# Investigating the Cooperative Behaviour and Performance of Multiple AI Agents Under Variable Conditions


Lewis Deakin-Davies - 1561926
Supervisor: Joshua Knowles/Hayo Thielecke

*BSc Mathematics and Computer Science*
School of Computer Science
University of Birmingham
March 2018

**Abstract**

Observing the nature in which multiple, independent AI agents interact and cooperate is an area with limited research. When explored, there isn't a focus on how differently the AI learn and perform when given alternative priorities and variables. Based on a stochastic Wolfpack hunting game, I have designed software that is capable of creating, training and simulating the performance of multiple, independent neural network based AI, with a focus on the ability to analyse how they learn when their characteristics are changed.

# Contents

# Chapter 1

# Introduction

The behaviour and effectiveness of independent AI agents when placed with the same task in an environment should differ when key parameters are changed. They should either defect, or perform in a cooperative way with varying degrees of success. The aim of the work described in this report is to provide a software tool with which people are able to observe this behaviour, and explore how impactful certain parameters such as extent of training and reward amounts can be.

Machine learning is a field of computer science that gives computer systems the ability to "learn" (i.e., progressively improve performance on a specific task) with data, without being explicitly programmed (Samuel 1959). Machine learning is a quickly growing field of computer science, the technology behind the various methods is improving, and the ability of these systems is being enhanced rapidly. Machine learning has been a popular area of interest as of late, breaching into mainstream media with the achievements of AI such as AlphaGo (Silver et al. 2017), which was recently able to consistently beat the highest ranked Go players alive.

Complex AI systems such as AlphaGo use what they describe as deep reinforcement learning methods to compute optimal solutions for their board game states. Their first method of deep reinforcement learning used neural networks to learn to play Go by studying millions of moves from games against humans. Slowly improving as it learned what could be deemed as 'good' and 'bad' moves. Evolutionary neural networks in conjunction with reinforcement learning are used to reduce the need for human interaction in

the learning process. Understanding the structure of neural networks and the way in which they function are crucial for understanding the details of this project.
Although there are many models for this neural network structure, they are most often regarding one AI vs either a human or another AI. There is a distinct lack in observations made with multi-agent cooperation models. This project aims to fill that void. The tool will be specifically designed to allow for easy exploration into how multiple AI will evolve and adapt their behaviours when tested under varying circumstances.

## 1.1 Report Structure

**Background Research:** Exploration of relevant studies on neural networks and AI cooperation.

**Analysis and Specification:** Analysis of both project and software aims, explanation of significant resource requirements.

**Design:** Insight into design decisions including software structure and UI.

**Implementation:** Explanation of implementation choices, analysis of methods and functions used throughout.

**Testing and Progression:** Detailing of what testing methods were used throughout implementation. Discussion of progression towards final product.

**Project Management:** Evaluation of the effectiveness of management methods used in the development of the project.

**Results:** Analysis of results produced by the software, evaluation into the degree in which software aims were met.

**Discussion:** Discussion into the degree in which project aims were met. Reflection on deficiencies of the project and improvements that could be made.

**Conclusion:** Brief conclusion on how the final project meets the original aims.

# Chapter 2

# Further Background Material

Before exploring the different methods of implementing the cooperative AI, an effective way of testing them was required. A way in which the performance of both of the AI could easily be recognised and evaluated, and which had aspects of both teamwork and rivalry. It needed to be simple enough to be implemented effectively and without obscuring the focus on the AI implementation itself.

## 2.1 Markov Decision Process

The Markov decision process models decision making in situations where outcomes are partly random and partly under the control of the decision maker (Bertsekas 2011). Montague (1999) shows that this process is useful for many areas including reinforcement learning, they also show that if the probabilities or rewards are unknown then the problem is one of reinforcement learning. Another important factor of Markov decision processes are that it has been proven that Q-Learning eventually finds an optimal policy (Melo 2001).

## 2.2 Stochastic Games

Stochastic games are played via a series of states. Each player will choose their move and then the game will move to the next state. At each state,

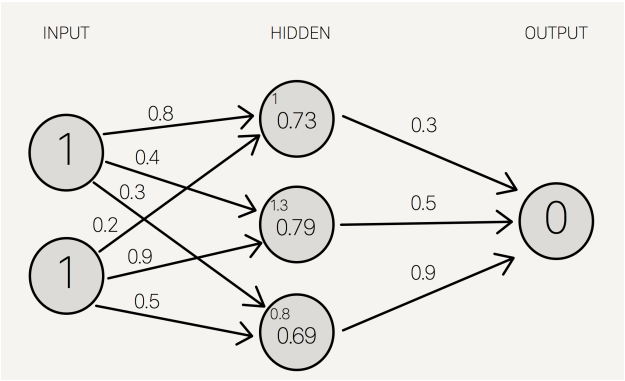

Figure 1: Neural Network (Steven Miller 2015).

a payoff is determined for each player. Stochastic games use a Markov decision process. Another important aspect of this type of game is that there can be an infinite number of states, having an open ended final score allows the learning process to continually improve without being limited by a top possible score.

Leibo et al. (2017) implements a social dilemma game that they have described as a 'wolfpack hunting game'. This game is a stochastic game that makes use of Markov decision processes and the AI will either cooperate or defect based on the reward payoff in different circumstances.

## 2.3 Neural Networks

As presented by van Gerven & Bohte (2017) artifical neural networks (ANNs) are an area of machine learning that loosely mimic the biological neural networks that constitute animal brains. ANNs are a collection of nodes (of which will be called neurons) and connections (weights). As detailed by Zhang (2000), a network will often be a multi layered perceptron. This will have three layers of neurons, input, hidden and output. This network structure will be referred to as a genome. Inputs are provided by the environment and are determined by the creator of the network beforehand. The input neurons are connected to each hidden neuron via weights, as shown in figure 1.

Neural networks have been used countless times in an attempt to learn to play games, the most commonly tested game being chess (Thrun 1995), (Fogel et al. 2004). Once neural networks became powerful enough to beat opponents in chess with ease, a new challenge was presented. Training a neural network to play Go. Silver et al. (2017) shows that although this was initially much more of a challenge than chess, it too has been cracked. This triumph in AI vs human has shown how powerful neural networks can be and how versatile they are when implemented correctly.

### 2.3.1 Genetic Algorithms

Genetic algorithms are a meta-heuristic inspired by the process of natural selection. Melanie (1998) shows that these are often used to evaluate solutions to high-quality solutions to optimisation and search problems, relying on functions such as mutation, crossover and selection. The representation of a genome is often designed as an array of bits (Whitley 1994). This structure allows for easier calculation and use of the crossover function. A genetic algorithm uses a set amount of genomes and will terminate when a fixed number of generations has been reached. Montana & Davis (1989) show the advantages that a genetic algorithm has over a network that uses backwards propagation when used in certain environments.

### 2.3.2 Sigmoid Activation Function

The activation function used within a neural network is important in defining the outputs. There are many different activation functions that can be used. Han & Moraga (1995) finds that the sigmoid function is commonly used and fairly flexible in conjunction with neural networks.

$$S(x) = \frac{e^x}{1 + e^x}$$

## 2.4 Google Deepmind

Leibo et al. (2017) shows the DeepMind implementation of this game has a matrix arena populated by three ANNs, two of which are depicted by red squares and the other a blue square. The two red squares are what are considered the wolves. They are tasked with catching the blue square, which

is the food. Each AI makes a move in each state to attempt to maximise its score. Score is given based on whether the food was caught, and if the other wolf was in the vicinity when captured.

