Evolving a Learning Agent using Neuroevolution in the FightingICE Game Framework

Deliverable 1: Final Year Dissertation
Heriot-Watt University

Robert John Dunn
H00163867
BSc Honours in Computer Science

Supervisor:
Dr Patricia A. Vargas
School of Mathematical & Computer Sciences
Heriot-Watt University

Co-Supervisor:
Dr Fabrício Olivetti de França
Center of Mathematics, Computing and Cognition
Federal University of ABC, Sao Paulo, Brazil

Second Reader:
Dr Mohamed Abdelshafy
School of Mathematical & Computer Sciences
Heriot-Watt University

Declaration

I, Robert Dunn confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Robert John Dunn

Date: 24/11/2016

Abstract

Neuroevolution is a popular technique for machine learning in which the topology and/or weights of an artificial neural network are adjusted by an evolutionary algorithm. The technique takes inspiration from the evolution of the biological nervous system and is a popular approach for reinforcement learning problems. One way to demonstrate the effectiveness of neuroevolution is through artificial intelligence in games. This project aims to implement a learning agent in the FightingICE platform, a two-dimensional Java fighting game organised and maintained by Ritsumeikan University, Kyoto. The agent is designed to evolve through neuroevolution to improve its performance in the game, eventually becoming competitive versus a human opponent. By implementing a neuroevolution method in a simplistic environment, we hope to evaluate the effectiveness of neuroevolution as a method of machine learning and explore the potential of our agent's performance.

Contents

1 Introduction

1.1 Motivation

The ability to learn is a fundamental attribute of intelligent behaviour [1], in which one acquires knowledge or skills through study, experience, or being taught. The field of machine-learning aims to imitate this learning attribute and apply it to machines in order to improve their performance at tasks without being explicity programmed.

Various methods of machine-learning exist which allow the agent (machine) to adapt itself in order to process new data. One of these methods is neuroevolution, which involves the weights and/or topology of an artificial neural network being adjusted by an evolutionary algorithm. The neuroevolution method is loosely based on the way the biological nervous system operates: with neurons communicating through axons, represented by nodes of the neural network with weighted connections. Neuroevolution has received huge popularity due to the fact that many artificial intelligence problems can be cast as optimisation problems, and since the method is ground in biological metaphor and evolutionary theory. [9]

One way to test the effectiveness of machine-learning methods is through artificial intelligence in computer games. Since the state of a game is usually easily determined (e.g. the hit-points of both player's characters), the effectiveness of the learning method when controlling the agent's actions can be easily evaluated. The FightingICE platform [8], is a two-dimensional fighting game written in Java by a group in Ritsumeikan University, Kyoto. The platform allows programming of artificially intelligent agents within the game and sends the agent delayed information about the current state of the game.

1.2 Objectives

This project aims to implement a neuroevolution algorithm acting on a learning agent in the FightingICE game platform. The inputs to the algorithm will be based on the agent's environment, such as the enemy's current position and the enemy's energy level, the outputs will be character actions e.g. move left, move right, attack. In order to increase the rate of the agent's evolution, an environment using incremental evolution will be used, exposing the agent to progressively more difficult challenges.

Once implemented, we will evaluate the algorithms performance versus a human opponent. We will use human volunteers of varying skill levels in the game, and record their performance versus the agent at different stages of its evolution. We will also compare the effectiveness of the algorithm to that of previous prototypes to evaluate any improvements made.

From this project, we hope to assess the effectiveness of neuroevolution as a method of machine-learning and gain the programming experience from implementing it. We also hope to find the potential to which an agent employing a neuroevolution method to learn can perform. Through comparisons of the algorithm to previous prototypes, we can also assess the progress of our algorithm over time and how improvements affect the agent's performance.

1.3 Professional, Legal, Ethical and Social Issues

1.3.1 Professional Issues

A professional mindset will be employed when approaching this project. The mindset will involve appropriate communication with staff on the project and meeting deadlines reliably. Drafts will also be provided to the project supervisor with suitable time to provide feedback. When dealing with human participants for the agent testing, care will be taken to anonymise the participants data and ensure the data is protected accordingly.

1.3.2 Legal Issues

In order to avoid any legal issues during the project, we will ensure that any code or external work used is correctly referenced and the work holds an appropriate sharing license. This should prevent any copyright claims from outside sources. Since the project is a dissertation project for Heriot-Watt University and is purely for research purposes, there is not likely to be any legal issues arising.

1.3.3 Ethical and Social Issues

The majority of this project will be completed within a controlled, software-development environment, where ethical and social issues are of no concern. When human participants are involved for the agent evaluation, care will be taken to keep the environment safe and risk-free. Since only the use of a computer is involved for the evaluation, there are few potential risks.

2 Literature Review

In this section of the dissertation, we will present any fundamental concepts used in the project. We will also discuss and evaluate relevant work in the field.

2.1 Machine Learning

Learning is a fundamental human function in which we modify our behaviour tendency according to experiences or teaching [1], to become better when a similar situation occurs. In the study of machine-learning, algorithms, computer applications, and systems, utilise learning to improve their performance at certain tasks. There are two main entities in the machine-learning model, the teacher and the learner. The teacher, while not always available, contains the knowledge to perform a given task while the learner has to learn the knowledge the teacher holds. [16] There are three types of machine learning:

- *Supervised Learning*
  A teacher provides the learner with a set of input and desired output pairs. The learner can then use these examples to improve its performance at the task.

- *Unsupervised/Self-organised Learning*
  There is no teacher, so the learner learns based only on the observed relationships. A common application is finding patterns or grouping within data, using methods such as cluster analysis.

- *Reinforcement Learning*
  The agent uses goal-directed learning where a notion of reward is introduced and the agent attempts to maximise this reward. There is no teacher for the agent and no explicit model of the environment. Computational approach to learning from interaction. [3]

Another way to categorise machine-learning methods is by the output of the method. The output of the method is usually a good indication of the application of the method, for example, two classes of outputs 'spam' and 'not spam' in spam filtering.

- *Classification*
  Inputs are assigned to two or more classes (groups) of output. Typically used with a supervised learning method. One application is spam filtering.

- *Regression*
  Supervised learning problem similar to classification, however the outputs are continuous rather than discrete.

- *Clustering*
  A set of inputs to be divided into groups. Unlike classification, groups are not defined beforehand. Typically an unsupervised learning task.

