**1. Introduction**

The use of advanced AI techniques in computer games has been growing steadily, along with the expectation of ever more realistic and immersive gamming experiences. As computer graphics within games continually reach new levels of realism, so too must the AI techniques deliver equally realistic and believable behaviours so not to break the immersion. One of the core aspects that determines whether a game character is acting intelligently is their ability to make intelligent decisions based on their current circumstance. The most used AI techniques by game developers for controlling character decision making are ones that allow a high level or accuracy and control over their actions, such as a finite state machine or fuzzy logic controller. One setback to such approaches though comes when the circumstances and interactions within the game world become increasingly complex. Such complexity makes it difficult to write the rules for a state machine or fuzzy logic controller that covers all eventualities (Millington, 2019). Another approach is to use machine learning and allow the AI to learn its behaviour through training examples. This is a technique that has been met with concern in the gaming AI community (Rabin, 2013) but success has been had using this approach in some commercial games. One such title is *Supreme Commander 2* (Wargaming Seattle, 2010) in which a neural network was used to control the fight or flight response of AI controlled platoons with reported great effect. One drawback to using a typical supervised neural network is that many training data sets are required for the network to learn. The quality of the training provided also has a significant impact on the performance of the AI, so if training data isn’t available, this might not be a viable option. An alternative approach is the use of an evolutionary neural network, which combines a genetic algorithm with a neural network, enabling an AI to learn in an unsupervised fashion through evolution. The aim of this project is to assess the effectiveness of using an evolutionary neural network to train an AI characters to make intelligent decisions in a basic deathmatch battle scenario. To achieve this, a basic deathmatch-style game has been created in the game engine Unity, in which an AI will be trained using an evolutionary neural network to make decisions based on the state of the game world. The report will begin with a literature review, looking at the components that make up the evolutionary neural network being used, along with the mathematical underpinning. The methodology behind the application will then be discussed, followed by a summary of the results and will finish with some conclusions on the project.

**2. Literature Review**

The primary source reference for the maths in the following section is (Nielsen 2015).

**2.1. Neural Networks**

An artificial neural network (ANN) is designed to mimic the evolutionary characteristics of the human brain. ANNs are comprised of an interconnected network of artificial neurons, capable of learning through pattern recognition. This is what makes them an attractive AI technique in computer games. Giving game AIs the ability to respond and adapt the state of the world around them adds another level of impressiveness and realism to the game. The computer game *Supreme Commander 2* (Wargaming Seattle, 2010) successfully utilised a neural network to successfully control a platoon’s reaction to encountering enemy units (Rabin, 2013). For example, data from an AI character, such as its health, ammo, distance to enemies, etc can be fed into an ANN to form the state of the game world around it. When trained, the ANN can extrapolate patterns from the data and classify the input into one of the output categories provided in the training data. This gives the character the ability to appear as if it is acting intelligently, making reactive decisions based on its current circumstance and surroundings. A similar effect can however be achieved using AI techniques that are both less computationally demanding and easier to implement in code, such as a Finite State Machine or Decision Tree. What sets the ANN apart from other such techniques is its ability to generalise by extrapolating into the grey areas within the training data (Millington, 2019), making their decisions less predictable. State Machines and Decision Trees are accurate and will always act in a certain way, given the same set of circumstances. This also makes them easy to predict, and potentially less challenging once a player has figured out the rules governing the AIs actions.

One of the most common learning methods when using an ANN is the backpropagation algorithm. With backpropagation, pre-existing training data is required to train the network, comprising of many examples of input data that correspond to each of the potential outputs. The performance of the network is dependent on the quality of the training data, so if the training data is poor, too few or inaccurate, the results of the network will most likely also be poor. Another approach is to use a genetic algorithm (GA) to evolve the network. Using this approach requires no pre-existing training data as the network evolves over many iterations of an example scenario. This is the approach that is the focus of this report.

**2.2. Evolutionary Neural Networks**

Research comparing the use of ANNs using backpropagation with evolutionary neural networks when training an AI in an action fighting game has shown that the evolutionary neural network can outperform a backpropagation network (Cho, Park and Yang, 2007). This report explores the effectiveness of using an evolutionary neural network to train an AI character in a close-quarters action combat game setting.

