Playing Games With Machine Learning

Connor John Macaskill

May 2017

Abstract

This paper describes the development of a system capable of independently learning to effectively play a variety of games. This is achieved through the use of a variety of machine learning techniques which will be detailed in this document, alongside any theory involved.

Project Dissertation submitted to Swansea University
in Partial Fulfilment for the Degree of Bachelor of Science

Prifysgol Abertawe
Swansea University

Department of Computer Science
Swansea University

Contents

1 Introduction

1.1 Project Introduction

First emerging in the 1980s, Machine Learning is a type of artificial intelligence that allows a computer to learn how to do a task without explicitly being programmed for that task. Algorithms which incorporate machine learning are typically used to develop a solution to problems for which designing an explicit algorithm is infeasible due to either the scale or the nature of the task. Examples of this include fields such as image recognition, optical character recognition, search engines and spam filtering where solutions which incorporate machine learning algorithms have proven to be very effective. In terms of scale machine learning has also been used to great effect in the processing of massive amounts of data, detecting patterns or extracting the more useful information. For example machine learning algorithms are used by NASA to detect occurrences of different space phenomena such as stars or planets in images taken by the Hubble space telescope. At its core the subject's main focus is on the development of algorithms that, when exposed to new data, can teach themselves to perform better at a task. There are many different approaches to this, some of which can be found in the background section of this paper.

The term "Machine Learning" has become somewhat of a buzzword in the technology world, from job advertisements to start-up companies the ill defined term is popular as a phrase used in order to draw interest and attempt to stand out. Alongside this recent trend the field has seen rapid increases in visibility in the public eye, from Google's AlphaGo's recent victories against a champion in playing GO [1] to Deep Mind's success in playing atari games the field is one which is rapidly growing. A particular focus has been placed on self driving cars [2][3][4], a technology which is promised to be revolutionary if/when it reaches the consumer market. However, this technology may not be seen as beneficial by everyone. Just as industrialisation put farm workers out of a job, automation from machine learning has the potential to force millions out of work as a single machine is more than capable of doing the jobs of 100s. For example self driving cars have the potential to put 10s of 1000s of truck and other transportation drivers out of work as the field rapidly automates [5].

This project delves into the field of machine learning, with the main aim being the development of an artificial intelligence capable of playing the Game Of Less Risk , described in the background section of this paper. This is accomplished through the application of a variety of methods and technologies which are described in this paper. From Q-Learning to neural networks, a wide variety of techniques are employed and explored in order to further the author's understanding of the machine learning field. A rather unique aspect of machine learning is the necessity of training the programs developed, a concept which will be explored at depth as a large part of this project.

This paper aims to give the reader a comprehensive background in the field of machine learning, with a particular focus on neural networks which are employed as the primary machine learning component of the artificial intelligence. Further to this, concepts such as 'Q-Learning' and 'Genetic Algorithms' are also explored and explained in detail in order to help improve the reader's understanding. The paper covers the technologies utilised, the methods and techniques used and the concepts discussed, as well as how the task was approached and the various successes and failure in the development of the artificial intelligence.

1.2 Project Aims

As briefly mentioned in the project introduction, the main aim of this project was the development of an artificial intelligence capable of playing the Game Of Less Risk which, as mentioned, is described in the background section of this paper. This was accomplished through the use of a variety of different learning techniques and concepts all detailed in the background section alongside all technologies used described in the technology background section.

The project also has the aim of serving as an introduction to the world of machine learning. The very concept of a computer being able to improve its own capability of completing a task is a fascinating one which has a wide variety of applications in improving both mankind's technological limits and quality of life. Examples of this include AI assistants such as Microsoft's Cortana [7] and artificially intelligent doctors such as IBM's Watson [6]. Machine learning has the potential to improve humanity's quality of life past anything seen before and is something the author has a particular interesting in seeing the future developments of.

All machine learning solutions developed as part of this project are developed by the author with no external machine learning based libraries used. This is in order to achieve a better understanding of the techniques used in the creation of the artificial intelligence and to be better able to explain the concepts used.

