# An Evaluation of Fast Multi-Layer Perceptron Training Techniques for Games

David G.Robertson
Department of Arts, Media and Computer Games
Abertay University
DD1 1HG, Dundee,
Scotland
E-mail: drrobo6@googlemail.com

**KEYWORDS**
Multi-layer perceptron - MLP
Error Back Propagation - EBP
Resilient propagation - RPROP
Random-Minimum Bit Distance Gram-Schmidt - RMGS
Artificial Neural Network - ANN
Artificial Intelligence - AI

**ABSTRACT**

Despite the rise of Artificial Intelligence (AI) in games leading to the adoption of many academic techniques, multi-layer perceptron (MLP) neural networks have bucked this trend and have rarely been used in a game scenario. This is normally due to long training and development times using the standard error back propagation (EBP) training technique. The purpose of this investigation was to compare alternative training techniques to EBP in order to see if they can be used to promote the use of MLP in games.
The application created to serve this purpose was a 2D top down racing game with three different training techniques to control the AI, including EBP, resilient propagation (RPROP) and Random-Minimum Bit Distance Gram Schmidt (RMGS), in which, each training technique was put through three tests.
Through these tests, it was shown that alternative training techniques, although not as accurate as EBP, reduce the training time drastically. The tests also concluded that in a racing game scenario the alternative techniques could also compete with EBP, with the RMGS training technique being the best in every test except accuracy.
This project has shown that MLP could easily be utilised in game scenarios using these alternative methods and would not require the lengthy training times of EBP.

**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 rules for the AI 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 permanent place in games.
There have been a number of attempts to implement ANNs into video games, but have only been used in very niche parts of games to do something that a far less complicated game AI technique could have easily 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 adaptation, but the majority of learning algorithms for neural networks are unsuitable and must be revised 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 updated training data. Hence 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 its famously long 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 paper 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 win. Since this method only takes one iteration to train the network, there is potential for

MLPs 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 paper aims to prove that feasibility.

## LITERATURE REVIEW

It has been proven that MLPs can control a car in a racing game. For example, Colin McRae Rally 2 utilises this for its game AI to make sure the car follows a racing line. However, training MLPs takes a lot of time and thus they are rarely used in games.

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), Resilient Propagation (RPROP) and EBP thoroughly.

### Multi-Layer Perceptron Network

The 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 4 shows an example of how an MLP network looks.

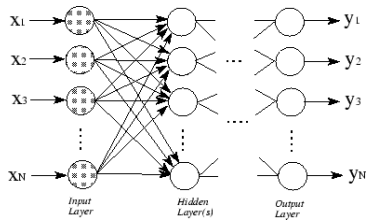


Figure 1: An example of a Multi-Layer Perceptron Neural Network (Kawaguchi 2000)

A simple description of the network is 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 each layer (except the input 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 5 shows the process for each neuron:

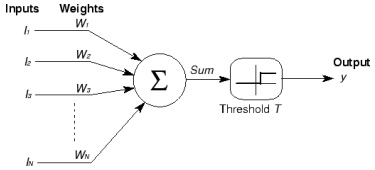


Figure 2: Symbolic Illustration of Linear Threshold Gate (Kawaguchi 2000)

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 input passes a certain value to more complex functions such as the "sigmoid" function", which the following equation describes:

$$S(t) = \frac{1}{1+e^{-t}} \quad (1)$$

This allows for the neuron to always activate, but with varying output values. Figure 6 shows the activation curve of the sigmoid function:

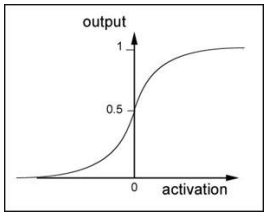


Figure 3: The sigmoid activation function (Buckland 2016)

The threshold activation function is used in combination with some newer techniques to create spiking neural networks, which aim to more accurately model the brain.

### Training Methods

*Error Back Propagation*

The error back-propagation method is the most common training method for MLP 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). 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 shown in equation 2:

$$E = \frac{1}{2}(target - output)^2 \quad (2)$$

The error is then used to calculate a delta value. This delta value is used to firstly adjust the weights of the output layer; it is then passed backwards to the hidden layer, which will calculate a delta value for each neuron in the layer for weight adjustment. This process is repeated for all of the hidden layers, working backwards from the output layer until the entire network has been corrected. The reason for the delta being calculated at the end of the network and being passed backwards is that any interaction with the network is only available via the input and output layers, therefore the error can only be calculated once the data has been passed through to the output layer. The entire training process is repeated for the all of training data multiple times until the calculated error reaches a minimum (Bourg and Seemann 2004). This process is known as the *delta rule* and *Back Propagation*.