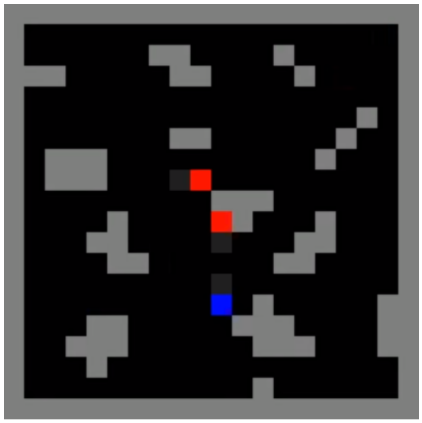

Figure 2: DeepMind Wolfpack Game.

# Chapter 3

# Analysis and Specification

Most important decisions are made before any implementation has been started. Understanding previous work, and background material is crucial in determining the specification and requirements of a system. Without prior knowledge, it is difficult to justify why development is conducted in a particular way. Using the research that was conducted, the functionality of the software was decided.

## 3.1 Project Aims

The research showed that there were many examples of neural networks being used to train AI, some of which used evolutionary algorithms. However, there was a lack of observation when it came to cooperation between multiple agents. Furthermore, consideration of how multiple AI agents would perform, given different combinations of variables and conditions was an area that had room for exploration.

The aim of this project is to develop a way in which observations can be made on how well multiple AI agents will perform when given these different combinations of variables and conditions. Using the software that has been created, it must be possible to effectively analyse the behaviour and performance of different genomes. The variables chosen must allow for a wide range of results that show how impactful they are. Network performance must be analysed when trained with different genome and generation counts. The specific behaviour of the AI must be observed and analysed when using the

software to train networks with different inputs and variables.

These are the aims that the project must achieve:

- Determine an effective implementation of a genetic algorithm based on background research.
- Effectively analyse the behaviour and performance of different genomes.
- Observe the impact of different combinations of variables and inputs.
- Observe the impact of training networks with different genome and generation counts.
- Analyse the behaviour of specific combinations of variables in regards to cooperation.

## 3.2 Software Aims

These project aims can be achieved by creating a piece of software that fills this gap of testable genetic algorithms. It needs to be easily accessed and informative.It needs to correctly demonstrate the nature of mutli-agent AI learning.

These are the aims that the software must achieve:

- Create a simple environment in which AI agents can be tested.
- Allow users to train networks using variables that they have chosen.
- Present results in an informative and analysable way.
- A save and load feature that allows users to view previous models must be implemented.
- A simple UI that clearly shows how to alter variables and conditions must be developed.

## 3.3 Resources

### 3.3.1 Programming Language

The software will be developed entirely in Java, this was chosen due to familiarity with the language. Eclipse will be used as the IDE.

### 3.3.2 Libraries

Java Swing will be used to implement the UI parts of the software. There will be no use of Neural Network libraries throughout the implementation of the network. This decision is explained further in the implementation section.

# Chapter 4

# Design

## 4.1 Network

ANNs are an effective choice regarding the method of learning that will be implemented in this software. Limiting the human input as much as possible is important to allow for as natural learning process as possible. Reinforcement learning is used in this implementation as it is an unsupervised learning paradigm and therefore doesn't require a data set. The agent learns entirely based on its own performance. The only guidance is the score that the ANN achieved after running a simulation. This score will be referred to as the fitness.

There are two main components of an evolutionary neural network, the evolutionary algorithm that progresses the networks, and the form in which the genomes and their weights are stored.

### 4.1.1 Genome

As shown by Kumar & Mishra (2013), the nature in which an ANN will calculate the values in the hidden and output nodes translates exactly into matrix multiplication. Storing the inputs, first set of weights, and second set of weights as matrices, it is very effective to perform the calculations on each stage and produce the desired output results.
The structure in which the genome will be constructed is in the format of two matrices of weights and two bias matrices. The bias weights are at a set value of 1. The bias weights are only added to the hidden layer and output

layer, these values are not impacted by the previous layer. The fitness of each genome will be stored within itself so that each fitness/genome can easily be checked and ordered when required. The way in which these genomes and weights are interpreted as matrices is shown:

$$\begin{bmatrix} in_1 & in_2 & in_3 \end{bmatrix} . \begin{bmatrix} wt_{11} & wt_{21} & wt_{31} \\ wt_{12} & wt_{22} & wt_{32} \\ wt_{13} & wt_{23} & wt_{33} \end{bmatrix} = \begin{bmatrix} hn_1 & hn_2 & hn_3 \end{bmatrix}$$

Based on the research conducted previously, the activation function that will be used alongside the weights will be the sigmoid function.

The process in which the outputs will be calculated will be as follows:

1. Matrix multiplication is applied to the inputs and the weights to produce the hidden layer values.
2. The bias values of 1 are multiplied by the weights and added to the hidden layer values.
3. The sigmoid activation function is applied the the hidden layer to normalise the values between $[0, 1]$.
4. Matrix multiplication is applied to the normalised values and the secondary weights to produce the output values.
5. The secondary biases are multiplied by the secondary weights and added to the output values.
6. The activation function is applied to the output values to normalise them.

### 4.1.2 Evolutionary Algorithm

Throughout the research, there was primarily a focus on neural networks with rigid topologies, because of this, it was decided that a set structure would be used, and that a set amount of neurons will be used in the hidden layer.
To begin, a certain number of randomly weighted genomes need to be created for each AI. The first collection of genomes make up the first generation.

1. $n$ amount of genomes are created for each AI.
2. Each AI is assigned one of its genomes and then simulated to completion.
3. After a simulation is completed, each genome has its fitness saved.
4. After all the genomes have been simulated, they are sorted by fitness.
5. The $n/2$ worst performing genomes are discarded.
6. Crossover and mutate functions are used on the remaining genomes to make up the missing half.
7. The new set of genomes are the next generation and steps 2-7 are repeated until the specified generation has been simulated.

The crossover and mutate functions are what allow networks to improve. They are both methods of tweaking the weights that connect the neurons.

### Crossover

After half of the genomes have been removed from the gene pool, they will be sent to the crossover function where they will be used to produce offspring. The crossover function will take two random parents from the existing gene pool and randomly distribute their weights to a third genome which will then be added to the next generation.

### Mutate

During the crossover function, a mutate function will be applied to the offspring. The mutate function will take any weight chosen for the offspring and possibly alter it. Curran & O'Riordan (2003) shows that an appropriate rate in which mutation should occur can be found with their model at around 0.02. This will be used as guidance when finding a mutation rate that works for the model that is being implemented.

## 4.2 Simulation

The design of the simulation environment that will be used in this task follows closely to the one used in the DeepMind project seen in figure 2. The two AI wolves will be red and the food will be blue. The environment that will be used in this task is similar but without the additional obstacles.
The game operates by first placing the two wolves and one food randomly within the arena. This is the starting state. Both the wolves choose where they want to move and then move on to the next grid spot at the same time. This is the second state. Immediately following the wolves' step, the food moves in its desired direction based on the location of the wolves. This process is repeated continuously until one of two states occur:

1. A wolf catches the food
2. Both wolves die

In the first case, once the wolf has stepped onto the tile that contains the food, the network that is controlling that AI is awarded with a user specified amount of points. Depending on variables set at the start, more points are awarded to the two wolves based on the current position. After points have been added, both the wolves and the food are moved to a random location in the arena once again, and the game continues as before.

A wolf will 'die' if it either attempts to step out of the arena, or if it has made a total of 1000 steps. It was decided that the wolves were to die if they attempted to step out of the arena so that there will be an indicator, in the earlier stages of development, that they are actually learning to some extent. If after excessive learning, they are still perishing quickly without learning to avoid walls, it can reasonably be assumed that the network or game has not been implemented correctly. Having a high limit to the number of steps is important once the network has been implemented to a testable degree. This will stop the wolves from being able to get stuck in an infinite loop of pacing back and forth. The limit will is high enough that if a network makes it to 1000 steps, it is reasonable to assume that it has learned to survive effectively.