- *Density Estimation*
  Finds the distribution of inputs in some space.

- *Dimensionality Reduction*
  Inputs mapped into a lower-dimensional space.

Machine-learning is being applied extensively in modern day life, though not everyone is aware of it. The personalisation of search engine results and social networking feeds are one application, providing user's with appropriate content. [4] Other examples of machine-learning application include optical character recognition (OCR) and computer vision. [5]

2.2 Neuroevolution

2.2.1 Fundamentals

Neuroevolution is a biologically-inspired method of machine learning in which an artificial neural network is evolved using an evolutionary algorithm. The biological inspiration for the method comes from the nervous system in which neurons communicate through axons, translated to machine-learning by the nodes of a neural network communicating through weighted connections.

The method has proved popular, especially in the fields of artificial life, evolutionary robotics and computer games. One reason for its popularity is the fact that neuroevolution is a form of reinforcement learning, which can be applied more generally than its counterpart, supervised learning. The popularity also stems from the fact that the method is ground in biological metaphor and evolutionary theory. The method consistently performs well in many areas of application and can handle large action/state spaces. [9]

The applications of neuroevolution are numerous and diverse. One application is the implementation of artificially intelligent agents in computer games (see section 2.2.5). Other applications include dynamic resource allocation, optimising manufacturing processes and even creating musical melodies. [7]

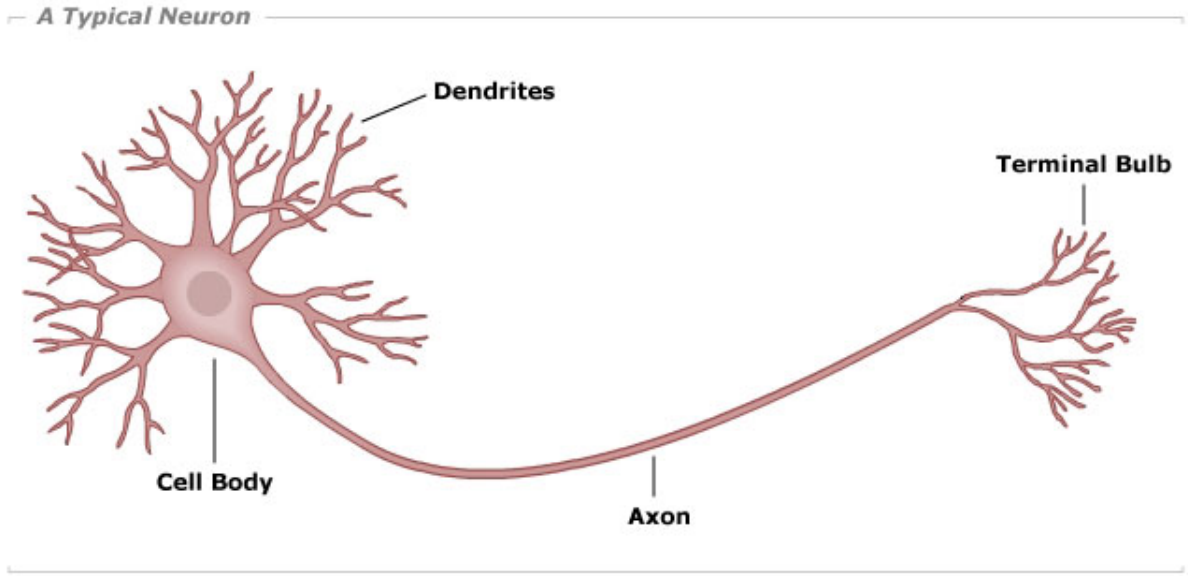


Figure 1: Simple diagram of a biological neuron.

2.2.2 Artificial Neural Networks

Artificial neural networks model the way the brain solves problems with collections of neurons communicating through axons, represented with nodes of the network with weighted connections.

The most basic version of an ANN (artificial neural network) is a perceptron. A perceptron is a simple machine which takes a variable number of input nodes which connect to a single output node. [17] The concept of the perceptron was extended by means of the the multi-layer feed forward network, which incorporates extra 'hidden' layers of nodes between the input and output. The more hidden layers there are, and the more nodes in these layers, the more complex behaviour a network can experience. [18]

Neural networks can be employed in a variety of ways, for both supervised and unsupervised/reinforcement learning problems. Networks can have an associative (content-addressable) memory and can also implement parallel processing across individual nodes. [2] Common applications for neural networks include character recognition, image compression and stock market prediction. [19]

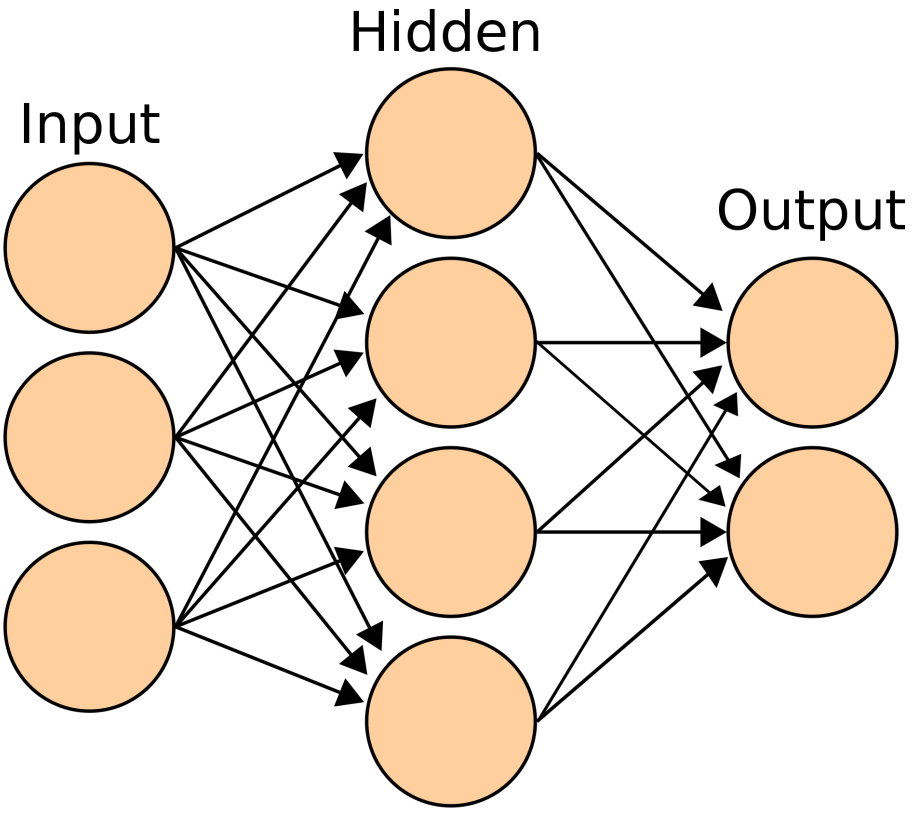


Figure 2: Multi-layer feed forward network with labelled layers.

