|  |
| --- |
| 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 some of the main concepts of academic AI. Ranging 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 that has been an outlier to this trend is 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 achieved. 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, its knowledge is fixed and it can no longer learn at runtime. Online learning allows this kind of dynamic learning, but the majority of learning algorithms for neural networks are unsuitable for this and must be adapted for real-time 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 dynamically 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 dissertation 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 their 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 to make sure the car follows a racing line. 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

Artificial Neural Networks are described as a digital imitation of a biological brain (Buckland 2002). Although they are an imitation of how a biological brain works, they are a long way from being as advanced as an actual brain. Instead they rely on the basic processing features of the brain such as passing information via neurons and synapses to do a multitude of tasks. Such tasks can range from signal processing to much more complex problems like speech generation (Fausett 1994).

ANNs 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 use 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. Each neuron “competes” to have the lowest difference to the input value. When the winner neuron is chosen a process of updating the surrounding vectors takes place. This updates those neurons to be similar in value to the winning neuron and will allow the network to do as the name describes: self-organise.

ART networks use 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 design of this network can be described with the following figure:

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

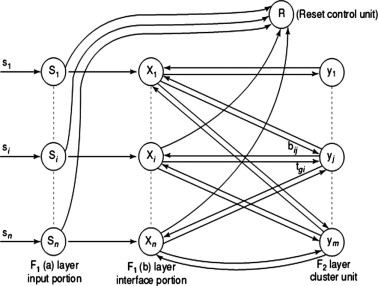


Figure () Representation of an ART network(<http://blog.oureducation.in/art-adaptive-resonance-theory/> Deeppriya Jaiswal 2013)

The network has two modes, training mode and operational mode which can be switched between at any point. This allows the network to learn new knowledge once it has already been trained and thus allows the network to overcome the main problem of ANNs. ART networks not only have feed-forward weights like MLP or SOM networks, but they also have feed-back weights too. These are used to pass a continuously changing input vector between both of the layers in a “cyclic” fashion (King 2016).

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 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)*

A simple description of the network details that values are passed to the input layer, the values are processed by the hidden layers and are returned through the output layer. The neurons that process the values in the layer work by receiving all of the outputs from the previous layer, multiplying them by corresponding weights, then summing all of the resulting values together, and finally, feeding this sum into an activation function to create the output of the neuron.

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 1)

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 paper describes a project is similar to the current investigation, 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 output neurons of the network were simple on/off flags representing whether or not a button on the controller is pressed (Buckland 2004). He also stated that the neural net was only programmed to follow the racing line as any complex manoeuvres like overtaking were scripted instead. This shows that another, simpler AI technique could have been implemented in its place. It was also noted that in this interview Hannan had to constantly test different structures and number of neurons in the network to get it to train properly, this would increase development time and cause the need for an alternative faster training technique to EBP. 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 2

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. Equation 3 demonstrates the process of identifying the step determination:

Equation 3

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 using equation 4 the new update value can be calculated:

Equation 4

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 equation is the Minimum Bit Distance (MBD) as seen below (Verma 1997):

Equation 5

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 process involves using the QR decomposition of a matrix to solve linear equations. QR decomposition is the process of turning a matrix X into its orthogonal matrix Q and it’s upper triangular matrix R. Equation 6 shows the layouts of these matrices:

6

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 (Tools for mathematicians):

7

Equation 7

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*:

8

Equation 8

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

9

Equation 9

And the weight matrix has been solved.

### 3.3 RMGS training

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 processed by this layer and a matrix of the outputs 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 processed by 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: using the previously discussed method of solving the linear equations, the weight vector for the neuron is calculated

Once the network has been trained, the network can be used similarly to a normal network, however the minimum bit distance must be carried out for the second hidden layer on any input.

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 technique described in the previous sections. 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 (game link 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 more of a test it will be for the training techniques.

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 trial of the training technique. The time trial 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 normalised to values between 0 and 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 π 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 π. Finally, the value is then divided by 2 π to get a value between 0 and 1.

The current angle of the tires is far simpler, they are divided by π /2 and 0.5 is added. This is because the tires are limited to π /4 in either direction, therefore when divided by π /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. This is similar to how Yee and Teo used raycasts in their “Spiking Vs Multi-layer perceptron neural networks” paper(Teo and Yee 2015). However, less raycasts are used and they are only used to locate the horizontal positioning of the car in relation to the next waypoint. 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.