EBP is known as the steepest decent method (El-Sharkawi, Marks and Weerasooriya 1991) for finding the minimum of a function; this is due to it using the optimisation method gradient decent. The process of gradient decent is a way of

reaching the minimum of a function by updating the parameters in proportionally to the gradient at the current point. (Ruder 2016)
Another training method that relies on gradients heavily is Resilient Propagation.

*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 and Braun 1993). Reidmiller and Braun found that their training method outperformed the classic EBP with ease and other training techniques such as "Quick Propagation" and "SuperSAB". RPROP works somewhat similarly to EBP 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 at each 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 too large 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. Equation 3 demonstrates the process of identifying the step determination:

$$\Delta w_{ij} = \begin{cases} -\Delta_{ij}(t) & if\ \nabla E(t) > 0 \\ +\Delta_{ij}(t) & if\ \nabla E(t) < 0 \\ 0 & otherwise \end{cases} \quad (3)$$

With $\Delta w_{ij}$ being the step, $\Delta_{ij}$ being the update value and $\nabla E(t)$ being the gradient of the error for all of the training samples as $t$ is the current epoch.
Champandard shows that using equation 4 the new update value can be calculated:

$$\Delta_{ij} = \begin{cases} n^+ * \Delta_{ij}(t-1) & if\ \nabla E(t) * \nabla E(t-1) > 0 \\ n^- * \Delta_{ij}(t-1) & if\ \nabla E(t) * \nabla E(t-1) < 0 \\ \Delta_{ij}(t-1) & otherwise \end{cases} \quad (4)$$

With $n^+$ and $n^-$ being constants with $0 < n^- < 1 < n^+$. 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*
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 "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 and resilient propagation as it trains the entire network in one epoch.
There are two mathematical processes employed, which are: the Minimum bit distance method and the Modified Gram-Schmidt process. The layers that both of these techniques are used to train are shown below in figure 4.
*W1* is trained by setting all of the weights to random small values. *W2* is trained via the minimum bit distance technique and *W3* is trained via solving linear equations using the modified Gram Schmidt method.

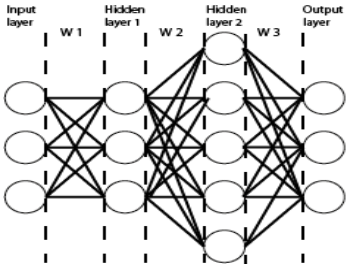

*Figure 4. Structure of the RMGS network*

The Minimum Bit Distance (MBD) as seen below is a simple measurement in vector similarity (Verma 1997):

$$MBD(x,w) = \left\|\underline{x} - \underline{w}\right\| = \sqrt{(\sum_{i=1}^{n}(x_i - w_i)^2 \frac{i}{n})} \quad (5)$$

$\underline{X}$ is the input vector and $\underline{w}$ 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.
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 type of processing is part of a new generation of MLP networks called Deep Learning Neural Networks (Marr 2016) in which multiple layers use different processing calculations and different activation functions to calculate the output of the network.

**METHODOLOGY**

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 use of an MLP to control the driving in Colin McRae Rally 2 (CodeMasters 2000), it was apparent that a top down racing game would provide the perfect environment for such a test.
The game was developed using the Games Education framework (Clarke 2017) with the Box2D physics engine (Box2D 2017), which provides accurate physics for the game

environment and allows for very simple and effective ray-casting calculations. 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 inputs for the network that the car uses 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. Multiple techniques and calculations are done on different variables of the car to allow this to happen.

To get the value of the Angle in comparison to the next waypoint: firstly, the current angle of the car is compared to the angle of the waypoint to get the difference in radians. This value is checked by using the modulo function of $2\pi$ to calculate the angle within 1 turn. This is because 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 simpler, they are divided by $\pi/2$ and 0.5 is added. This is because the tires are limited to turning $\pi/4$ in either direction, therefore when divided by $\pi/2$ they will give a value between 0.5 and -0.5. Thus 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 (Yee and Teo 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 5 this is done by casting a ray out for each side of the car parallel to the angle of the current waypoint. This is represented by two red lines being drawn from the centre of the car outwards at the angle of the waypoint, red boxes are drawn where the rays collide with the barriers on the edge of the track.

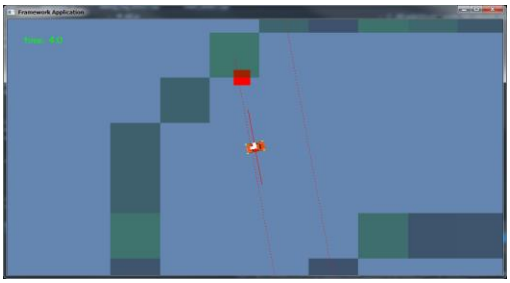

Figure 5: AI raycasting from the centre of the car

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 into the MLP.