Activation Function

In order to calculate the output of a certain node in a network given the input or set or inputs, we use an activation function. [20] This output of the node is usually saturated to a value between minus and positive one by the function and then scaled appropriately when output. [16]

The simplest activation function is the step function [2], where if the weighted sum of the inputs to the node falls below a certain threshold the output is zero, else the output is one. This can be thought as the node either sending signals or not sending signals. The step function is used in perceptrons and often shows up in other models. [17] Since there are only two possible outputs of nodes using the step function, it is common to use different, more general functions.

One of these more general, non-linear activation function is the hyperbolic tangent function tanh, shown in figure 3. This allows nodes to have an activation output of a real number between negative one and one. The usage of these more general functions allows us to exhibit more complex behaviour within the network.

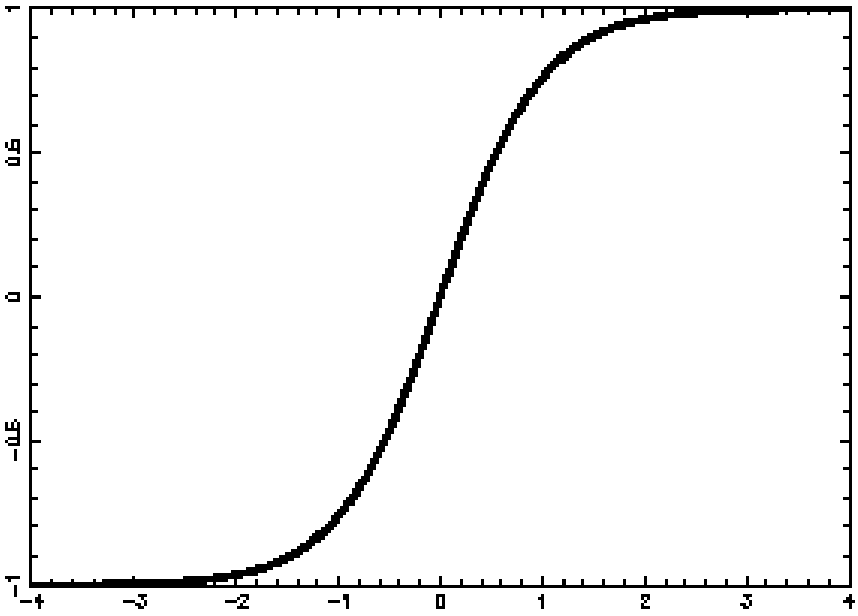


Figure 3: tanh activation function plot.

2.2.3 Evolutionary Algorithms

Evolutionary algorithms are methods of optimisation, where potential solutions to the problem are seen as individuals of some population. Each individual in the population is assigned a fitness value which is calculated according to the performance of the provided solution, and the algorithm works to find the best (fittest) of these solutions. Using techniques inspired by biological evolution such as reproduction, mutation, recombination, and selection, new populations of solutions can be generated and assessed. Using evolutionary algorithms allows fine tuning of the search space through constants such as rate of reproduction and mutation rate. [21]

The implementation of an evolutionary method involves three steps which are repeated indefinitely [6]:

1. Generate population of initially random individuals.
2. Evaluate the fitness of each individual in the population.
3. Update and replace population with methods such as crossover and mutation.

2.2.4 Incremental Evolution

When an agent is provided with a complex behaviour, it may have trouble evolving to perform at its potential. Using incremental evolution, the behaviour can be learnt incrementally with tasks gradually increasing in difficulty. This form of evolution proved effective in at least one implementation [12]. The agent's task was a prey capture task: the agent moves through the environment and must catch its prey before the set number of time-steps. With increasingly difficult tasks, the agent was able to rapidly improve its performance, and skip many potential generations of evolution.

We hope to experiment with the usage of incremental evolution in our project to assess whether it has a positive impact on the evolution of the agent.

2.2.5 NEAT (NeuroEvolution of Augmenting Topologies)

NeuroEvolution of Augmenting Topologies (NEAT) is a method of neuroevolution developed by Ken Stanley in 2002, which involves both the weights and the topology of the ANN being altered. The method is a genetic algorithm and involves applying the following three techniques: [22]

1. Track genes with history markers to allow crossover among topologies
2. Apply speciation to preserve innovations
3. Developing topologies incrementally from simple initial structures

A notable extension of NEAT which could potentially be appropriate for the project is HyperNEAT. HyperNEAT is a hypercube-based extension of the method developed by the Evolutionary Complexity Research Group at UCF. [24] [23]

2.3 FightingICE

2.3.1 Game Platform

FightingICE is a Java based game platform organised and maintained by Intelligent Computer Entertainment Lab., Ritsumeikan University. The game is based in an arena where two fighters are competing versus each other, attempting to deplete the other's hit-points while preserving their own. The FightingICE platform was designed to allow easy development and evaluation of artificially agents in the game for research or hobby purposes. Once implemented, an agent receives information about the state of the game from the platform periodically, such as the opponent player's location and current energy levels. A delay is added to this game information in order to simulate the delay a human player would experience from reaction time. [8]

2.3.2 Game Agents

There are four characters available in the game: Zen, Garnet, Lud, and Kfm. Each of the characters is capable of moving, performing attacks, and combining these attacks into unique combos. The game starts with each player at 0 hit-points and 0 energy. Once a player successfully connects an attack against its opponent, the player's energy is increased and the opponent's hit-points decrease. In order to compete against an opponent, the character must dodge opposing attacks and make effective use of energy to land attacks and reduce the opponent's hit-points.

Figure 4: Screenshot of the FightingICE platform in action.