The output of the network is interpreted similarly to the AI in Colin McRae Dirt 2 (Ref to Hannan interview) in that each output neuron represents an on/off flag controlling the car. By only giving the AI access to the same controls as the player, it creates a fair race.

All of the training of the MLP occurs before the actual game starts; this means the player would have to wait while the network is being trained. This issue was solved by saving the weights of the network to a text file once training was complete and then having the application load these weights on the initialisation of the networks.

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

## 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 only used in the RPROP training technique for this project. It is not used in EBP as this project is comparing the most basic form of the technique to others and is not used in the RMGS technique because it has not be adapted to constitute the use of 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 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 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. 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. This allows for the network to overcome a local minima when approaching the minimum of the network but it also stops the network from going too far in the wrong direction. This value can be raised or lowered, but most success was found with this particular value.

### 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 below shows this clearly:

\*\* code snippet \*\*

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

To fully compare the effectiveness of each of these networks in a game scenario, multiple tests had to be done.

The first of these tests was a simple timed training test. The data for this test was easily gathered by starting a clock at the beginning of the training process and then stopping it at the end of this process. The test was repeated and an average was taken. The average error of the network was also calculated once training was complete however, any tests, in which the network failed to train correctly, were discarded and restarted.

The second test was a time trial race in which each technique raced 10 laps around the track and an average was taken. Any laps the car failed to complete were discarded and the car was reset to try again.

The final test was a human participant race. This involved having human testers compete against each technique in a two lap race and evaluate their performance by filling out a questionnaire.

For each of the cars there were three questions, a simple yes or no to answer if the player beat the car in a race and then two questions using a level-five Likert scale (Likert Scale ref) to ask the players opinions on the AI’s ability to drive and the challenge that it put on the player. A Likert scale was used as it is a bipolar scaling method that allows the tester to state their reaction to the question on a scale of negative to positive. This kind of scale also allows for excellent statistical analysis on the answers provided by the participants and accurately shows the opinions of the testers.

A precaution that was put in place to stop the testers from having more experience against the final AI was some counter balancing (<https://explorable.com/counterbalanced-measures-design> ). Counter-balancing is used to stop any biases against conditions that are always tested in the same order, so for this project if EBP was always to be tested against first, some of the participants may rate its ability to drive better because they have not had the chance to get better at the game themselves.

Since there are three different techniques, the participants tested the techniques in six different combinations. These were as follows:

1. EBP -> RPROP -> RMGS
2. EBP -> RMGS -> RPROP
3. RPROP -> EBP -> RMGS
4. RPROP -> RMGS -> EBP
5. RMGS -> EBP -> RPROP
6. RMGS -> RPROP -> EBP

The participants that took part in the test were divided among these combinations as equally as possible.

The next section will detail the results of these tests.

# 5 Results

Each training technique was tested in three different ways to provide sufficient information in grading their effectiveness. The first of which was testing the speed at which the network trains, the second was a time trial race around the track and the third put each technique up against a human tester, in which they answered a survey on completion of the race.

## 5.1 Training Results

The difference in training times massively between each of the training techniques. As shown below the average training time of EBP is the longest, RMGS taking the shortest amount of time by far and RPROP being in-between these. Both EBP and RPROP were run for 5000 iterations and because RMGS can only be trained in one iteration, it was only run for one. The dataset provided to the networks had a size of 10000 training pairs. Both EBP and RPROP were structured with the layout of four input neurons, fifty neurons in the hidden layer and four neurons in the output layer for the purposes of this test. The RMGS was structured with four input neurons, four neurons in the first hidden layer and fifty neurons in the second hidden layer and finally four output neurons. This was done as discussed in section 3 the RMGS training technique involves a second hidden layer.

*Table 1: Training time and accuracy*

|  |  |  |  |
| --- | --- | --- | --- |
| Training Technique | EBP | RROP | RMGS |
| Average Training Time  (m:s:ms) | 14:03.34 | 8:23.96 | 0:01.28 |
| Accuracy Once Trained | 84.6% | 78.8% | 70.2% |

## 5.2 Time Trial Results

The time trial results were recorded using the in game lap timer and as stated in 4.3 any laps the car could not complete were discarded. Table 2 can show the results of this test clearly:

*Table 2: Average Lap Time*

|  |  |  |  |
| --- | --- | --- | --- |
| Technique | EBP | RPROP | RMGS |
| Average Lap Time (s) | 41.5 | 43.0 | 37.5 |
| Number of Laps that were reset | 4 | 6 | 3 |

The results show that the fastest around the track on average was the RMGS training technique followed by EBP and then RPROP. The RMGS technique also had the lowest number of laps that had to be reset, thus definitely being the best technique for this particular test.