The outputs of the network are flags mapped to the directional keys for controlling the car, an output above or equal to 0.5 means that the button is pressed down and an output below 0.5 means it is not.

The output of the network is interpreted similarly to the AI in Colin McRae Rally 2 (Buckland 2004) 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.

## 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, the following is the results of the first two tests.

### Training Results

The difference in training times (see Table 1) varies 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 number of iterations that EBP and RPROP were run for was decided through trying to give EBP and RPROP enough time to have trained properly, but not be over trained and unable to generalise. 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 because 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% |

### Time Trial Results

The time trial results were recorded using the in game lap timer and any laps the car could not complete were discarded. Table 2 shows the results of this test:

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.

## CONCLUSIONS

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.

This project clearly demonstrates that some alternative training techniques have the potential to replace EBP in the training of a MLP neural network in a situation where high accuracy of the network is not necessary.

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

## FUTURE WORK

There are many directions that future work on this project could be taken.

Firstly, testing the AI's performance on other tracks would be a very interesting test of the actual effectiveness of the training. Since the networks have been trained on data that should be transferable to other tracks as long as they are set up similarly to the track used.

The AI implementation in this application has been developed to be modular, thus allowing for easy implementation into more racing cars in the game or other applications by only adding a few lines of code. Therefore, implementing the use of the RMGS training technique in another situation would also be a very interesting direction this work could be taken. Especially into different game types like platforming and fighting games as these require completely different judgement by the AI and would likely test the effectiveness of the technique very well. The way in which all of the training techniques have been implemented in this particular application means that they can easily be plugged into another application and as long as training data is provided, they would be able to take control of the AI.

## REFERENCES


Bourg, D. and Seemann, G. 2004. *AI for Game Developers*. Sebastopol, United States of America: O'Reilly Media, Inc.

Champandard, A. 2004. *AI Game Development*. Indianapolis, United States of America: New Riders Publishing.

Charles, DK and McGlinchey, S. 2004. The Past Present and Future of Artificial Neural Networks in Digital Games. *Proceedings of the 5th international conference on computer games: artificial intelligence, design and education*. Pp. 163-169.

*Colin McRae Rally 2.0*. 2000. [Computer game]. Codemasters.

El-Sharkawi, M. Marks, R. Weerasooriya, S. 1991. Neural Networks and Their Application to Power Engineering. *Control and Dynamic Systems V41*. San Diego, California. United States of America. Academic Press.

Reidmiller, M. Braun, H. 1993. A direct adaptive method for faster backpropagation learning: the RPROP algorithm. *IEEE International Conference on Neural Networks*. 1. pp. 586 – 591.

Verma, B. 1997. Fast Training of Multilayer Perceptrons. *IEEE Transactions on Neural Networks*, 8(6), pp. 1314-1320.

Yee, E. and Teo, J. 2011. Evolutionary Spiking Neural Networks as Racing Car Controllers. *11th International Conference on Hybrid Intelligent Systems (HIS)*, pp. 411-416.


## WEB REFERENCES


Buckland, M. 2004. *Interview with Jeff Hannan*. Avaliable from: http://www.ai-junkie.com/misc/hannan/hannan.html [Accessed 23 September 2016]

Buckland, M. 2016. *Sigmoid*. [Online Image] Available from: http://www.ai-junkie.com/ann/evolved/nnt5.html [Accessed 7 March 2017]

Clarke, G. 2016. *Games Education Framework source code*. [Source code] Available from: https://github.com/grantclarke-abertay/gef [Accessed 12 September 2016]

Catto, E. 2017. Box2D. (Version 2.3.0). [Software Library] Available from: http://box2d.org/ [Accessed 25 November 2016]

Kawaguchi, K. 2000. *The McCulloch-Pitts Model of Neuron*. [Online Image] Available from: http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html [Accessed 7 March 2017]

Marr, B. 2016. *The difference between Deep Learning, Machine learning and AI*. Available from: https://www.forbes.com/sites/bernardmarr/2016/12/08/what-is-the-difference-between-deep-learning-machine-learning-and-ai/#1dc7a8a026cf [Accessed 20 April 2017]

Neilson, M. 2017. *How the back propagation algorithm works*. Available from: http://neuralnetworksanddeeplearning.com/chap2.html [Accessed 2 April 2017]

Ruder, S. 2016. *An Overview of Gradient Decent Optimisation Algorithms*. Available from: http://sebastianruder.com/optimizing-gradient-descent/ [Accessed 30 March 2017]

Standford University. 2000. *History: The 1940's to the 1970's*. Available from: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html [Accessed 10 October 2016]