2.4 Related Work

When implementing neuroevolution, a computer game is a common choice of environment. Neuroevolution methods have been used as artificial intelligence in games in the following external projects:

- Improving AI for simulated cars. [10]
- Neuroevolution approach to general video game playing. [15]
- Calculating optimal jungling routes in the game DOTA2. [13]
- Using neuroevolution to play atari games. [14]

The work has found neuroevolution to be an effective machine-learning algorithm in a computer game environment and the method proved to operate consistently across a diverse range of games.

There are also numerous agents programmed in the FightingICE platform to compete in competitions. [8] We will test and assess the performance of a select few agents at a later stage of this project. Hopefully from this assessment of the agents we can evaluate the strategies of the programmer to improve the agent's performance as well as the effectiveness of different methods of AI.

3 Organisation

In this section we present the decision process on how to approach the task including analysis of the task itself, analysis of the requirements, and analysis of the risks. A gantt chart of the project timeline is also included.

3.1 Project Task Analysis

The objective of this project is to evolve an agent in the FightingICE game platform with a neuroevolution method. The agent will be evolved to the point of being competitive versus a human opponent.

In order to implement the neuroevolution method, we will first need an artificial neural network to control the actions of the agent. The neural network will need to take as input various factors from its environments such as the location of the opposing player, the energy of the opposing player and whether the opponent is performing an attack. The outputs of the neural network will be actions for the character, for example move left, jump, attack. Once a neural network controlling the agent has been implemented, the network must be evolved to imitate the learning process. This will involve deciding on a suitable evolutionary algorithm and the encoding of the network's genotype. It will also need to be decided whether only the weights of the network are evolved, or also augmenting the topology (NEAT) and/or altering the activation function.

To improve the rate of evolution for the agent, we will create an incremental evolution environment where the agent is exposed to tasks of gradually increasingly difficulty. To implement this environment, we will limit the capabilities of the training opponent and gradually return them.

To test whether the agent's performance has improved to the point of being competitive versus a human opponent, we will find volunteers of varying skills to face our agent in the game. The results of the player versus our agent at various stages of its evolution can then be evaluated to assess the effectiveness of the neuroevolution algorithm which we implemented.

3.2 Requirement Analysis

3.2.1 Table of Requirements

No.	Requirement	Priority	Predecessors
1	Implement a neural network to control agent's behaviour	High	
1.1	Initialise a network with random weights, test if the sensors can be read and the actions can be output	High	
1.2	Implement a simple rule-based agent for the agent to compete versus	High	1.1
1.3	Test and execute at least one agent from the FightingICE AI competition	Medium	
2	Implement an appropriate evolutionary algorithm to evolve the neural network	High	1
2.1	Test whether a simple evolutionary algorithm evolving the weights of the network improves the agent's performance	High	1
2.2	Adapt and use the HyperNEAT method, test whether it improves on the previous algorithm	Medium	1, 2.1
3	Create an incremental evolution environment for the agent	Medium	1, 2
3.1	Limit the capabilities of the opponent and incrementally return them to test whether speed of evolution is improved	Medium	1, 2
3.2	Evolve against a simple agent, replace opponent with best agent after convergence and continue	Medium	1, 2
4	Evaluate the agent's performance versus a human opponent at various stages of its evolution	High	1, 2

3.2.2 Requirements Textual Descriptions

1 - Implement a neural network to control agent's behaviour

To implement an agent in the FightingICE platform controlled by a simple neural network.

1.1 - Initialise network with random weights, test if sensors can be read and actions can be output

In order to test whether sensors can be read and actions can be output, initialise a simple neural network with random weights to control the agent. Ensure game information is being received and number of outputs nodes match character actions.

1.2 - Implement a simple rule-based agent for the agent to compete versus

Create agent in FightingICE which follows simple rule-based logic. Use as an opponent for the neuroevolution agent while it is evolving.

1.3 - Test and execute at least one agent from the FightingICE AI competition

Find agents from the FightingICE AI Competition and test them to see strategies of other programmer's work. Attempt to read and understand the code.

2 - Implement an appropriate evolutionary algorithm to evolve the neural network

Find and implement an appropriate evolutionary algorithm to evolve the ANN of the agent. Decide on implementation based on performance evaluation.

2.1 - Test whether a simple evolutionary algorithm evolving the weights of the network improves the agent's performance

Implement a simple EA such as hill-climbing or an algorithm using only mutation to evolve the agent's neural network. Test whether the evolution improves the agent's performance and if so, to what extent.

2.2 - Adapt and use the HyperNEAT method, test whether it improves on the previous algorithm

Replace the previous evolutionary algorithm with an implementation of the HyperNEAT or other appropriate neuroevolution method. Test the new implementation versus the rule-based agent to test whether it improves on the previous algorithm.

3 - Create an incremental evolution environment for the agent

Create an incremental evolution environment to speed the process of evolution for the agent.

3.1 - Limit the capabilities of the opponent and incrementally return them to test whether speed of evolution is improved

Initially, completely limit the capabilities of the agent's opponent and gradually return these capabilities over time. Evaluate whether gradually exposing the agent to these capabilities improved the speed of evolution of the agent.

3.2 - Evolve against a simple agent, replace opponent with best agent after convergence and continue

Begin evolution with simple rule-based opponent. Once convergence in evolution occurs, replace the agent's opponent with the current best agent. Repeat this process incrementally and compare performance of each opponent with that of its successors.

4 - Evaluate the agent's performance versus a human opponent at various stages of its evolution

Test the performance of different agents at different stages of their evolution against human players. Attempt to find volunteers that rank at different skill levels from novice to expert. From this we can assess whether the agent was evolved to become competitive versus a human opponent.

3.3 Performance Assessment

3.3.1 Table of Project Prototypes

Objective	Date	Assessment
Prototype 1: Agent using neural network to sense environment and output simple actions in FightingICE	15/12/2016	Agent can proficiently perceive its environment and output simple actions
Prototype 2: Agent controlled by neural network being evolved by simple evolutionary algorithm, altering only the weights of the network	20/01/2017	Evaluate agents performance versus simple AI opponent at different stages of its evolution
Prototype 3: Agent evolved with HyperNEAT or other appropriate neuroevolution method	10/02/2017	Evaluate agent's performance versus AI and compare whether method was more effective than simple EA
Prototype 4: Agent with appropriate learning method implemented in incremental evolution environment	10/03/2017	Agent's environment is adapted to utilise incremental evolution. Agent's performance assessed and compared versus learning without incremental environment.