The evolutionary neural network used in this study utilises a genetic algorithm to evolve a multi-layer perceptron neural network. The following section comprises of a basic review of the multi-layer perceptron and genetic algorithm, along with a summary of their mathematical underpinning.

**2.3. Multi-Layered Perceptron**

A multi-layered perceptron (MLP) comprises of a collection of artificial neurons, known as perceptrons. The perceptrons are arranged in layers, where each perceptron is connected to all the other perceptrons in the layers that immediately proceed and follow it. Figure 1 below shows a typical layout for an MLP with three layers.

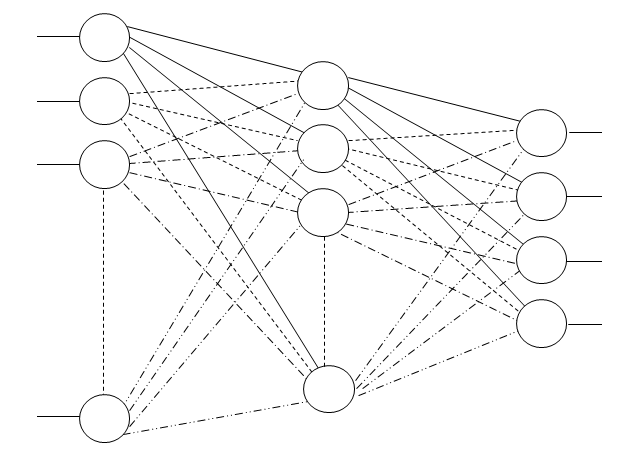


Figure 1: MLP Architecture

The left-most layer is called the input layer. Each perceptron in the input layer takes a value relative to the problem that is trying to be solved as an input. In the context of this application the input layer will be comprised of seven nodes, each one representing a different indicator of the AI characters current state in the game world. These will be their health, magic level, whether health is available, whether magic is available, the number of enemies, if they have an enemy pursuing them and the distance from their pursuer.

The perceptrons in the middle layers, which are referred to as hidden layers, take the output from all the nodes in the preceding layer as inputs and sends a single output value to all the layers in the following layer. Therefore, an MLP is known as a feedforward network. The outputs from the right-most layer, called the output layer, are the networks classification for the given pattern that has been provided as input to the network. In this instance this will represent the state that the AI should be in given its current circumstance.

Each perceptron contains an algorithm to generate its output that will be passed on to the next layer. The output for a perceptron is in the range 0.0 – 1.0. As mentioned previously, each perceptron takes all the outputs from the previous layer as it’s inputs. Each of those inputs also has a weight associated with it. Weights are real numbers that signify the influence that an input will have on the perceptrons output value. The combined total of inputs multiplied by their weights is known as the weighted sum, and can be expressed as the following:

As well as weights for all the inputs, each perceptron also has a bias value. The bias represents a measure of how likely a perceptron is to output a significant value (how close to 1 it will be). A large bias would mean the output will be more influential on the overall output of the network, and a small bias means it would be less so. In order to get the output value into the desired range, the weighted sum plus the bias are passed through a function called the activation function. The activation function can be represented as follows:

Different activation functions can be used to influence the output value in different ways. For the purpose of this application, the sigmoid activation function will be used as it allows small changes in the weights and bias to have a small effect on the output. This change to the weights and bias is what will be used to evolve the network, and in effect, enable it to learn. The sigmoid function is defined as:

Where is equal to the weighted sum plus the bias, so more explicitly, the output of a perceptron with a sigmoid activation function is equal to:

The shape of the sigmoid function can be observed below in figure 2:

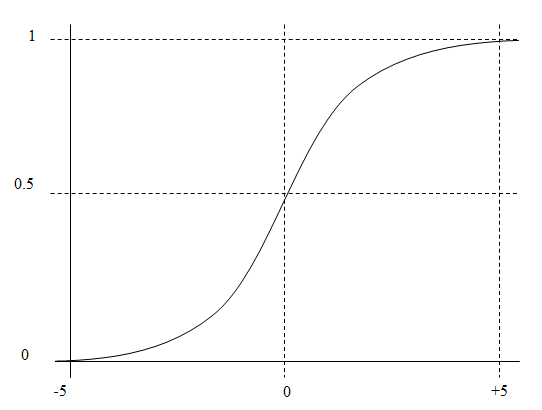


Figure 2: Sigmoid function

From observing the shape of the sigmoid function, we can see how small incremental changes to the weights and bias will affect the output value. These weight and bias values will be tuned in an evolutionary fashion using a genetic algorithm to try and reach an optimal solution.