## 5.3 Questionnaire Results

Ten people participated in the testing of the application. As previously discussed this involved racing against each of the training techniques and answering a questionnaire.

The questionnaire was broken down into five sections.

* Section 1 asked the participant general information about their age, and experience with games in particular racing games. This was to identify whether or not having more experience in playing games would affect the perception of how well each training technique could drive.
* Sections 2 through 4 were identical questions about each of the cars; these were completed at the end of each race so that the experience with the AI was fresh. The questions consisted of asking whether or not the player won the race, how well they thought the AI could drive around the track and what level of challenge the AI gave them in the race.
* The final section asked the participant which car they thought was the best at racing and allowed for any additional comments that the participant may have about the application or the AI to be stated.

Following are the results of the questionnaire.

*Figure() results of question 1 how well did the car drive*

This showed that clearly the best driver was the RMGS training technique as it had an average rating of 3.9 with EBP coming second with an average rating of 3.5 and finally RPROP coming in last place with an average rating of 3.

*Figure() results of question 2 what was the difficulty of the race*

The winner in terms of the race difficult was RMGS as well. An average rating of 3.2 was enough to beat out EBP’s average of 3.0 and RPROP’s average difficulty rating of 2.1.

*figure() results of the final question*

Seven out of the ten participants also rated the RMGS training technique as the best racing car, with EBP delegated as the best by the remaining three. RPROP received no votes for the best racer and stood out as the worst.

Unfortunately, none of the participants on this test had much experience in racing games thus not allowing a conclusion to be drawn in relation to question 2 of the survey.

In relation to how often testers played games in general however, some distinct trends can be seen in Figure ():

Although an initial dip in rating when the number of hours spend gaming increases from 0-5 to 5-10 the RMGS technique has a consistent increase in rating from more experienced gamers. There is no trend shown with EBP as it fluctuates as the number of hours increases and RROP takes a massive decrease in rating initially, climbs to its original rating and then has a steady decline as hours per week increases.

Participant feedback

There were a number of comments left in the participant feedback section, with the most prominent mentioning that the player controlled car had very sensitive controls.

The main consensus in terms of feeback related to the AI controlled cars was that when they crashed, they were stuck and the majority of the time could not correct themselves, thus making the race easier to win.

It was also noted that if the player was particularly good or bad at the racing game, it was difficult to actually grade the AI’s car as it could not be seen for the majority of the race.

# 6 Discussion

This section will firstly discuss the results of each test in more detail and analyse what they mean in terms of the effectiveness of each technique. It will then give a final comparison of the techniques and what this means in terms of this project. Finally, a critical evaluation of the entire application, and the approach to creating it will follow.

## 6.1 Training Times

As shown in section 5.1; the RMGS training technique was by far the outlying fastest training time; this comes about through a number of ways however. The number of iterations being the stand out reason why it has such a fast training speed. Its accuracy does take a massive set back from this however, as seen in table 1 it is almost 15% less accurate than EBP and around 8% less accurate than RPROP.

Some problems were noted during the running of this test.

A big issue discovered with all of the techniques is that using the training data provided: there is no guarantee for them to train properly. This becomes gives this test a slight misleading representation of how well each technique did. This is especially noticeable in using RMGS to train the network, in which it appears to be the case that the random weights of hidden layer one completely dictate whether or not the network is going to train correctly. This resulted in the network having to be retrained many times before the results given were adequate to drive the car around the track.

RPROP also had some problems in the training test. If the technique made one incorrect step in the process of training, it would go off completely. This means that although the network would be initially approaching the minima, the training technique would take a wrong step and suddenly within 100 iterations, it would have gone from a mean squared error of something as low as 0.30 to a mean squared error of over one. This would not be nearly accurate enough to complete the track and thus training would be rendered useless.

EBP on the other hand was very good at achieving accurate results almost every time. It would occasionally train incorrectly by the chances were minimal in comparison to the other two techniques.