3.3.2 Prototypes Textual Descriptions

Prototype 1

The initial prototype will involve an agent using an artificial neural network to test if the agent's environments can be sensed and the agent can output actions on the platform. Once we have evaluated the agent's functionality we will move towards evolving the neural network.

Prototype 2

The second prototype will implement a simple evolutionary algorithm in order to alter the weights of the neural network. With an appropriate fitness function for the agent, this method of evolution should result in improvement of the agent's performance. The agent's performance versus both a simple rule-based opponent and a human opponent will be evaluated.

Prototype 3

The third prototype for the project will involve implementing the HyperNEAT neuroevolution method for the agent. Should we find another, more appropriate, neuroevolution method to apply, we will take it into consideration. Once implemented, the agent utilising the HyperNEAT algorithm will be assessed versus both the rule-based and a human opponent.

Prototype 4

The final prototype for the project will utilise incremental evolution techniques to improve its rate of evolution. This prototype is due to be finished by the 10th March 2017 and will mark the end of our agent development. Once the final prototype has been completed, evaluation of the agent's performance will commence.

3.4 Risk Assessment

3.4.1 Table of Risks

No.	Risk Name	Probability	Response
1	Laptop failure	Low	Online storage (github), backups
2	Difficulties with Java programming	Low	Spend more time familiarising with language and practising
3	Difficulties with FightingICE platform	Medium	Consult tutorials and relevant documentation from website

3.4.2 Risks Textual Description

1 - Laptop failure

In the case of the laptop being used to develop the project failing, essential code or documentation could be lost. To reduce the impact of this risk, we will ensure to regularly upload the code to a online repository such as GitHUB. We will also back-up any work on external hard-drives or on the university computers.

2 - Difficulties with Java programming

To avoid any problems with lack of Java programming expertise, we will spend time familiarising ourselves with the language before development.

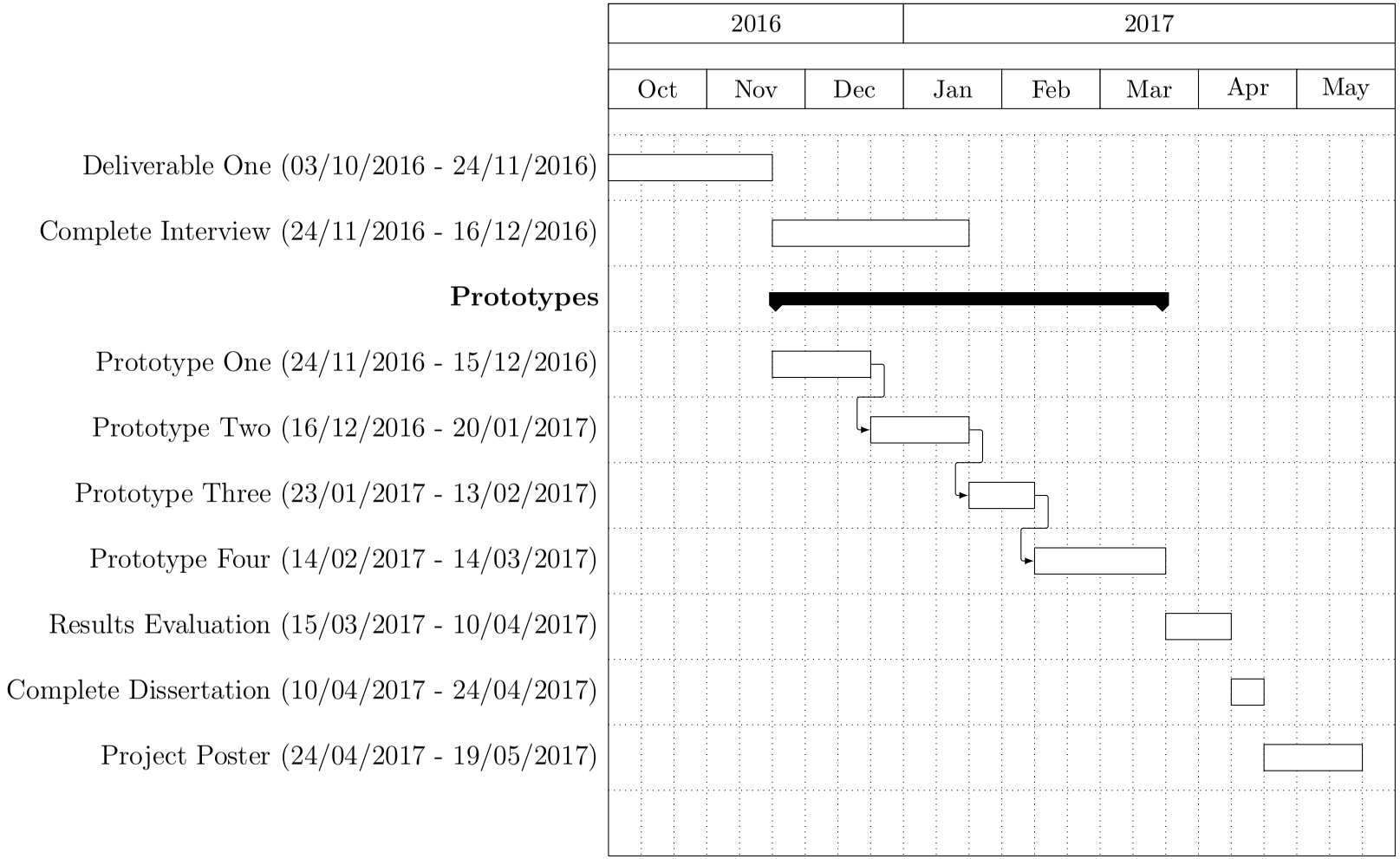
3 - Difficulties with FightingICE platform

Since there is plenty of information and lots of tutorials on agent development with the platform, we should be able to easily familiarise ourselves with it.

3.5 Project Plan

Figure 5 shows a gantt chart representing a timeline for the completion of the project including all prototypes, the dissertation and the project poster. With this project plan we hope to leave appropriate time to evaluate any results as well as carefully thought out prototype deadline which will result in any code problems occurring early. The plan has been shown to and approved by both the project supervisor and co-supervisor.