Summarised, the aims of this project are as follows -

- Develop an artificial intelligence. Also referred to as a 'bot' or 'agent', capable of playing the Game Of Less Risk through the use of machine learning techniques. The main concepts incorporated being neural networks and q-learning.
- Develop implementations of all machine learning concepts used, with a particular focus on a black box type of development for for future integration with a wide range of projects.
- Develop the Game Of Less Risk using Unity, described in the technology background section, to provide an interactive environment for the artificial intelligence.
- Identify suitable methods for training the agent to effectively play the Game Of Less Risk .
- Provide a background of the field of machine learning, exploring the various concepts used to allow the reader to gain a deeper understanding into what machine learning actually is. As well as it can be applied to solve common problems thrugh research and implementation of machine learning algorithms.

2 Related Work

All of our knowledge comes from the past, and so this section will explore papers and other works relating to the fields of both machine learning and playing games. A recent piece of word is titled "Playing Video Games Using Neural Networks and Reinforcement Learning" [?]. The paper examines a variety of different attempts to train a neural network to play retro games such as Space Invaders. The general idea behind this approach was to use the game's screen data as input to be fed into a neural network and then train the neural network using the game's score as a reward. This approach of training through the use of rewards is called reinforcement learning. Unfortunately the project in the referenced paper was unfinished due to time constraints and issues with training, resulting in an AI that simply made moves at random. This is useful to take into consideration during the development of this project to prevent making the same mistake, meaning that time will need to be dedicated to the training of any artificial intelligences developed as a result of this project. The paper's author also commented on a lack of overall understanding of just what was being implemented "I still found myself needing to brush up on vital information throughout.” [?]. This could be taken as a warning to not begin implementation before the underlying techniques are well understood and should be considered during this project’s development. Another issue with the project was the poor performance of the neural network, as it ran slowly making it impractical for use in a real time game in which time is of the utmost importance. Slow performance should not be an issue for this project if it occurs due to the turn based nature of the game being implemented. Finally, other advice given was to use a third party library as opposed to attempting to develop a custom implementation. This is advice which shall be disregarded for this project as one of the underlying aims is further understanding through implementation, and so this project will still involve active implementation of all machine learning techniques used.

2.1 TD Gammon

TD-Gammon is one of the earlier examples of an artificial intelligence capable of playing a game. TD-Gammon as described in the paper ”Temporal Difference Learning and TD-Gammon” [20], which details the concept of using a neural network to play the game of backgammon. TD-Gammon is a ”neural network that trains itself to be an evaluation function for the game of backgammon by playing against itself and learnign from the outcome.” [20]. The use of a neural network is of particular interest due to its relation to this project’s goal of using machine learning techniques, including neural networks, to play games.

Although TD-Gammon was extremely successful and outclassed previous attempts at programs developed to play backgammon, this was not the reason for its development. Rather, ”its purpose was to explore some exciting new ideas and approaches to traditional problems in the field of reinforcement learning.” [20]. A purpose that is invaluable due to its similarity to the main purpose of this project; the research, implementation, and experimentation with machine learning applied to playing common games. The methods applied during development of TD-Gammon could also be applied to the development of the artificial intelligence intended to play the Game Of Less Risk . This is due to the similarities between the two, such as the fact they are both turn based games with multiple players. The difference in game-play between the two should be considered, but this could theoretically be resolved through the use of suitable training data.

TD-Gammon is an excellent example of success in the field of game playing AI, boasting a great deal of citations from other projects that involve the application of machine learning to the field of playing games. The paper truly shows the potential in this field. Due to this it should prove to be an excellent resource during the development of this project as both a point of reference and a point of inspiration. Notably, TD-Gammon makes use of the temporal difference algorithm as means of adjusting the weights of the neural network, something which could potentially be applied to this project due to the expected reward vs actual reward factor of the algorithm.