**2.4. Genetic Algorithm**

A genetic algorithm (GA) is an optimisation algorithm based on the concepts of biological reproduction and evolution. It is a fast and efficient method for searching a large and complex solution space. This makes it well suited for the problem this application is trying to solve, given the vast amount of possible combinations of input data the network will be faced with.

A GA works by generating a population of candidates to solve a given problem. Each candidate contains a chromosome, defining its genetic makeup. Usually this would be represented using a string of characters or an array of digits. In the context of this application, each candidate will be a neural network that is attached to an AI and its chromosome will comprise of its weights and biases. After the population has been initialised it is assessed to see how well each candidate performs at solving the problem. This assessment is called the fitness function. The fitness function is unique to a given problem and is used to assign a fitness score to each candidate. Once candidates have been assessed, members of the population are selected to form the mating pool for the next generation and the rest are discarded. New members of the population are added in a process known as crossover. During crossover, two members of the mating pool are selected, and their chromosomes are merged (or crossed over) to create a new candidate. Once a new candidate has been created it is then subject to mutation, where there is a chance that some of the data that makes up its chromosome will be altered slightly. This process adds some randomness into the population and can prevent the population from converging to a sub-optimal solution. The new population is then assessed, and the process starts over again. This process repeats itself until a good enough solution is found or for a predefined number of generations.

**3. Method**

The application demonstrates how an evolutionary neural network can be used to train an AI to make decisions based on the state of the game world. In order to achieve this, the game engine Unity has been used to create a basic deathmatch style game scene in which the AI can be trained.

**3.1 The Deathmatch Instance**

Two teams of AI characters fight in a deathmatch; the Green team and the Purple team. The winning team is the first to kill all the enemies on the opposing team. The AIs have health and magic levels. When they attack an enemy, their magic level is decreased by one. When they are attacked, their health level is decreased by one. Additional health and magic items are available to collect around the edges of the battlefield. All but one of the NPC’s actions are controlled using a basic state machine. If they have good health and magic levels, they will seek out an enemy opponent and attack. If their magic has ran out and their health is ok, they will find some magic. If their health is low, they will find health. If their health is low and they have an attacker in firing range, they will flee.

**3.2. The learning AI**

One of the NPCs on the green team is a special learning AI. This is the AI that will be trained using an evolutionary neural network to make decisions that will allow it to rival, or better the AIs controlled by the state machine. The learning AI has the same abilities as the state machine AIs, where it can attack a near-by enemy, find health, find magic and flee. The output from the neural network will determine which of those states it should enter at each frame update during the game. In order to assess the learning AIs performance vs a standard AI, the learning AI can be run in control mode. When running in control mode, the learning AI behaves like the rest of the AIs on the map, where its actions are determined by a state machine. This allows metrics to be recorded in both learning and control mode for comparison.

**3.3. Application Workflow**

1. Parameters for the network and genetic algorithm, along with the game settings are first set through a Scene Manager interface in the Unity scene. For the network, the hidden layer configuration can be adjusted. For the genetic algorithm parameters that can be adjusted are population size, mutation chance and mutation strength. Game settings that can be adjusted are battle duration and game speed.
2. neural network instances are created with a user-specified layer configuration, where refers to the population size. Their weights and biases are assigned a small random value between -0.5 and 0.5.
3. deathmatch instances are created in a grid formation in the scene. Each deathmatch instance comprises of a basic square battlefield with some health and magic collectable items scattered around the edges. Six AI characters are placed in the battlefield, three on each team. One of the AIs on the Green team is the learning AI.
4. The learning AI within each deathmatch instance is assigned a neural network
5. The battles in each instance play out for a given period. During gameplay, the state of the game world from the perspective of the learning AI is fed into its network as inputs on each frame update. The data fed into the network is the AIs health and magic levels, if there is health and magic available on the map, how many enemies are present, if they have an enemy pursuing them and the distance to their pursuer. These values are fed into the network’s feedforward algorithm and the output is used to set the AI state. The state will be one of Attack, Flee, Find Health or Find Magic.
6. After the time has elapsed, the fitness of each learning AI is calculated. The networks are then sorted by their fitness level.
7. Once the networks have been sorted, the poorest performing half of the population are removed from the gene pool. Random crossover is then performed on the remaining networks with parents picked at random to make up the missing numbers. During crossover, weights and biases are picked at random from both parent networks to create a new child network to add into the population.
8. Finally, after a new network has been added into the population, it is subject to random mutation of some of its weights and biases. The chance that mutation will occur on a weight or bias is user configurable, along with the upper and lower limit of the possible strength of the mutation.
9. Once all the discarded networks have been replaced, the existing deathmatch instances are destroyed, a new set is created, and the cycle starts again.