3.5.1 Gantt Chart



4 Methodology and Results

4.1 Prototype One: Artificial Neural Network in Game Framework

The first prototype of the dissertation was to implement an artificial neural network which takes inputs from the game framework and whose outputs correspond to agent actions in the game. With this prototype we aim to familiarise ourselves with the workings of the framework and artificial neural networks.

4.1.1 Design

In order to better understand the theory and solidify the mathematics behind neural networks, this prototype will be implemented without the use of any external Java libraries. The neural network will be a feed-forward network, meaning information will move in only one direction (forward) from the input nodes. The network will consist of three layers: an input layer, a hidden layer, and an output layer. The number of nodes in the hidden layer will later be optimised to an appropriate value through experimentation.

Each node in the input layer of the network will correspond to a piece of game information available from the CommandCenter class provided with the FightingICE framework. The CommandCenter provides delayed data to the AI such as positioning of both players, player energy, player hit-points and more. For this prototype, I decided to use only three inputs: the x distance between the two characters, y distance, and the agent's energy. In order to evolve a competent agent at later stages, more inputs will be included such as information about the player's current action. However since this prototype is solely to test the neural network and begin working with the game framework, I decided to keep it simple for the time being.

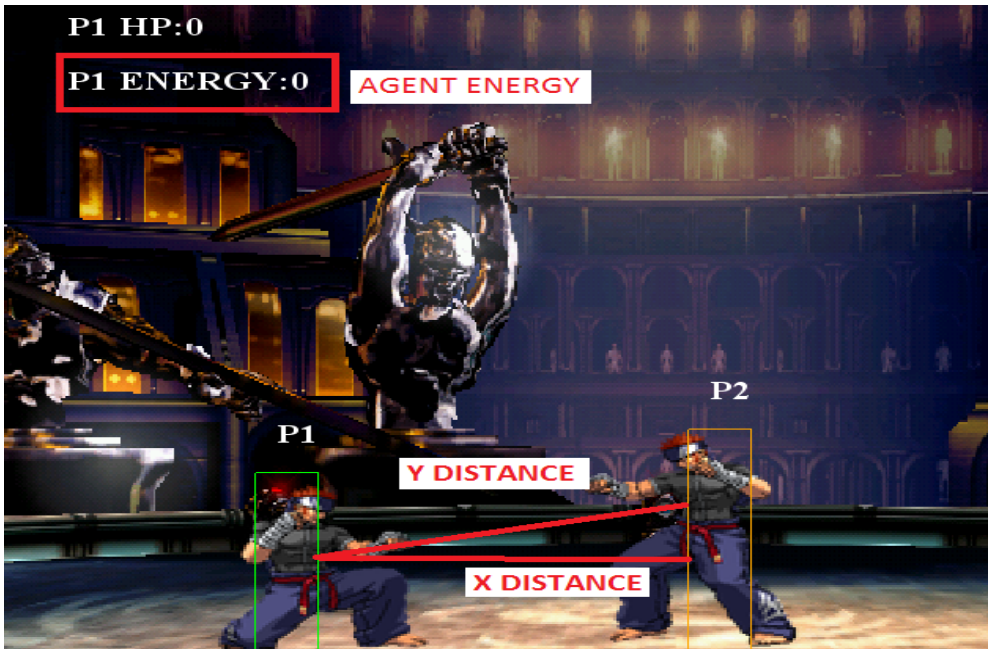


Figure 5: Labelled inputs which are taken by the artificial neural network.

There will be seven output nodes for the neural network, for each of the possible actions of the agent: up, down, left, right, attack A, attack B, and attack C. The output of the network will determine whether each action is undertaken. For example, if the output nodes corresponding to right and attack A are activated, the agent will move right and attack with attack A. If the implementation is successful, the agent should exhibit different outputs (actions) based on the current state (position of both characters etc) of the game.

4.1.2 Implementation

Prototype1.java

The first class implemented was Prototype1 which is used to control the agent. The class implements the AIInterface interface, provided by the game framework. The interface provides methods for initialising the agent, processing game information, and sending agent outputs (actions) to the game.

The initialize() method initialises the various classes used by the agent. The Key object is used to output agent action and contains seven boolean variables (A, B, C, U, D, L, R) representing all possible actions. During each frame of the game, the Key object is read and the agent performs the appropriate action(s). The initialize method also initialises the FrameData object which contains information about the current game state such as frames left in the round and visual information, and the CommandCenter object which provides delayed game information to the agent with character positions and other useful data. The ANN class used for feeding for the neural network is also initialised in this method.

```
@Override
public int initialize(GameData gameData, boolean playerNumber) {
    // TODO Auto-generated method stub
    gd = gameData;
    p = playerNumber;
    inputKey = new Key();
    fd = new FrameData();
    cc = new CommandCenter();
    ann = new ANN();
    return 0;
}
```

Other methods used by the AIInterface include the getInformation() method which is used to set data for the FrameData object using the setFrameData() method provided from the CommandCenter object. Also included from the interface is the input() method which returns a Key object representing the agent's output, the close() method which is called when the game is closed and can be used to finalise any processing, and getCharacter() which returns the character of the agent: in this case 'CHARACTER_ZEN'.

The final method used in this class is the processing() method. This method is called at every frame and is used for processing game information and determine the agent's actions. The method first ensures that frame data has been provided using the methods getEmptyFlag() and getRemainingTime() from the FrameData object. The X-distance, Y-distance and agent energy are then used to feed through the neural network with the feed() method. Before feeding, the inputs are normalised using the tanh() (hyperbolic tangent) method from the Java Math library. Feeding through the network returns an array of seven doubles, representing the output of the neural network. Following feeding through the network, variables in the Key object are set to either true or false depending on the output. Since neuron activations in the network are between minus one and one, if the output is greater than zero then the variable (action) is set to true. Once set, at the next frame when the input() method is called the agent outputs the appropriate actions.

ANN.java

The next class implemented was ANN which provides methods to create an artificial neural network and feed through the network. Weights of the network are stored in two matrices of doubles: weights0 and weights1, each representing a synapse of the network (input layer to hidden layer, and hidden to output layer).

The class constructor initialises these matrices with an appropriate number of weights. For this prototype three input nodes were used, ten hidden layer, and seven output layer nodes. A Java Random object from the Java.util library is then used to set these weights to random values between minus one and one. The Random object provides a method nextDouble() which returns a random double value in between zero and one. In order to generate a value between minus one and one, which is needed for the neural network, this value is multiplied by two and then one is subtracted from the result.