2.2 Deep Mind

DeepMind, a company recently acquired by Google, are the developers of one of the most recent works in the development of a neural network to play video games. First published in their paper "Playing Atari with Deep Reinforcement Learning" [10], their work has been a great source of motivation and inspiration in the development of this project. The paper details the development of an artificial intelligence capable of playing Atari games purely through the observation of image data in the form of the visible game screen. This is achieved through the inclusion of both neural networks and Q-learning. These are incorporated together so that the neural network is used to approximate the Q-Values used in the Q-Learning algorithm, this is called 'Deep Q learning'.

A remarkable feature of the AI described in the paper is its method of learning to play a game purely through the observation of the game screen's pixels and receiving feedback via the increase of the game's score. As a result of this relatively unique method the AI was capable of learning seven, "we apply our method to seven Atari 2600 games" [10], with no change to the AI's architecture. The AI actually exhibits features of a 'strong AI', also known as a 'general artificial intelligence', the definition of which is a machine that can perform any intellectual task that a human can. This is due to the AI's ability to play separate distinct games without being explicitly programmed or tailored for each game.

The AI developed by Deep Mind is of particular interest to this project due to the intelligence's use of a concept known as 'Deep Q Learning'. This is where Q learning and neural networks are used together to create an AI capable of learning policies that, when isolated, would be infeasible for either in isolation. Q learning works by iterating through the states of an environment until a reward is reached, and the "Q-Value" of the action leading to the reward is then set as the value of the reward. The "Q-Values" of every state-action pair leading to that action is then calculated. Once the "Q-Value" of each state has been calculated the optimum policy is to take the action with the highest "Q-Value" in each state. Q-Learning is effective but infeasible for environments in which there are a lot of states. For example, trying to apply Q-Learning to chess would be impossible due to the fact that chess has a lower bound of 10^{120} [11] states. With current computer technology it would be impossible to store and iterate through this many states, meaning a different approach is needed. The solution to this problem is the use of a neural network to calculate the Q value of each action for each state. This is known as 'Q-Value approximation' and is the core of deep Q-Learning.

2.3 Summary of Main Points

The idea of using a computer to play games is nothing new, with a well-established field that should be a valuable source of information during this project's development. Research into related work has brought up some interesting points that should be kept in mind during the development of this project. Neural networks are a common feature of game playing AIs and should serve as an excellent backbone to any work developed during this project. They are typically used for their approximation capabilities where it is infeasible to actively store states due to either the size or complexity of the possible states. As the game of risk has an astronomical amount of states it is likely that a neural network will be used for the same purpose in this project as well. Another common feature is the use of reinforcement learning, this is due to the same issue that makes the use of a neural network necessary, namely that the sheer number of states for each game makes supervised training impractical as it would be extremely difficult to have a training set of what action to take in each game state. To this end reinforcement learning is used to its ability to over time find out for itself the best action to take for each state without any need for observation or interference. Finally, something else that should be kept in mind is the need for proper training for an AI to be effective. The amount as well as the type of training should be taken into consideration, for example training a neural network using the exact same game repeatedly is just going to train the neural network for that specific pattern. A solution to prevent this could be the use of a random opponent to train against which will use different strategies each time so that the AI does not specialise itself to expect and carry out the exact same action every time.

3 Background

3.1 Markov Decision Process

A Markov Decision Process or 'MDP' is a mathematical framework for the modelling of decision making in an environment which is partially random. A single MDP is a discrete-time state transition system in which at a certain time-step **t** is in state **s** and can choose to execute any action **a** that is available in state **s**. The process then moves into a new state **s'** and provides a reward to the decision maker which decided to execute action **a** in state **s**.

Formally, a MDP has the following definition -

$$MDP = \langle S, A, T(s, a, s'), R(s), \gamma \rangle$$

- S - Finite set of states in the environment.
- A - Set of possible actions.
- T(s, a, s') - Transition model that determines how a state traverses from state s to state s' using action a.
- R(S) - Reward function for a given state s.
- γ - Discount factor for future rewards.

A MDP is useful for reinforcement learning where an agent must learn from its environment. Through modelling the environment as a maarkov decision process the agent can effectively traverse the environment and find an optimal policy through feeding back future rewards to earlier state action pairs. For more information regarding reinforcement learning refer to the reinforcement learning section later in this paper.