**3.4. Fitness Function**

The fitness of the AI is calculated in order to reward good decision making with positive fitness and punish poor decision making with negative fitness. The AI’s decision-making ability is determined by comparing the state that has been output by the network with the input data at each update. A fitness score is accumulated for each possible output throughout the duration of each generation, and then added together when the AI fitness function is called to form the base fitness level. The four output fitness levels are calculated as follows:

* Attack fitness
  + When the attack state is output by the network, attack fitness is incremented when the AI’s magic level is greater than zero and, if health is available, when its health level is above a certain level. Otherwise, attack fitness is decremented.
* Flee fitness
  + When the flee state is output by the network, flee fitness is incremented when the AI has a pursuer, its health is below a certain level and its pursuer is within attacking range. Otherwise, flee fitness is decremented
* Health fitness
  + When the find health state is output by the network, health fitness is incremented when the AIs health is below a certain level. Otherwise, health fitness is decremented
* Magic fitness
  + When the find magic state is output by the network, magic fitness is incremented when the AIs magic is below a certain level. Otherwise, magic fitness is decremented

Along with the base fitness level, fitness bonuses are also awarded for each enemy AI that the AI kills, when the AI follows through on a correct decision to collect health or magic and if all enemy AIs are killed.

**3.5. Stopping point**

In order to determine a stopping point for GA, the average fitness over each generation was recorded and observed. As can been seen in the results, the average fitness fluctuates heavily, while still showing an increasing trend in the fitness level. Refinement of the network and genetic variables did not show any noticeable stabilisation in the fluctuations. When comparing the average fitness per generation data from several runs, it can be observed that by 180 generations the trend in the average fitness level has either flattened out, or started to dip, making 180 generations a reasonable stopping point.

**3.6. Ease of Coding**

Implementation of the evolutionary neural network in code is relatively straightforward once an understanding of how the networks feedforward algorithm works is obtained, along with an understanding of the steps involved in the genetic algorithm. The code for the network itself makes up a small percentage of the overall code for the application. Integrating the network into the learning AI’s controller script was also easily achieved by passing the inputs into the feedforward method during the script’s update call and using the output to set the AI’s state.

**3.7. Computational Efficiency**

The computation performance of the network is dependent on the number of nodes that it contains. The performance can be described as where is the number of nodes and is the number as inputs per nodes. The feedforward algorithm performs floating point calculations, which modern process can handle very efficiently.

**3.8. Saving Sate**

In order to allow comparison of the learning AIs performance with the control AI, functionality has been added to save the state of the weight and bias values for the fittest member of the network after each iteration. This allows the saved weight and bias values to be loaded into each network instance that is populated at the start of the run, so every AI starts with the fittest configuration from the selected save file.

**3.9. Performance Optimisation**

It can be observed from the results below how adjusting the variables of the network impacted on its performance. As mentioned previously, the average fitness fluctuated heavily over the generations but a positive trend towards the solution could still be observed. Because of this, the average has been taken over every 10 generations and plotted in the chart in figure 3. The results in the table below report the parameters used for the neural network and genetic algorithm, along with average fitness over the last 10 generations for each run. From previous experiments, a population of 60 with 1 hidden yielded the best results will keeping the application performant. The focus for adjustment is on the remaining parameters.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Run** | **Population Size** | **Hidden Layers** | **Hidden Layer Nodes** | **Mutation Chance** | **Mutation Strength** | **Avg. fitness over last 10 generations** |
| 1 | 60 | 1 | 8 | 0.03 | 0.01 | 440.715 |
| 2 | 60 | 1 | 16 | 0.03 | 0.01 | 726.74668 |
| 3 | 60 | 1 | 8 | 0.01 | 0.01 | 741.22957 |
| 4 | 60 | 1 | 8 | 0.01 | 0.05 | 423.19667 |
| 5 | 60 | 1 | 8 | 0.005 | 0.005 | 704.06334 |