The sole method provided by this class, apart from the constructor, is feed() which takes three double inputs as arguments and returns an array of doubles which are the output of the network. The method initially declares and initialises two arrays of doubles to store the activations of neurons in the hidden and output layer. Next, inputs are fed through to the hidden layer by calculating the weighted sum for each node in the hidden layer using the inputs and the first synapse matrix (weights0). After, output activations are calculated: also using weighted sums however this time using the second synapse matrix, weights1. A message is printed to the standard output for each output activation for testing purposes and finally the array of doubles for the output activations is returned.

```
/* Feed through the network with x-distance, y-distance and energy */
public double[] feed(double x, double y, double e) {
    double[] hid = new double[10];      // Hidden layer nodes
    double[] out = new double[7];       // Output layer nodes
    for(int i=0;i<10;i++) {          // Feed through to hidden layer
        hid[i] = (double) (weights0[0][i] * x      // X
            + weights0[1][i] * y                   // Y
            + weights0[2][i] * e);                 // Energy
    }
    for(int j=0;j<7;j++) {   // Feed through to output
        out[j] = 0;
        for(int k=0;k<10;k++) {
            out[j] += (weights1[k][j] * hid[k]);
        }
        System.out.println(j+"th output: "+out[j]);
    }
    return out;
}
```

4.1.3 Evaluation

Since this prototype was to familiarise with the game framework and artificial neural network theory, there was no need to evaluate the agent's performance versus another artificial intelligence or human player. The agent was able to successfully read inputs from provided game data and feed through the neural network to send outputs to the game framework. This resulted in the agent's action changing based on the current game state (location of both characters and agent energy).

One shortcoming of the prototype was the agent only rarely attacking. This was due to the fact that if the agent attempted to output more than one attack at once (e.g. attack A and B are both activated) then they cancel each other out. In future prototypes this problem can be avoided by using if and else statements to ensure that only one attack is output at a time, regardless of whether more than one is activated in the neural network. Similarly, the agent jumped excessively due to the fact that jumping is prioritised over other movement commands. To avoid this in future prototypes, the activation needed for jump to be output can be increased to make it less likely to occur. Both problems could also potentially be avoided during the evolution of the agent, with the agent evolving to a point where it is able to avoid these conditions.

Other improvements which can be applied to future prototypes is with the inputs used for the neural network. Using the X and Y distance between the two characters does not allow the agent to determine whether the opponent is on its left or right side. Instead, using the x and y co-ordinates of both characters will allow the agent to adapt its behaviour based on which side of the agent the opponent is positioned. Inputs for the network should also be properly normalised and scaled. Using the hyperbolic tangent function results in values between minus one and one, as needed, though since the maximum and minimum values of the inputs are known (e.g. minimum X co-ordinate is -120 and maximum is 680) we can properly scale and normalise them to the required range. This should provide more effective and accurate activations for the neural network, resulting in improved interactions with the agent and its environment.

A final way to improve the agent in future prototypes is to run experiments to determine the number of hidden nodes needed in the network. Since the number of input and output nodes in the network depends on the game framework, they will remain static, though different numbers of hidden layer nodes will have an impact on the effectiveness of the neural network, and therefore the agent's performance. During evolution, experiments can be run to evaluate the agent's improvement over time and determine an appropriate number of hidden layer nodes to use, as well as being able to optimise various other network parameters.

4.2 Prototype Two: Weights of Neural Network Evolved by Genetic Algorithm

For the second agent prototype, an evolutionary algorithm was used to evolve the weights of the artificial neural network. This will be the first prototype to make use of the neuroevolution method. Also, the code for the neural network was rewritten in order to use the dl4j Java library.

The aim of this prototype is to provide the agent with the ability to train itself. With training, the agent will be able to improve its fitness (and therefore performance) in the game over time. Another aim of this prototype is to make use of the dl4j library in preparation for prototype three. The library provides any functionality that is needed for the neural network in this project and is known to be a robust and reliable library. Advanced network configurations in the library may also aid in the implementation of the HyperNEAT method.

Figure 6: dl4j logo

4.2.1 Design

Artifical Neural Network

The artificial neural network will be implemented using the Java library deeplearning4j (dl4j), a deep learning library for the JVM. The library provides appropriate classes to implement neural networks, as well as OpenBLAS, which provides CPU optimisations to ease use of the network and run calculations in parallel. From the dl4j library we will use the MultiLayerConfiguration and MultiLayerNetwork classes to implement our adaptation of an artificial neural network.

This prototype also aims to improve the processing of inputs to the neural network. In the first prototype, X and Y distance between were provided as inputs, though since the distance will always be positive it was not possible to determine whether the opponent was on the left or right side of the agent. Using the X and Y coordinates of both characters will allow the agent to determine both the distance between the two, and also the direction of the distance. Another problem in prototype one was the fact that if the agent attempted to output more than one attack at once, no attack was output. If the agent's evolution does not fix this we will implement IF-ELSE loops to only allow the agent to output a single attack per frame. This prototype will also properly scale and normalise inputs to the desired -1,1 range without the use of the hyperbolic tangent function. Since we have access to the minimum and maximum values of each variable we can properly scale them so therefore the the far left X-coordinate of the arena corresponds to an input of -1 and the far right to an input of 1.

Genetic Algorithm

Since this prototype will be implementing the neuroevolution method, we will also need to implement a genetic algorithm which will modify (evolve) the neural network. The genetic algorithm will need to implement methods for generating the initial population as well as appropriate methods for crossover and mutation to evolve the population. A storage method for evolution data will also need to be decided.

Selection for crossover in the genetic algorithm will be completed using tournament selection. With tournament selection, individuals are chosen at population at random and a tournament is run of which the individual with the best fitness wins. The tournament size is predetermined and will be optimised for later prototypes. Mutation will be applied by replacing the mutated weight with a random value from the vector [-1, 1].

In order to evaluate the agent's evolution and load previously evolved agents we will need a method of storing all relevant data. From inspecting other evolution projects implemented in the FightingICE framework (cite), we decided to use comma-separated values (CSV) files. The agent will create a different file for each generation of the evolution and will first write the weights of each genotype to the file. This will allow us to access previous agents and is used by the AI to update network weights. After each genotyype is run it will also write its fitness to the file to allow evolution of the population with crossover and mutation.