3.2 Game Theory

As the goal of the project is the application of machine learning to playing games, game theory is a topic that is worthy of a brief review. Defined as "the study of mathematical models of conflict and cooperation between intelligence rational decision-makers" [12], game theory is an area of research with roots in a large variety of fields. Examples of fields in which the concepts of game theory can be applied include economics, political science and of course, computer science. Game theory attempts to apply mathematical techniques for the analysis of situations in which multiple individuals will make decisions that affect the position of each other. This of course applies to the area of playing games, where a player can make a move that has the potential to increase the value of their position and/or simultaneously decrease the value of their opponent's position.

An area of particular interest is the concept of a 'solved' game. A solved game is a two player game where the outcome can be accurately predicted from any position with the assumption that both players are playing perfectly to maximise their score and minimise the opponents score. Another concept is a 'perfect play', defined as a strategy used by one player that ensures they achieve the best possible outcome no matter the response by the other player. Naturally an AI capable of perfect play is optimum goal to strive for in the development of this project. There are three types of solved game as follows -

- **Ultra-Weak** - "For the initial position(s), the game-theoretic value has been determined." [13]
- **Weak** - "For the initial position(s), a strategy has been determined to obtain at least the game-theoretic value of the game, for both players, under reasonable resources." [13]
- **Strong** - "For all legal positions, a strategy has been determined to obtain them game-theoretic value of the position, for both players, under reasonable resources." [13]

Although RISK can be played as a two player game, it cannot be solved due to the random elements involved. Making it impossible to accurately predict the outcome of certain moves. In particular it is impossible to determine the outcome of attacks from one region to another due to the random element of dice rolls to determine the outcome. On the other hand TicTacToe is a solved game where the game will always end in a draw if both players are playing perfectly. This is due to the small amount of possible moves and the lack of any random elements, making it a trivial task to work out the outcome of making a certain move in a particular state.

3.3 Perceptron

Also known as a 'single-layer perception', the perception is a learning algorithm used for binary classification. It works by taking in some number of weighted inputs, applying an activation function to the sum of these inputs and outputting a single value. Refer to Figure 1 for a diagram of a perceptron.

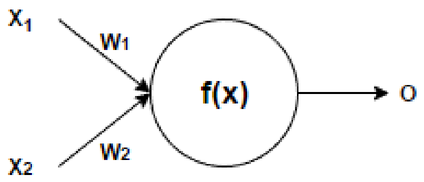


Figure 1: A Perceptron

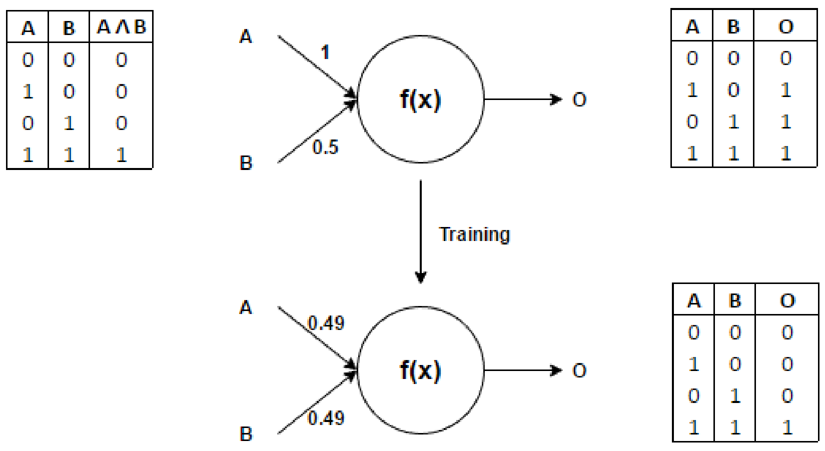
The perceptron is trained using supervised learning in order to map some input pattern to a desired output. This is done by adjusting the weights of each input in order to affect the output and improve the performance of the perceptron. A common example of a perceptron's use is with modelling simple logic gates, although a notable exception to this rule is the XOR gate which cannot be modelled with binary classification and required a neural network. For an example of a perceptron used to model an AND gate refer to Figure 2.