To conclude which training technique was the best through just this test would be very difficult as although RMGS was clearly the fastest, it had the most issues with training correctly, and the opposite was true for EBP.

## 6.2 Time Trials

The time trial results as shown in table 2 clearly show that the training technique with the fastest racing time was RMGS followed by EBP, with RPROP in last. If you compare this result to the results of the training times in 5.1, it does not make sense that the RMGS technique was the fastest around the track as it was the most inaccurate. However, this could also mean that it has an advantage. At sharp corners on the track, the training data dictates that the car should slow down and take the corner carefully, with a higher inaccuracy the RMGS technique could ignore the level of slowing down that the other more accurate techniques do and thus being faster around the track.

This however does not ring true for the RPROP training technique as although it was more accurate than the RMGS technique and less accurate than the EBP technique it had the slowest racing time.

EBP had the middle position in terms of speed around the track.

An issue that arose during this test was the cars getting stuck and being unable to correct themselves. This problem is most likely down to the fact that none of the training methods were above 85% accuracy and they were unable generalise what to do when stuck against a wall even though the training data had detailed how to handle this particular situation. To give the network complete control of the driving experience as this project has done does bring up some issues involving what to do in the scenario of a crash. To get the network to learn to reverse and turn left at an angle that is very similar to one that would make the network want to drive forward and turn right is very difficult as it has to be able to generalise what to do in-between these points. In the case that the cars became stuck up against a wall or on a corner, the majority of the time they were either trying to accelerate and reverse at the same time resulting in no movement or they were doing the opposite and neither accelerating or reversing, also resulting in no movement. This problem could have been solved by adding a script to make the car reset itself on the track once it became stuck, similarly to Colin McRae Dirt 2 as discussed in section 2.6, but the aim of this project was to get the network to be in complete control of the car. This does raise the question of whether or not the network should be in full control of the car. As the ability to perform complex manoeuvres like resetting from a crash or overtaking would require extra inputs to the network.

## 6.3 Questionnaire Results

The results of the questionnaire are quite definitive in that the RMGS technique proved to be the best when racing against participants. With seven out of ten people rating the technique the best out of all three and the highest average ratings in both questions relating to how well the cars performed at 3.9 for ability to drive and 3.2 for difficulty.

The testing also clearly outlined that in the views of the participants, the number of hours that they spend on gaming had little effect on their rating of EBP, showing that the consensus was that it was a decent driver. RPROP varied in terms that generally as the number of hours increased the performance in the view of the participant decreased, the exception to this being the dip from 3.6 to 1.0 between the first two categories returning to 3.6 in the third category. RMGS on the other hand other than an initial drop in rating from no gaming hours to some gaming hours showed that the more hours that the participant plays per week the higher they thought of the RMGS car’s ability to drive. Generally the more that a person plays games the better they are at criticising the AI of the game, thus showing that the RMGS technique was critiqued better and the RPROP technique was critiqued to be worse.

\*\* ^^^^ REWRITE THIS SECTION ^^^^ \*\*

An issue that arose during training was that because the player was racing against the AI instead of watching it, they could either pull ahead or fall behind the AI to the point that they were unable to actually watch the performance of the AI’s car. Thus, they would have to guess based on how the first few corners of the race went.

## 6.4 Final Comparison

It is evident that the RMGS technique was the best performing technique in all stages of the testing, other than its accuracy and its training consistency. The EBP would definitely place second in this comparison as its ability to train to high accuracy and a consistent success rate along with being the second best rated in the Questionnaire results and having the second fastest in lap time. The RPROP training technique would be a definitely last place as it received the worst results in both the time trial and questionnaire results along with coming second in both training time and training accuracy.

## 6.5 General Discussion

These results do show that some faster training techniques can be just as good, if not better than the standard Error Back Propagation technique in a game scenario. Resilient propagation was not a good example of this and failed quite impressively at every stage of the testing. It was rated the worst in every test other than the speed-training test in which it came second. It was quite troublesome in this process too as discussed in section 6.1 it had an inability to converge on a minimum effectively. The Random Minimum-bit Gram-Schmidt method on the other hand, shined through in comparison producing the best results in every test except for the accuracy of the network. This shows that in a game scenario a technique such as the RMGS technique could be used instead of EBP. However, the randomness in success of training the network creates some uncertainty in using this technique. If it was to be used for an application in which accuracy was extremely important then this technique would be completely unsuitable, as stated in 6.1 the technique is completely reliant on the random weights of the first hidden layer to produce effective results. Although on paper this may seem like a massive negative factor in using the network, if it is used in a game scenario this could actually be very beneficial. For example in a racing game with multiple opponents, the game designer may want each car that the player is racing against to behave differently to constitute how different drivers would approach driving around the track. This technique would be perfect for creating that kind of unpredictability without any external calculations or coding. The best kind of racing game this would be suited for is something like Mario Kart (ref here) as the game is not aimed at showing realistic driving, in a game like Forza 4 (ref here) this technique would suffer a lot more as the aim of this game is to provide a realistic racing simulation.