When a wolf dies, that particular AI stops functioning and cannot gain any more points, if the other wolf is still alive, their network keeps functioning and the wolf keeps moving until it also dies. Once both wolves have died, the

fitness of each wolf is saved to their current genome. The next two genomes are selected for the corresponding wolves and the locations of the food and the wolves are randomised.

The general premise of the DeepMind version of this game is appropriate for this task, the matrix layout is effective and simple for determining the location of the AI, food, and walls. However, the way in which their AI was implemented is slightly too complex for this model. They used three ANNs to control both wolves and the food. In this model, it has been decided that only focusing on the cooperation of the wolf AI is needed. The food doesn't need to be a learned network to complete this task. Instead, a hard programmed system will be designed which will effectively avoid the predators and can only be caught via cooperation by the wolves. It needs to be implemented this way as if it is attainable by only one wolf, it would be unreasonable to assume that variables are changing the amount of teamwork.

The inputs of their version occur in the form of a matrix of coordinates obtained from the 'front' of the wolf. Complex inputs such as these are not required for this model. The software will be designed to function with more basic inputs. To see how effective they could be with more restricted information, they will only be given values for their immediate surroundings and a general direction to which the food is. In addition to this, using a matrix of coordinates in front of the wolf as inputs would limit some of the design choices that are paramount to the final project. Variables such as distance to other wolf or angle to other wolf wouldn't be relevant inputs as they would be able to see where each other were in a very explicit way. These additional variables were an interesting consideration as they can be used to observe different behaviours of the wolves that aren't so obvious. In theory, one wolf knowing how far away the other wolf was would help it in making a decision on whether to capture the food with or without the other wolf being close.

## 4.3 Displaying results

Providing a user friendly and effective way of observing how different genomes performed is a key part of the task. A way in which genomes can be saved and later viewed is crucial for research purposes. Two ways have been ex-

plored in which the performance of the genome can be recorded. Genome weights themselves could be saved straight into text files as they are simply lists of doubles. These could then be read to recreate the genome for testing. Another method is simply to record the coordinates of both agents and food in each state of the simulation. This method follows a similar principle that the move set could then be read and displayed at a later point.

Recording the genome itself would allow for multiple simulations with differing results as the position of the food and wolves would be different when testing. However, this would result in the simulation not being consistent. If observed that the wolves performed well with a particular set of random locations, it wouldn't be guaranteed that they would display the exact same behaviour when tested again with new randomised locations.

Relying on just having the exact moves and coordinates recorded allows for exact examples to be displayed. You would be able to view the simulations multiple times to see the exact nature in which the wolves performed. This is the better option for completing this task as it will be easier for users to identify and analyse a certain genome. It doesn't allow you to observe how the same genome will perform when given slightly different environments, however, multiple genomes can be trained under the same variables to achieve the same outcome.

When displaying the performance of the AI it needs to be clear what is happening at each state. There need to be clear indicators of when a wolf has died, and when a wolf has successfully caught the food. When a wolf has run out of steps or wandered out of bounds, it will turn black. When a wolf captures the food it will turn yellow.

# Chapter 5

# Implementation

Before beginning the implementation, libraries were that could aid the process were explored. Libraries would save time and require less intense research into the specifics of neural networks. Libraries such as Neuroph (Severac 2018) and DeepLearning4J (Skymind 2018) were looked into for use. Neuroph seemed appropriate as it had ways to quickly and easily implement the networks and train them without difficulty. This wasn't exactly useful for the task at hand as a more integrated connection between the game and the network was more appropriate. Also, it was against the original aim to create training data for the network. Their studio software proved useful in presenting the network architecture in a way that made it easier to implement. DeepLearning4J followed a similar design and also wasn't deemed appropriate. After ample research and testing, the decision to implement the neural network from scratch was made. In the process of understanding how machine learning can be used, and how to design a system that applies it, developing it without libraries would be best suited to implement it in a way that is specific to the aim. Wanting to explore the intricacies of neural networks also played a hand in this decision.

## 5.1 Neural Network

### 5.1.1 Genome

A Genome class was created to more easily manage the values in each neural network. The genomes need to be readily computable and stored in a format

that allows the neural network to process them effectively.

The weights in an ANN are normalised real numbers often in a range $[-n, n]$. For my simple ANN a range of $[-1, 1]$ is effective. The genome needs to store 2 matrices of these weights. A way in which matrices can be implemented in Java is to store these doubles as a 2D array. 2D arrays can be interpreted as matrices. Helper functions were written in a Matrix class that allowed for easy addition and multiplication. The first matrix is the weights between the inputs and the hidden layer, the second is between the hidden layer and the outputs. The size of each matrix is derived from the number of neurons on either side of the weights. When a genome is created, the arrays are initialised and filled with randomised doubles within the specified range. The two bias arrays are also initialised and each weight is set to 1. A Genome is stored in the type of a List size 4 of 2D arrays so that each set of weights can easily be processed by the neural network.

### 5.1.2 Network

The ANN was implemented in a general way. The network was designed to be constructed in a manner that allowed any game or system to interact with it seamlessly. The topology is determined by the game. The methods and functions it uses are transferable to any environment. A predetermined number of input, hidden, and output nodes are initialised when the network is created.

After the number of neurons and genomes has been determined, and the network is created, the genomes are initialised. Each genome is created with their random weights, and added to a list of genomes specific to that neural network. Once the list has been filled with the desired number of genomes, the game will then begin to simulate states. It takes the first genome from the two lists (one for each AI) and iterates between them calculating each wolves next move.

Pseudocode 1: calculating outputs.

```
public double[] getOutputs(double[] inputs, int currentGenome) {
   ...
   hiddens = Matrix.multiplyMatrix(inputHiddenWeights, inputs);
```

```
hiddens = Matrix.addMatrix(hiddens, bias);
hiddens = Matrix.sigMatrix(hiddens);

outputs = Matrix.multiplyMatrix(hiddenOutputWeights, hiddens);
outputs = Matrix.addMatrix(outputs, bias);
outputs = Matrix.sigMatrix(outputs);

double[] output = (Matrix.toArray(outputs));
return output;
```

The next move is calculated by calling a function in the network called *getOutputs*, it takes the array of inputs that have been determined by the game and the number assigned to the current genome. The weights and biases of the genome are extracted and assigned to 2D arrays using the assigned number and list of genomes. The weights are described by their variable name in Pseudocode 1. The values that are assigned to the 2D array named *hiddens* are the outcome when matrix multiplication is used on the weights and input values. The biases are added by multiplying each weight by the bias value and adding it to each of the values in *hiddens*. These values then need to be normalised using the sigmoid function. $sigMatrix$ takes each value in the 2D array and applies the sigmoid function to it to ensure that it is in the range $[0, 1]$.

The same process is repeated to provide the *output* array. Matrix multiplication is used again on the weights and the normalised values in the hidden array, the biases are added, and then the sigmoid function is once again applied to each value. The matrix of output values are then converted into a regular array and returned.

Once a simulation has arrived at the completed state where both wolves have died, the fitnesses of the two genomes need to be saved and a check on whether or not the genomes should be evolved is performed.

Pseudocode 2: Saving fitness and beginning evolution.

```
public void newGenome(double fitness, int currentGenome) {
      Genome curGenome = curGenomes.get(currentGenome);
      curGenome.setFitness(fitness);
      ...
```

```
    if (currentGenome == maxNumberOfGenomes) {
       if (noProgress) {
          // Create new set of genomes
       } else {
            Collections.sort(curGenomes,
               Collections.reverseOrder());
            curGenomes = evolve(curGenomes);
       }
    }
  }
```

This check is performed by the *newGenome* function. It is called when a simulation has finished. The function is called on each genome that was used in the simulation. Firstly, it takes the fitness that was achieved by the AI and saves it to its genome. It is then checked whether or not the genome that was just tested was the final genome in that generation. If it is found that every genome has been simulated, another check is made to ensure that some progress was made. The boolean *noProgress* is set to true as soon as one genome achieves a score over a certain threshold. This threshold was more important when the AI wasn't implemented to be as effective as they are in the final model. A generation will almost always have at least one genome that scores a relevant amount of points. If it is found that there has been no progress, a new set of genomes are made and the process is restarted. When a generation has been completed, and progress has been made, the corresponding list of genomes is then ordered processed by the evolve function.