Parameter Tuning

In order to optimised parameters of the neural network and genetic algorithm we will run experiments. These experiments will involve running the neuroevolution with different parameters and evaluating the speed of the agent's evolution and other relevant data. For example, the agent will be run with different population sizes to determine an appropriate value for evolution in which the population is large enough to allow for diversity within genotypes but not too large to the point of extending runtime without improving results.

We will also use similar experiments to determine a suitable value for the number of nodes in the hidden layer. The number of neurons will impact the complexity of the network and we aim to find an optimised value for the application of our agent. We can also determine from experiments whether three layers is sufficient for the network or more need to be added.

5 Testing and Performance Assessment

6 Conclusions

7 Appendices

Figure 1

Image of a biological neuron
https://online.science.psu.edu/sites/default/files/bisc004/content/neuron.jpg

Figure 2

Image of a simple multi-layer feedforward artificial neural network.
https://upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Artificial_ neural_network.svg/2000px-
Artificial_neural_network.svg.png

Figure 3

Plot of the tanh activation function.
http://www.mpe.mpg.de/ ott/dpuser/tanh.png

Figure 4

Screenshot of the FightingICE game platform.
https://i.ytimg.com/vi/_Poz6qwIwbc/maxresdefault.jpg

Figure 5

Gantt chart of the project timeline.
Created in GanttProject application.

8 References

References

[1] Michalski, R., Carbonell, J. and Mitchell, T. (1983). Machine Learning: An Artificial Intelligence Approach. pp. 5-20.

[2] Wilde, P. (1997). Neural Network Models. [book] pp. 4-14.

[3] Barto, S. and Sutton R. (1998). Reinforcement Learning: An Introduction. pp. 23-30.

[4] McCallum, A., Nigam, K., Rennie, J. and Seymore, K. (2001). A Machine Learning Approach to Building Domain-Specific Search Engines. [online] Available at: http://www.kamalnigam.com/papers/cora-ijcai99.pdf [Accessed 23 Nov. 2016]

[5] Wernick, M., Yang, Y. and Strother, S. (2010). Machine Learning in Medical Imaging. [online] Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4220564/ [Accessed 23 Nov. 2016]

[6] Prins, C. (2004). A simple and effective evolutionary algorithm for the vehicle routing problem. [book] pp. 20-35.

[7] Neural Networks Research Group (2014). Research on Neuroevolution Applications. [online] Available at: http://www.cs.utexas.edu/users/nn/pages/research/ne-applications.html [Accessed 23 Nov. 2016]

[8] Intelligent Computer Entertainment Lab, Ritsumeikan University. (2010). FightingGameAICompetition. [online] Available at: http://www.ice.ci.ritsumei.ac.jp/ ftgaic/ [Accessed 23 Nov. 2016]

[9] Risi, S. and Togelius, J. (2015). Neuroevolution in Games: State of the Art and Open Challenges. [online] Available at: https://arxiv.org/pdf/1410.7326.pdf [Accessed 03 Nov. 2016]

[10] Pace, A. (2014). Improving AI for simulated cars using Neuroevolution. [online] Available at: http://commerce3.derby.ac.uk/ojs/index.php/gb/article/view/3/1 [Accessed 07 Nov. 2016]

[11] Lampropoulos, A. (2005). Machine Learning Paradigms. [online] Available at: http://file.allitebooks.com/20150722/Machine%20Learning%20Para-digms-%20Applications%20in%20Recommender%20Systems.pdf [Accessed 18 Nov. 2016]

[12] Gomez, F. and Miikkulainen, R. (1997). Incremental Evolution of Complex General Behaviour [online] Available at: http://nn.cs.utexas.edu/downloads/papers/gomez.adaptive-behavior.pdf [Accessed 18 Nov. 2016]

[13] Batsford, T. (2014). Calculating Optimal Jungling Routes in DOTA2 Using Neural Networks and Genetic Algorithms. [online] Available at: http://commerce3.derby.ac.uk/ojs/index.php/gb/article/view/14/12 [Accessed 18 Nov. 2016]

[14] Hausknecht, M., Lehman, J. and Stone, P. (2014). A Neuroevolution Approach to General Atari Game Playing. [online] Available at: https://www.cs.utexas.edu/ mhauskn/papers/atari.pdf [Accessed 15 Nov. 2016]

[15] Braylan, A., Hollenbeck, M., Meyerson, E. and Miikkulainen, R. (2015). Reuse of Neural Modules for General Video Game Playing [online] Available at: https://arxiv.org/pdf/1512.01537v1.pdf [Accessed 17 Nov. 2016]

[16] Dehuri, S., Ghosh, S., and Cho, S-B (2011). Integration of Swarm Intelligence and Artificial Neural Network. [book] P1-P23

[17] Widrow, B. and Lehr, A. (1990). 30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. [online] Available at: https://pdfs.semanticscholar.org/8b73/adda1-5fa71b0a35ffedb899d6a72d621923b.pdf [Accessed 24 Nov. 2016]

[18] Hornik, K. (1989). Multilayer Feedforward Networks are Universal Approximators. [online] Available at: http://deeplearning.cs.cmu.edu/pdfs/Kornick_et_al.pdf [Accessed 24 Nov. 2016]

[19] Stanford University. (2013). Applications of neural networks. [online] Available at: http://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Applications/index.html [Accessed 24 Nov. 2016]

[20] Hornik, K. (1990). Approximation Capabilities of Multilayer Feedforward Networks. [online] Available at: http://zmjones.com/static/statistical-learning/hornik-nn-1991.pdf [Accessed 24 Nov. 2016]

[21] Vavak, F. and Fogarty, T. (1996). Comparison of Steady State and Generational Genetic Algorithms for Use in Nonstationary Environments. [online] Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.55.3585-&rep=rep1&type=pdf [Accessed 24 Nov. 2016]

[22] Kenneth, S. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. [book] pp. 1-30.

[23] Kenneth, S., D'Ambrosio, D. and Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. [book] pp. 05-09.

[24] Evolutionary Complexity Research Group at UCF. (2003). The hypercube-based neuroevolution of augmenting topologies (HyperNEAT). [online] Available at: http://eplex.cs.ucf.edu/hyperNEATpage/ [Accessed 24 Nov. 2016]