In section 1 it was stated that if the RMGS training technique was accurate enough it could in theory be used to retrain a neural network in real time. However, due to its inability to provide consistent training success, it would not be useful for this purpose. This issue would most likely make any AI that it was controlling work completely randomly as the outcome of the tests showed in 6.1 that although fast the chances of the training to be unsuccessful were high.

## 6.6 Project Findings

This project clearly demonstrates that some alternative training techniques have the potential to replace error back propagation in the training of a multi-layer perceptron neural network in a situation where high accuracy of the network is not necessary.

It also found that there is potential for a MLP network to have the majority of control of the AI in a game.

This project has also found that using the RMGS training technique to update a neural network in real time would not be feasible.

## 6.7 Critical Evaluation

The application as a whole was very successful in terms of trying to prove that alternative training techniques can compete with EBP in a game scenario. Through the testing that was undertaken it clearly showed that the RMGS technique outperformed EBP in nearly every test. RPROP on the other hand was unsuccessful in competing with either of the other two training techniques and was definitively the worst of the three in this game scenario.

Although successful, there are a number of improvements that could have been made to the application in general. One of which being the control scheme for the player’s car. It was brought up multiple times in the feeback section of the questionnaire that the controls for the Car were far too sensitive and it was difficult to correct your driving once you had crashed. This could be corrected by reducing the angle at which the car tires can turn to, however this would impact the performance of the AI as it would need to be retrained to suit this new level of turning.

It was also stated that it was sometimes difficult to tell which way round the car actually was; as in the testing version of the game there was very little art and the car was only defined by box sprites for the body and the wheels. This was fixed for the final version of the application by defining a sprite for the car’s body instead of just having a box.

In terms of the AI, multiple things were discovered throughout the development of this project. The first being that the development time of building a multi-layer perceptron network from scratch is fairly long and laborious especially if the training technique is not going to be EBP. Another in that to create a general parent class for any training technique many unnecessary values were being created in some of the techniques. For instance, since the RPROP technique was the only one that contained a bias, the base neuron class had to contain a bias; therefore every neuron that was created in either of the other two techniques was allocating extra space that was not used. This could be solved by creating a different set of neurons for each of the techniques but that would mean the base class would need to be far more complex than it already is.

The training technique that took the longest to implement by far was the RMGS technique, this is due to a number of reasons. The first being that the description of the network in the original paper was quite vague and thus a lot of guess work had to take place in order to get the technique to work. Another issue in getting it to work was that there was some fairly complex mathematical processes that take place for instance the modified gram-schmidt process. This meant that extra development time was used researching these other techniques and building testing applications to debug each section of the technique. The development time of this particular technique was very long due to much of the guesswork that had to take place. If the technique had been more clearly defined initially then the time taken to program this technique could have been much shorter. This paper has made an attempt to clearly state what needs to be done to implement the RGMS training technique and should provide a clearer description for any future implementations.

The next section will detail what has been concluded by the completion of this project and will state any future work that could be carried out as a continuation of this investigation.

# 7 Conclusion

Possible Future Research

# Appendices

# List of References

Tools for mathematicians <https://books.google.co.uk/books?id=V7dUONBrHF0C&pg=PA315&lpg=PA315&dq=modified+gram-schmidt+algorithm+for+solving+linear+systems+of+equations+how?&source=bl&ots=EZcbZqmpJH&sig=vZFb0mxmW1tjXoZS888StB5ev5k&hl=en&sa=X&ved=0ahUKEwjbgZuWoNfPAhVGIsAKHdK2C_s4ChDoAQhcMAk#v=onepage&q=modified%20gram-schmidt%20algorithm%20for%20solving%20linear%20systems%20of%20equations%20how%3F&f=false>

# Bibliography

If required