The evolve function is what causes the genomes to improve. It implements the crossover and mutate functions. It is paramount to the concept of this task and must be implemented carefully to ensure progress is made.

Pseudocode 3: Creating the new generation.

```
public List<Genome> evolve(List<Genome> currentGeneration) {
       List<Genome> newGeneration = new ArrayList<Genome>();
      for (int i = 0; i < maxNumberOfGenomes / 2; i++) {
        newGeneration.add(currentGeneration.get(i));

        int parent1 = randomBetween(0,maxNumberOfGenomes/2)
        int parent2 = randomBetween(0,maxNumberOfGenomes/2)
```

```
        newGeneration.add(crossover(currentGeneration.get(parent1),
            currentGeneration.get(parent2)));
      }
}
```

The evolve function firstly sorts the list of genomes in descending order and takes the best performing half of the current generation and saves them to a new list of genomes. Next, two random integers are used to choose two genomes. These are sent to the crossover function as the two parents that will be used for the creation of a new genome.

Pseudocode 4: Performing the crossover.

```
public Genome crossover(Genome a, Genome b) {
Genome c = new Genome(inputN, hiddenN, outputN);
for (int i = 0; i < hiddenN; i++) {
        for (int j = 0; j < inputN; j++) {
           randNum = randDouble(0, 1);
           if (randNum > 0.5) {
              (c.get(0))[i][j] = mutate((a.get(0))[i][j]);
           } else {
              (c.get(0))[i][j] = mutate((b.get(0))[i][j]);
           }
        }
      }
       // Repeated process for the weights between the hidden and
          output layer.
       return c;
}
```

This *crossover* function extracts weights from each of the two parents and assigns them to a new genome to be added to the next generation. Initially, a new genome is created with the predetermined structure, this genome is then assigned either a weight from the first or second parent based on a random double between $[0, 1]$. Firstly, the weights between the input and hidden layer are assigned, this is shown by $c.get(0)$ being used. The first array in a genome is the first set of weights. The process is then repeated using $c.get(1)$ to determine the weights between the hidden and output layer.

Before the weights are assigned to the new genome, a function called *mutate* is called on them, this mutate function is crucial in ensuring that progress can be made over several generations, it possibly tweaks the values slightly to ensure that the entire genome pool is not homogeneous. If the gene pool becomes too homogeneous it can become very difficult for the genomes to break out of a bad cycle of evolution.

Pseudocode 5: Mutating the weights.

```
public double mutate(double weight) {
      double rand = randDouble(-1, 1);
      double r = Math.pow(rand, mutatePower);
      double newWeight = weight + r;
        randNum = randDouble(0, 1);
        if (randNum < 0.2){
         return newWeight;
            } else {
            return weight;
            }
}
```

The mutate function takes the weight that is to be assigned to the new genome. A random double is generated and then it is put to power of the a variable called *mutatePower*, this variable determines how drastically a weight can be altered each generation. In this implementation it is set to 9 so that if the random double takes a small value, not much is changed, however, if a value close to the bounds is taken then there will be significant alteration. The mutated value is only taken 20% of the time, not every weight is mutated. This mutate value was determined by many tests of how consistently the genomes would perform with different percentages and also the guidance found in Curran & O'Riordan (2003)

Once the new generation has been fully populated by the previous best performing genomes and the mutated child genomes, it is ready to be tested and simulated again. This process is repeated continuously until the specified number of generations has been reached.

## 5.2 Simulation Environment

The aim of the trainer class is to be able to simulate every state of the game very rapidly and effectively, to quickly test each genome.
Before implementing anything to do with the game itself the logistics needed to be determined. This involved choosing what exactly each input and output should be, and in conjuction, determining the number of neurons in the hidden layer. Previously having decided not to follow the same model that DeepMind did, the way in which the wolves operated was evaluated. As stated earlier, the decision that they would only be able to see their immediate surroundings was made. This was the basic premise originally. Once I had made sure that I had a network that was functioning at least to a basic level I evaluated how I would extend it towards the wolfpack game.

The implementation of the wolfpack game took many forms and went through many stages. It was started by creating a basic coordinate based environment with just one moveable AI and a stationary food. This was to ensure that my neural network was working in some capacity. It was important to have some model in which the network could be tested against to further improve it. In this early stage, the aim was to have the AI learn to avoid perishing by avoiding the edges of the arena. Inspiration in what inputs should be used was taken from the implementation of snake games (Korolev 2017) and (Binggeser 2017). These snake games were similar in the fact that they had to avoid the edges but the snake took forms that wouldn't allow it to turn back on itself. This led to the initial decision of using 4 inputs to determine what element was on each adjacent side of the AI. This input took the form of an integer:

- -1 represents a wall
- 0 represents an empty space
- 1 represents a food object

In addition to these four variables, two more were implemented. The idea that the wolves would have to lock down the food, effectively trapping it between the two wolves or into a corner, allowed for more inputs to be considered. Knowing the direction in which the other wolf was would be helpful in ensuring that they were both heading towards opposite sides of the food.

Knowing the absolute distance between the two wolves was also useful in ensuring that the other wolf was or wasn't in the range to receive the proximity bonus. This resulted in a final six inputs:

1. Adjacent above
2. Adjacent below
3. Adjacent left
4. Adjacent right
5. Distance to other AI
6. Angle to other AI

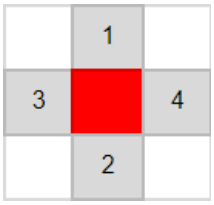


Figure 5.1: Adjacent inputs.

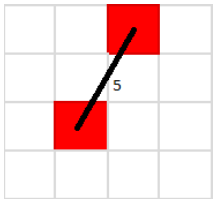


Figure 5.2: Distance between AI.

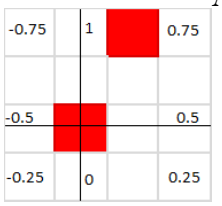


Figure 5.3: Section based angle.

The angle between the AI is determined by which section the other AI falls into relative to the current AI. If the other AI is directly above, it takes a value of 1, if it is directly to the right, a value of 0.5. Between these two directions is the section which gives a value of 0.75 as shown in the diagram. It was implemented this way to ensure that the input value was normalised in the range $[-1, 1]$.

Inputs 5 and 6 were implemented in a way that allows them to be toggled on or off before training a network. The network will see which inputs it needs to consider and construct the network with the correct number of neurons in each layer. Giving the user the ability to decide whether or not they wanted the distance and angle to be inputs that were considered allowed them greater freedom over the results they wanted to acquire.

At each state, the inputs are calculated by various methods within the AI class. The inputs are sent to the ANN and outputs are received. The outputs are in the form of a double array size 4. Initially one output was interpreted, with thresholds determining the correct move to make. However, as the sigmoid function is not linear, it resulted in certain thresholds closer to the middle to be more common.
Each of the 4 outputs is a double in the range $[0, 1]$. Each output is interpreted as being representative of one of the directions:

1. Step upwards
2. Step downwards
3. Step Left
4. Step right

Each of the output values are compared and the output with the highest value is chosen and the corresponding move is performed. It is conducted in this way because there will be multiple viable moves at each state that can return the same value. Using four outputs means that the weighting will allow for the most desired move to be more easily identified and performed.