Formally, the value of the perception in Figure 1 is calculated using the following equation -

$$o = f(x_1 * w_1 + x_2 * w_2)$$

3.4 Feed-forward Neural Network

A neural network is made of up multiple layers of nodes referred to as neurons which are functionally identical to perceptrons, where the output of a perceptron in one layer is fed as input to every perceptron in the next layer. Each layer of neurons has weighted connects to every neuron in the next layer, there are three types of layers in a typical fully connected feed forward neural network. The types of layers being the input layer, hidden layer and the output layer.



A	B	A∧B
0	0	0
1	0	0
0	1	0
1	1	1

A	B	O
0	0	0
1	0	1
0	1	1
1	1	1

A	B	O
0	0	0
1	0	0
0	1	0
1	1	1

$$f(x) = \begin{cases} 0, x < 0.5 \\ 1, x \geq 0.5 \end{cases}$$

Figure 2: Perceptron modelling an AND gate.

Every neural network has a single input and output layer as well as an arbitrary number of hidden layers depending on the complexity and function of the network. A greater number of hidden layers results in a more complex and capable network, although too many layers often leads to redundancy and can cause more problems than it solves. A neural network with more than a single hidden layer is referred to as a 'deep neural network'. Another feature of a neural network is the addition of a bias neuron to each layer excluding the input layer. This neuron is similar to every other in that it has a weighted connection but it receives no input from other neurons and holds a constant value, typically '1'. The use of the bias neuron is to shift the results of the network's activation function to the left or right, depending on the desired results. An example of the structure of a neural network is available in Figure 3. The bias neuron is typically omitted from neural network diagrams for readability, but would be displayed as a single neuron connected to every non-input neuron.

Input is fed into a network via the input layer, where the value of each input neuron is set to the desired input. This input is then fed along weighted connections to every neuron in the next layer. The value of each neuron in the next layer is then calculated as the sum of every input multiplied by the corresponding weight, this value is then fed through the neuron's activation function to calculate the final value of the neuron. There are many different types of activation functions that can be used depending on the intended purpose of the neural network, the most

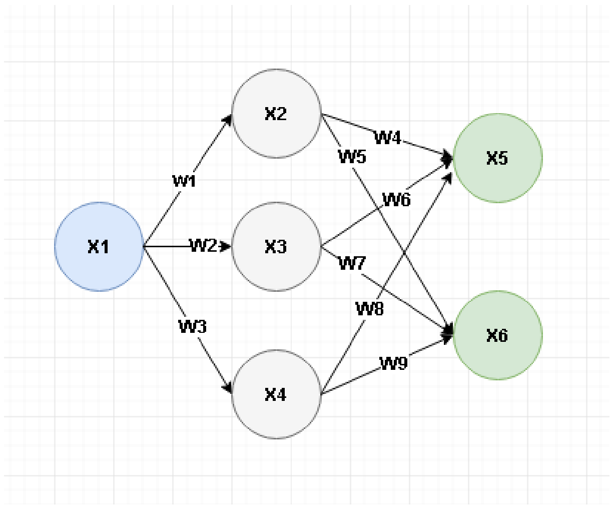


Figure 3: A neural network composed of 1 input neuron, a single hidden layer of 3 neurons and 2 output neurons.

activation function used is known as the sigmoid function, as shown in Figure 4.

