

Training techniques

## 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 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
  + 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.

### Error Back Propagation

* Error Back Propagation
* Resilient Propagation
* RMGS
  + MBD
  + Gram Schmidt
  + Linear Equations
* Training data creation

### RMGS

### Training Data Creation

As the data needed to train an MLP is massive, a method of generating training data was required. To do this a python script was created that

## Data Collection

Training Times

Computational Performance

Time trials

Survey collection method

# Results

Training Times

Time trial results

Human Participant Survey Results

Computational Performance

# 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

# Conclusion

Possible Future Research

# Appendices

# List of References

# Bibliography

If required