With the number of inputs and outputs decided, the number of neurons in the hidden layer must be determined. There has been a lot of research into what the best number of hidden layers there should be, and also how many neurons it should have, one case being Stathakis (2009). This expresses that there isn't a set correct number for layers and neurons but a 'rule of thumb' that can be used as a starting point. Using this paper and given that there are 6 input values and 4 output values in this implementation, it was decided that using only one hidden layer containing 5 neurons was appropriate.

The process that iterates through each of the game objects and updates their positions is in the form of a while loop. This run function makes the necessary checks that determine what function should be called at each of the various game states. It is essentially the engine.

Pseudocode 6: Simulation engine.

```
while (running) {
        for (int i = 0; i < aiCount; i++) {
           AI curAI = aiList.get(i);
           NeuralNetwork curNN = nnList.get(i);
           if (CurAI intersects the target food) {
              refresh(curAI);
              steps--;
           }
           ...
           learn(curAI, curNN, i);
           ...
           steps--;
           //Calculations are made that determine whether the wolf
               is closer or farther away from the food after the
               step. If they are farther, they lose 2 points, if
               they are closer they gain 1.
        }
        // Check if CurAI intersects the target food again

           if (steps <= 0 || oob(curAI)) {
              gameOver(curAI, curNN);
           }
        }
        //Calculate which AI is the closest to the food and send
            that AI to the proximity calculation.
        proximity(aiList.get(closestAI));
   }
}
```

The first AI is chosen and a check is made to see whether it is currently on top of the food target. If it has captured the food, the *refresh* function is called. The AI and current neural network are sent to the learn function. This function extracts the input values from the AI and sends them

to the *getOutputs* function in the Neural Network code. As explained in Pseudocode 6, if the wolf has moved away from the target it is penalised, a move towards the target is rewarded. This was implemented like this to avoid a wolf having its fitness inflated by pacing back and forth. Reducing the wolves fitness when they moved away pushed them toward trying to capture it. Each time a move is performed, the number of steps that the wolf has is reduced by one. If the number of steps left reaches 0 the wolf is considered dead. If the wolf attempts to step outside of the arena it is also dead. This is calculated by the *oob* function. If either of these are true, the AI is sent to the *gameOver* function. Once these checks have been made on both the wolves, the moves have been performed and the positions updated, the food target takes the coordinates of the closest wolf and uses them to make a decision on where its next move should be. This decision is made by a series of case by case calculations observing the optimal solution in each state.

With the overall structure and design of the simulation decided, the different states needed to be coded. If a wolf were to capture the food, the score needs to be updated, proximity calculations need to be made, and randomisation of positions needs to be performed.

Pseudocode 7: Adding points and refreshing gamestate.

```
public void refresh(AI ai) {
...
      if(distanceBetweenAI <= range){
         //Other AI gains the proxmity points.
         //Scoring AI gains half the proximity points.
         }
      ai.score = ai.score + foodPoints
      for (int i = 0; i < aiCount; i++) {
         AI curAI = aiList.get(i);
         int aiCoordX = randomCoordinate();
         int aiCoordY = randomCoordinate();
         curAI.x = aiCoordx;
         curAI.y = aiCoordy;
         }
            ...
      int tarCoordX = randomCoordinate();
      int tarCoordY = randomCoordinate();
```

```
        tar.x = tarCoordX;
        tar.y = tarCoordy;
}
```

The $refresh$ function is called when wolf catches the food target. The proximity points that were determined when the neural network was first set up are distributed to the wolves if they fall within a certain range of each other. The range that is chosen for this implementation is 3 steps. Both wolves are awarded points if they are close, the wolf that captured the food is awarded the additional $foodPoints$ that were chosen before training. The coordinates of all the objects in the arena are randomised and updated.

When a wolf dies, the *gameOver* function is called, this first checks to see which wolf has died and updates its status. If the gameOver function is called and it results in both the wolves being dead, the process to save the fitnesses and choose the next genomes is started.

Pseudocode 8: Terminating the simulation and updating genomes.

```
public void gameOver(AI ai, NeuralNetwork nn) {
        ai.running = false;
        ai.dead = true;
        fitness = ai.score;
        for (int i = 0; i < aiList.size(); i++) {
           if ((aiList.get(i)).running == true) {
              aiRun = true;
              break;
           } else {
              aiRun = false;
           }
        }
        ...
        if(!aiRun){
           if(finalGeneration){
              ...
              SaveLoad.write(aiMoves, tarMoveSet, aiMoveCount,
                  tarMoveCount, arenaSize, saveName);
              SaveLoad.writeGen(saveName);
              (new Thread(new Displayer(aiMoves, bestTarMoveSet,
```

```
                aiMoveCount, tarMoveCount, arenaSize,))).start();
            } else {
            ...
            //Iterate through the both networks and genomes and
                update their fitness.
            //Reset all of the variables for the AI and randomise
                their locations.
            }
        }
}
```

The *gameOver* function performs numerous checks and controls when the movesets are recorded for each genome. Firstly, the wolf that activated the function is changed to the dead state and ceases to run. A check is performed to see whether any wolves are still running. If there is still a wolf that hasn't died, the simulation continues with that wolf until completion. If both wolves have now died, a check is performed to see whether the genome that was just simulated was the last genome of the last generation. If it isn't, the algorithm continues to function as before, updating the fitnesses and resetting the game state. If it is found that the algorithm has completed all the training that was set, all of the recorded move sets are saved to a text file. These movesets are obtained by performing the final evolution of the genome and simulating the first 5 for each AI. In each of these simulations the move set is recorded. The average fitness of the two AI in each simulation is used to decide which moveset should be saved. These move sets are sent to the Displayer class where the simulation is played out in a way that is easily observable.

## 5.3 User Interface

The simulations are not displayed whilst performing the learning part of the process. Running thousands of simulations a second would be inefficient. Therefore, only displaying the most effective set of genomes after all the generations had learned was appropriate.
The final simulations move coordinates need to be saved. When the final generation has completed its last few simulations the lists of genomes are sorted one final time and the top 5 genomes in each list are simulated. Each time a genome is simulated, the coordinates are saved in 2D arrays along with the fitness of each genome. After the 5 have been simulated, the best

genome from each list has its move set saved as a text file and is displayed. Having the top 5 recorded and compared worked more effectively than just displaying the best genome. It allowed for there to be more variance and a higher chance of an effective genome. Having a genome perform badly isn't very likely when thousands of generations with huge gene pools are tested, but, when testing smaller sizes there can be issues with performance.

### 5.3.1 Variable selection panel

The user interface that needed to be produced for this task isn't very complex. A simple and obvious way to see which variables can be changed and a way to alter them was required. Input boxes and check boxes are used to collect the information that is sent to the neural network so that the correct simulations can be performed.

There are two parts to the UI that allow you to see the neural network functioning. The first is to input the variables you wish to observe and then have the neural network completely train a new genome and then display it. The performance of these genomes is saved as a text file.

Secondly, you are able to select an already processed neural network for displaying. It will read the text file that has been created and display the moves that that genome took in its final generation of simulation. Saving and displaying the genomes in this manner allows the user to easily and conveniently observe how different genomes have performed.

This is how the variables are interpreted:

**Size of grid x by x:** Input one integer here to determine the size of the arena. The arena will be made into an x by x square with x amount of steps in the horizontal and vertical direction.
**Number of Generations:** Input an integer here to determine how many generations will be ran before the genomes are displayed.
**Number of Genomes:** Input an integer here to determine how many genomes will be initialised and used in the training.
**Points for Proximity:** Input an integer here to determine how many points the AI will receive if it close to another AI when the food is captured.