Figure 3: Average Fitness per 10 generations

**4. Results**

In general, training the AI using an evolutionary neural network yielded excellent results. After refinement of the network and genetic algorithm variables, along with the fitness function, the AI was able to make appropriate decisions based on its current circumstance. In order to assess how a trained learning AI performed against the non-learning AI, the learning AI was placed in a control mode. Metrics were recorded over 600 battles (10 iterations) for the control AI, as well learning AIs runs with a pre-saved configuration from each of the five run configurations in the performance optimisation table. Metrics recorded were win percentage, survival percentage and average survival time. Results are recorded in the table below

|  |  |  |  |
| --- | --- | --- | --- |
|  | **% Won** | **% Survived** | **Avg. Survival Time** |
| **Control AI** | 77.83 | 79.83 | 112.40 |
| **Run 1** | 68.83 | 66.50 | 105.17 |
| **Run 2** | 68.33 | 69.67 | 109.92 |
| **Run 3** | 72.17 | 73.17 | 109.50 |
| **Run 4** | 59.00 | 52.83 | 94.94 |
| **Run 5** | 70.50 | 68.17 | 108.64 |

From looking at the results it can be observed that the control AI outperformed all the learning AIs that had been pre-trained by the evolutionary algorithm. In the best performing run (Run 3), the margin of difference is relatively small, showing that well configured network can perform at a similar level to the state machine. Interestingly, the best performing training run (Run 3) as can be seen in the performance optimisation table, had the best win ratio, and the worst performing training run (Run 4) has the worst win ratio. This indicates that fitness function is indeed giving a reflection on the AIs ability to perform well in the given scenario. Overall, even though the evolutionary neural network was not able to better the state machine in these experiments, the results are still encouraging. With further refinement of the fitness function and optimisation of the network and GA hyper-parameters, it should be possible to gain further improvements. What can be seen is that it is indeed possible to achieve affective and intelligent AI decision making using an evolutionary neural network.

**5. Conclusions**

Creating believable, intelligent and interesting AI behaviour is challenging. Especially if the problem space within which the AI is to operate is complex. The prospect of unsupervised offline training of an AI is therefore appealing. Despite this, the use of machine learning within games has historically been a point of concern within the game AI community about whether they are the right fit for games (Rabin 2013). Despite the controversy, there is evidence here that evolutionary neural networks can be a useful tool for creating convincing AI behaviour. There are however some drawbacks worth noting. For this relatively simple example, it takes a considerable amount of time to train the networks in real time, even with the game speed increased 5x. It took around 2 hours to for the AIs to reach peak fitness. Considering all the runs required to find the optimum settings, it adds up to a lot of training time. Overall, the use of an evolutionary neural network for AI decision making has proved to be a useful technique, especially if a high level of control over the characters actions is not required. For more precise and predictable behaviours in the AIs decision making, other techniques such as a finite state machine may be more suitable.

**References**

1. Nielsen, M. (2015) *Neural Networks and Deep Learning*. Determination Press.
2. Millington, I. (2019) *AI for games.* Boca Raton: CRC Press.
3. Rabin, S. (ed.) (2013) *Game AI Pro*. Boca Raton: CRC Press.
4. Cho B.H., Park C.J. and Yang K.H. (2007) *Comparison of AI Techniques for Fighting Action Games - Genetic Algorithms/Neural Networks/Evolutionary Neural Networks*. In: Ma, L., Rauterberg, M. and Nakatsu, R. (eds) *Entertainment Computing – ICEC 2007. ICEC 2007. Lecture Notes in Computer Science, vol 4740*. Heidelberg: Springer. Available at <https://doi.org/10.1007/978-3-540-74873-1_8> (Accessed: 17 December 2020)
5. Wargaming Seattle (2010) *Supreme Commander 2* [Video game]. Square Enix.