**Points for Eating:** Input an integer here to determine how many points the AI will receive if it captures the food.
**Choose a save name:** Write a name for the genome performance to be saved as.
**Check AI Angle:** Check this box if you want the angle between the AI to be taken as an input.
**Check AI distance:** Check this box if you want the distance between the AI to be taken as an input.

The implementation of the UI is a done using a JFrame, each of the numerical inputs are JTextFields and the two additional inputs are JCheckBox. See figure 5.4.

### 5.3.2 Displaying results

The simulations were performed using a coordinate grid system, this worked well because it allowed the move set to easily be displayed using a JFrame. The coordinates for each object at each game state are sent to the displayer. Every 300ms the coordinates are updated with the next ones in each array. This allows ample time for the user to follow what is going on throughout the simulation.
The two AI are coloured red and brown to differentiate them. The food is coloured blue. The edge of the arena is coloured black and there are gray grid lines to display each position a wolf can step into. See figure 5.5.

If it is found that the AI coordinates result in a wolf heading out of bounds, the wolf is turned black and the coordinates are no longer read. If the wolf comes into contact with the food, the wolf is turned yellow and there is a pause before refreshing the locations of each of the objects. See figured 5.6 and 5.7.

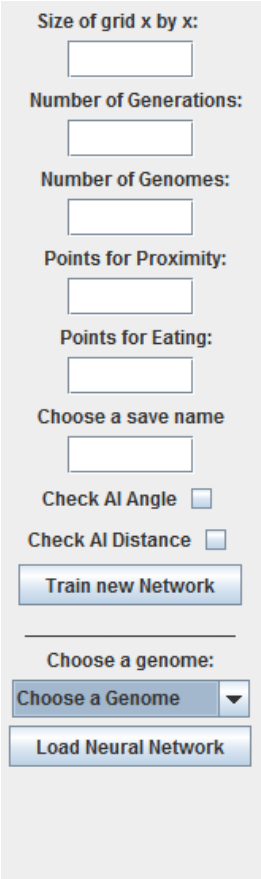


Figure 5.4: User Interface.

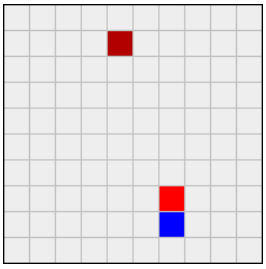

Figure 5.5: Performance Display.

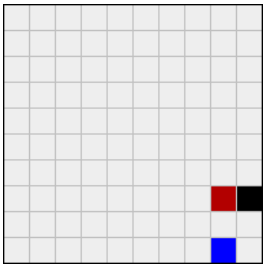

Figure 5.6: Dead Wolf.

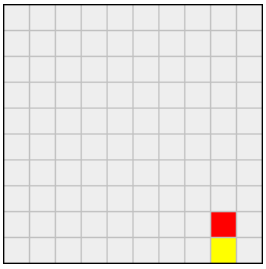

Figure 5.7: Successful Wolf.

# Chapter 6

# Testing and Process

Throughout development of this software there was an immense amount of testing. Both white box and black box testing were used, however, white box testing was far more significant throughout the project as the user requirements to operate the software were simple, and the implementation of the UI was not complex.

## 6.1 White Box Testing

Printing lines at various parts of different functions provides a lot of information on how the method is operating. Each major aspect of the implementation of the network had to be thoroughly tested. The network and genome structures contain complicated features that need to be implemented perfectly to ensure that there isn't a slight deviation from what is expected.

### 6.1.1 Genome

After deciding on the 2D array structure for this class, ensuring that the arrays were being translated into matrices properly, and that the multiplication was correct was paramount. Many functions were made in a helper class called Matrix that allowed for lots of testing to occur. Methods were implemented that allowed the 2D arrays to be printed in the matrix form. Having the matrices be printed whenever a genome was altered allowed for checking as to whether or not the genome had been altered in the expected

way. Being able to view the arrays was incredibly helpful in debugging almost every other aspect of the network code, as the network's main aim is to modify the genome weights.

#### JUnit Tests

Junit tests were carried out on the matrix functions. It was very important that these matrix functions operated correctly. The entire foundation of the network is based on these matrix additions and multiplications being accurate. If the values that the functions were returning were not correct, then the entire basis of genome evolution wouldn't have worked. All of the JUnit tests for the matrix functions came through successful. The JUnit tests can be found on GitHub.

### 6.1.2 Network and Environment

Due to the fact that after conducting lots of research into neural networks, they were being understood more thoroughly throughout implementation, there were many iterations that worked as stepping stones towards the final version. Each of these stepping stones served as tester functions that ensured each part of the network was functioning as it became more complex.

The first iteration of the neural network was to attempt to construct a simple backwards propagated network that would be able to solve the XOR problem. This was used to ensure that the weights of the genomes were being manipulated in a way that would progress towards a solved model.

Following a greater understanding of a network in practice, the design of the final iteration was started. Taking inspiration from Fabiorino (2017), a basic model in which a neural network is used to learn to play pong, a simple evolutionary network was implemented, taking two inputs; the Y coordinate of the paddle, and the Y coordinate of the ball. Using only one output to determine whether the paddle should go up or down, this model allowed for further testing of the capabilities of the network.

Printing the outputs that the network generated and observing whether or not the model would follow these correctly was a key part of ensuring the network was functioning. There were problems regarding the interpretation

of outputs in choosing which move to take, these were solved by constantly printing the output and then which move it chose to take. This allowed for the area in which the problem was arising to be identified.

Throughout implementation, although incredibly inefficient for the network, it was incredibly helpful to implement a basic display system for the actions that were being simulated. The performances were displayed as they were being simulated. This made it possible to see exactly how the networks were performing and what decision/traps they were getting stuck with. This wasn't a problem for the processing time at the start as the network was basic and didn't require much learning initially.

As the network and environment became more complex, it wasn't always often clear what was stopping the AI from learning effectively. After lots of investigation into various parts that could go wrong, it was discovered that one of the reasons that they weren't learning effectively was because they were not trained for enough generations with a large enough genome count. After allowing a larger amount of learning, the AI displayed behaviour that was expected. This became a problem as the more complex the network and environment got, the longer the training needed to be to analyse if it was working. This is where displaying the simulations as they were processed was stopped.

## 6.2 Black Box Testing

Input boxes, check boxes and buttons were used in the implementation of the UI. These all needed to function correctly in training the AI. The input values could be checked by test running the software and observing whether the correct variables were chosen. Text files being saved correctly in a way which allowed them to be replayed was also tested.

# Chapter 7

# Project management

Undertaking a task that was heavily related to AI was a challenge having no prior taught knowledge of AI systems. Because of the vastness of machine learning and the scope of the task, a great deal of time was spent researching.

The first month of undertaking the project was primarily focused on doing this research, exploring the many facets of AI and machine learning to find what best suited the task at hand. Creating simple applications of different types of AI helped throughout this process to more effectively narrow down appropriate area. Once neural networks had been settled on, sufficient learning about them was required for one to be implemented. Roughly six hours, five days a week how much time was taken to research and work to complete the software to a level that appropriately satisfied the task at hand. The project was thoroughly enjoyable for the most part, this was a great help in ensuring that motivation was at a level that would guarantee the project to be completed.

# Chapter 8

# Results

The software can now be used to observe the behaviour between mutiple AI agents. The different variables that can be changed before training the networks results in a large amount of data. There are different aspects of the network that can be changed that will result in different observations.

## 8.1 Changing variables

### 8.1.1 Number of Genomes/Generations

Using the software, it is possible to observe how much of an impact is made when changing the number of genomes/generations when training the AI. Here are the results of changing the number of genomes when used with the following standard variables:

- Gridsize: 10
- Number of Generations: 200
- Points for proximity: 150
- Points for eating: 100
- Check AI angle: No
- Check AI distance: No

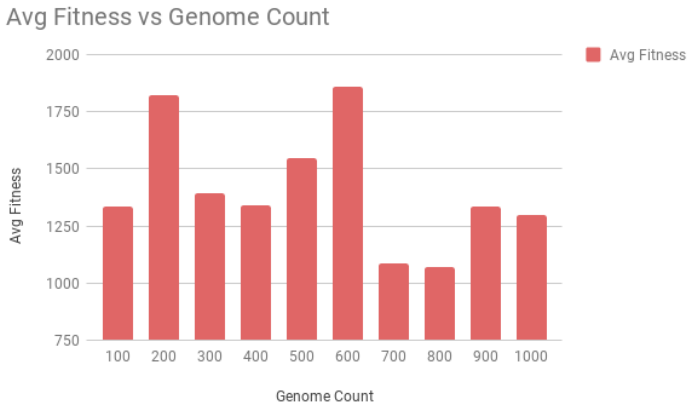


Figure 8.1: 10 results were averaged for each amount of genomes.

Using the software, it is shown that there isn't a clear cut trend in performance. What is expected is that the fitness increases as the number of genomes is increased as evolutionary genomes rely on their initial diversity to allow for a vast gene pool to build from. However, in these results, there was a large variance in the performance in each model. Sometimes the AI would perform very poorly even when tested with many genomes. The genomes that performed poorly were often when one of the wolves perishes early on. This completely limits the amount of points that the solo wolf can gain as the target food is uncatchable without the cooperation of two wolves.

The number of generations plays a huge role in determining how thorough the AI will learn. The main impact of using a large gene pool was that the variance was reduced. When using a smaller gene pool, it was found that sometimes very effective genomes were developed but for the most part, they were weak. This is because this algorithm relies heavily on chance, just like evolution in the non virtual world. It would have been better to perform the same tests using 500 generations to try and reduce the chance aspect as much as possible. This wasn't very practical however because running hundreds of these tests takes an incredible amount of time.

When testing the impact increasing the number of generation has on the effectiveness of the AI, a set amount of 500 genomes was used. There are only 3 tests performed per increment of generations due to the amount of

time it takes to perform simulations of this magnitude.

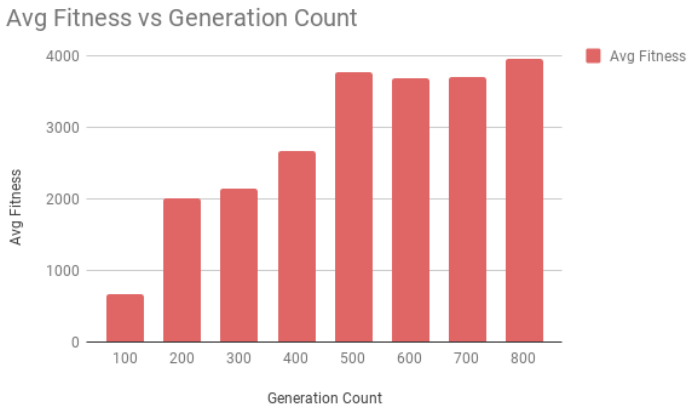


Figure 8.2: 3 results averaged.

In this model there is a clear upwards trend in the effectiveness of the AI. The impact in which the number of generations makes is much more clear cut than the number of genomes. The number of generations plays a large role as it allows for even a smaller gene pool to go through enough crossovers and mutations to develop an effective genome. The fitness tends to plateau as the generation amount increases above 500 as all the genomes are reaching close to their maximum potential.

These results show that the software can be used to observe the performance of the network when the number of genomes and generations is changed.

### 8.1.2 Additional Inputs

Two additional inputs can be used when training the wolves, the direction toward the other wolf and the distance between the two wolves. Using the following settings, it was tested whether or not these play a large role in the performance of the AI when compared to the performance without the additional inputs:

- Gridsize: 10
- Number of Generations: 1000

- Number of Genomes: 1000
- Points for Proximity: 150
- Points for Eating: 100

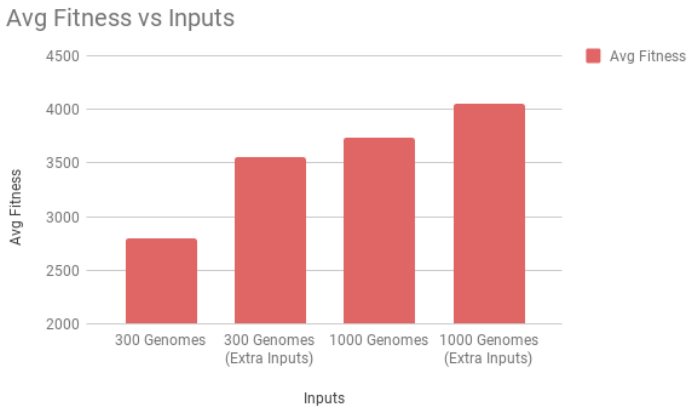


Figure 8.3: Performance of additional inputs.

Due to the fact that the wolf is rewarded with a portion of the proximity points when another wolf is nearby, the average score is found to be higher when they have a better sense of where each other are. This difference is not greatly pronounced in an arena of this size. If a larger arena were to be used, the extent of how far the teamwork aspect goes may be observed.

### 8.1.3 Observing behaviour

To observe how the AI learn different behaviour, the variables for the amount of points received when catching the food, and the amount of points awarded when being close when the food was captured, can be changed. The software allows these changes to be made easily, and after training it is clear how they performed. Tests were ran with many different combinations of points systems, for example how the AI cooperate when given no points for capturing the food or negative points for proximity. In some cases, it was found that AI would attempt to wait near the food, cornering it, and allow the other AI to eat it instead, to maximise its points.

*All the data and results in these sections can be found at the GitHub link.*

## 8.2 Processing Times

The processing time of effective machine learning algorithms is often very long. Often, machine learning algorithms simulate every play or move set to determine if it is a good play. This form of learning is generally limited solely by the processing power of the computer that is running it.

The way in which the algorithm was implemented means that there is a large variance in the time taken to complete a learning process. As every genome is simulated to completion, and the time taken to complete a simulation is correlated to how well they perform, if a gene pool is performing well, it will take far longer to complete every generation than it would if they were performing badly. The more effective the AI are at solving the task, the longer it takes for each simulation to be processed. In general, the way in which the algorithm has been implemented allows for reasonably fast calculations and simulations. To successfully have a set of genomes learn to fulfill their task using this implementation, it takes around 1000 generations of 1000 genomes. This in total will take around 30 minutes to complete. This is based heavily on the processing power of the machine.

The processing time of the algorithm could be improved in a few ways. Throughout implementation doubles were used to store the weights of the genomes, however, integers and integer multiplication could have been used to ensure that all the calculations were using integers. Integer calculations are significantly faster than multiplication with doubles. Due to the enormous amount of calculations required in each genome, the process time could have been reduced had it been implemented this way instead.

Allowing multiple simulations to be process simultaneously using threads would have reduced the amount of time it took to complete each generation. Being able to simulate multiple sets of AI in parallel was something that was considered during implementation but it wasn't a priority as the processing times at that point were already to a degree that was satisfactory.

## 8.3 Evaluation

Following these results, it is clear the final implementation mostly achieves all of the software aims. The results show that variables can be changed and the network will train appropriately. The UI is created in a way that makes it clear which variables you are changing before you begin the process. You are able to alter which inputs you use and modify values as to anything you want.

The implementation of the environment that the AI are placed into is appropriate at observing all the areas that were specified in the project aims. It allows for variance in performance and flexibility in the use of variables and different inputs.

Every time a genome is trained to completion, it has all of the moves that each of the AI took saved to a text file. This also saves the fitness of the genome. A load feature has been successfully implemented that allows you to easily select a certain genome that has already been trained and display how it performs.

Each genome is presented in a clear and understandable way once the learning process has been completed. The decisions that the AI make are visible and it is easy to differentiate between successful and unsuccessful genomes. This presentation of the genomes is appropriate for determining the difference in behaviour against specific genomes. However, this method falls short when attempting to aggregate mass data of genome performance.

# Chapter 9

# Discussion

## 9.1 Achievements

Through research of different uses of evolutionary algorithms throughout different projects, the algorithm that was settled on was a general purpose algorithm that worked effectively when used in this project. The implementation of the algorithm allowed it to successfully carry out the process of evolution through the crossover and mutate functions.

Each of the results in chapter 8 are obtained when using the software that has been developed throughout this project. The software allows users to change variables and observe how the AI performs in different circumstances and environments. It can be seen in section 8.1.1 that the software effectively evaluates how networks will develop at different rates when tasked with different genome and generation counts. It allows for any amount of genomes to be tested against any number of generations.

As shown in section 8.1.2 you are able to change the combination of inputs that the AI will use during the learning algorithm. It is shown in the results that changing these inputs can have an effect of the performance. There are four different input combinations that can be explored in conjunction with the limitless variable inputs that can be changed.

Observations can be made into the style of cooperation that each genome adopts when trained with certain parameters. The environment that was

chosen for this implementation, provides room for both cooperation and defection. Choosing certain combinations of variables can show somewhat conflicting learned behaviours between the AI.

## 9.2 Deficiencies and Improvements

Although the software is able to complete the task, there are shortcomings of the way in which it is implemented. As stated in the processing times section, the use of doubles throughout the code reduces its efficiency. Through a lack of understanding data structures to a level that could be expected, it was not understood that the use of doubles was detriment until after completion of the project. This would be rectified if the project were to be redesigned and attempted.

Although you are able to alter all the variables which were originally planned, there are still many more that could have been implemented. There are other variables that would greatly affect the significance of cooperation. Being able to change how much one wolf was awarded when allowing the other wolf to be nearby when capturing the food would show whether the wolves were able to learn to share in some capacity.

Due to the nature in which the simulation environment was implemented, it would have been difficult for extra obstacles to be added within the arena. Having these extra obstacles would allow for a much more diverse simulation in which the wolves would have different opportunities to work together in different ways to catch the food. It wasn't apparent that this would be a problem until after both wolves were able to function in the empty arena that was designed. This is a part of the project that wasn't explicitly stated to be an aim, but the project would have definitely produced more interesting results had it been implemented.

Although the software does allow you to view the fitness of different genomes that have been trained and displayed, it does not present these fitnesses in a useful way. Each fitness is tied to the genome which means that it takes a lot more work to compile data on each genome to make an understandable overall picture. Providing a way in which the fitness could be saved separately and in a way that allows for easy understanding would have allowed this project to fulfill its aims in a more effective way.

In actual use of the JAR file to run the software, there isn't any display on

the progress of the network. The Java process will freeze until the network is fully trained. The network will be trained successfully but there isn't an indicator on how far its progressed.

# Chapter 10

# Conclusion

The software was implemented in a way that in some capacity, satisfies the original task that was detailed. An environment was implemented in which multiple AI agents can be trained and tested. The environment allows for observation on the extent that these agents will cooperate. Variables that affect the outcome of the learning algorithms are able to be altered and the resulting performance is easily observable. The UI provided is easy to navigate and understand, and it allows the user to view the performance of previous genomes using the save and load feature. Overall, the task that was detailed in introduction has been completed to a satisfactory degree.

# Bibliography


Bertsekas, D. P. (2011), 'Dynamic Programming and Optimal Control 3rd Edition, Volume II Chapter 6 Approximate Dynamic Programming 6 Approximate Dynamic Programming'.

Binggeser, P. (2017), 'Designing AI: Solving Snake with Evolution'.
**URL:** *https://becominghuman.ai/designing-ai-solving-snake-with-evolution-f3dd6a9da867*

Curran, D. & O'Riordan, C. (2003), 'Evolving Crossover, Mutation and Training rates in a Population of Neural Networks'.

Fabiorino (2017), 'Neural Network Plays Pong'.
**URL:** *https://github.com/fabiorino/NeuralNetwork-plays-Pong*

Fogel, D. B., Hays, T. J., Hahn, S. L. & Quon, J. (2004), A self-learning evolutionary chess program, *in* 'Proceedings of the IEEE'.

Han, J. & Moraga, C. (1995), The influence of the sigmoid function parameters on the speed of backpropagation learning, *in* J. Mira & F. Sandoval, eds, 'From Natural to Artificial Neural Computation', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 195–201.

Korolev, S. (2017), 'Neural Network to play a snake game'.
**URL:** *https://towardsdatascience.com/today-im-going-to-talk-about-a-small-practical-example-of-using-neural-networks-training-one-to-6b2cbd6efdb3*

Kumar, A. & Mishra, R. B. (2013), 'A Parallelized Matrix-Multiplication Implementation of Neural Network for Collision Free Robot Path Planning', *International Journal of Computer Applications* **69**(28), 975–8887.

Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J. & Graepel, T. (2017), 'Multi-agent Reinforcement Learning in Sequential Social Dilemmas'.

Melanie, M. (1998), 'An Introduction to Genetic Algorithms Library of Congress Cataloging−in−Publication Data'.

Melo, F. S. (2001), 'Convergence of Q-learning: a simple proof'.

Montague, P. (1999), 'Reinforcement Learning: An Introduction, by Sutton, R.S. and Barto, A.G.', *Trends in Cognitive Sciences* .

Montana, D. J. & Davis, L. (1989), 'Training Feedforward Neural Networks Using Genetic Algorithms'.

Samuel, A. L. (1959), 'Eight-move opening utilizing generalization learning. (See Appendix B, Game G-43.1 Some Studies in Machine Learning Using the Game of Checkers'.

Severac, Z. (2018), 'Neuroph Studio'.
**URL:** *http://neuroph.sourceforge.net/*

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Van Den Driessche, G., Graepel, T. & Hassabis, D. (2017), 'Mastering the Game of Go without Human Knowledge'.

Skymind (2018), 'DeepLearning4J'.
**URL:** *https://deeplearning4j.org/*

Stathakis, D. (2009), 'How many hidden layers and nodes?', *International Journal of Remote Sensing* .

Steven Miller (2015), 'Mind: How to Build a Neural Network (Part One)'.
**URL:** *https://stevenmiller888.github.io/mind-how-to-build-a-neural-network/*

Thrun, S. (1995), 'Learning To Play the Game of Chess'.

van Gerven, M. & Bohte, S. (2017), 'Editorial: Artificial Neural Networks as Models of Neural Information Processing', *Frontiers in Computational Neuroscience* .

Whitley, D. (1994), 'A Genetic Algorithm Tutorial'.

Zhang (2000), 'Neural Network Structures'.

# Chapter 11

# Appendices

## 11.1 Appendix A - Zip File Contents

- Source Code (SourceCode)
- Report PDF (DeakinDavies1561926.pdf)
- Runnable JAR with data (RunnableJar)
- Git address (git.txt)

**Git address:** https://github.com/Lewisdd/FinalYearProject

## 11.2 Appendix B - Running the software

To run the program, extract the JAR file into its own folder. If the data used for the results is wanted it can also be extracted into the same folder as the JAR. Run the JAR file and the network menu will show up.

### 11.2.1 Training a new network

Input the variables you want to test as shown in section 5.3.1 and click the "Train New Network" button.
The process will freeze for the time taken to create and train the network. This time taken is heavily based on the generation count used. After a while, the display will show.

### 11.2.2 Loading a Network

Click the drop down and select the network you wish to view. Click the "Load Neural Network" button and the display will show.

### 11.2.3 Running via source files

Compile the code and run the Simulater.java class.