The sigmoid function is defined as follows -

$$y = \frac{1}{1 + e^{-x}}$$

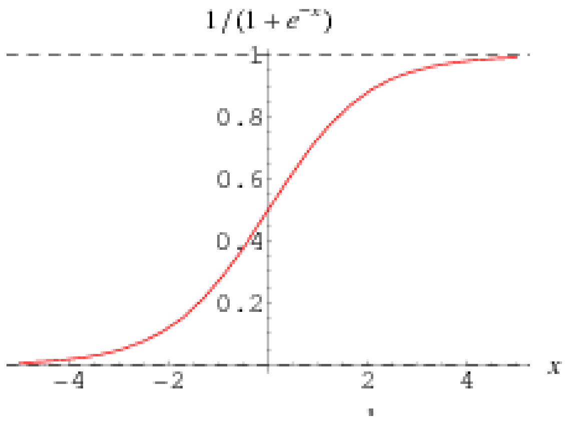


Figure 4: Mapping of the sigmoid function. [14]

A neural network is formally defined as follows -

$$x_n = \sum_i w_{ni} x_i + w_b b$$

- x_n is the value of the neuron being calculated.
- w_{ni} represents the weight of a connection from neuron i to neuron n.
- x_i is the value of the neuron i.
- $w_b b$ is the product of the bias neuron's value and the weight of the bias neuron's connection.

At it's core, a neural network is essentially a function that maps some input to some output. Similar to how a perceptron can be used to model something simple like a logic gate, a fully connected neural network can be used to model functions that are a lot more complex. This works through training a neural network on some training data and adjusting its weight's so that over time the output of the network more closely matches the desired output. For example a neural network could be used to identify pictures of birds by feeding it input in the form of some prepared image data and having it output a single value corresponding to true or false depending on if the network recognises the image as a bird or not. Initially the network will output purely random results but the network can be trained to over time adjust it's weights and learn to recognise images of birds. Depending on the type of training used the method of doing this can vary, but typically for image recognition tasks supervised learning is used where the network would be given a training set consisting of labelled bird and non-bird images. So image data is fed as input and the network trained to adjust its output to be closer to the input's label. The network's capability to learn complex functions that would be impractical or difficult to implicitly program is extremely useful, allowing for complex tasks to be learned and carried out by networks that are relatively simple to train and develop. Supervised, unsupervised and reinforcement learning are all training methodologies that can be used to train a neural network, further discussed in the following sections. The adjustment of the weights of a neural network is typically done using the backpropagation algorithm, detailed in the next section.

3.5 Backpropagation

The backpropagation algorithm is a function commonly used for the training of neural networks. This is achieved by calculating the overall error of the network and propagating this back through the layers, calculating the error of each neuron and adjusting its weights accordingly. For back-propagation to be used the desired output of the network must be known so that the error can be calculated properly. In the case of supervised learning this is simple as the user will provide the network with the desired input, output pairs. But in the case of reinforcement learning additional algorithms such as Q-learning must be used to obtain a desired output for use in error calculation.

Summarised, the main purpose of the algorithm is to simply try and reduce the overall error of the network through reducing the error of each individual neuron by adjusting each neuron's weights. The end result of this being an output which is closer to the goal than it was before the algorithm was applied.

The first step of the algorithm is to calculate the overall error of the network, which is done by calculating the error of each output neuron, calculated as follows [15] -

$$E_{o1} = \frac{1}{2}(target_{o1} - output_{o1})^2$$

- $o1$ is a single output neuron in the network.
- $target$ is the target output for the output neuron.
- $output$ is the actual output for the output neuron.

For example, if an output neuron $o1$ has an expected value of 3 but an actual value of 2. The error is calculated as follows -

$$E_{o1} = \frac{1}{2}(3 - 2)^2 = 0.5$$

Once the error of every output neuron has been calculated, the overall error is calculated as follows -

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

- E_{total} is the total error of the entire network.

- *target* is the target output for each output neuron.
- *output* is the actual output for each output neuron.

For example, given a network with two output neurons $o1$ and $o2$ that have actual values of 2.5 and 5 with expected outputs of 3 and 4 respectively, the total error is calculated as -

$$E_{total} = \frac{1}{2}(3 - 2.5)^2 + \frac{1}{2}(4 - 5)^2 = 0.125 + 0.5 = 0.625$$

Using this error, the algorithm then backtracks through the neural network starting at the output layer and for each neuron calculates its error and adjusts all associated weights in order to minimise the error and cause the overall output to be closer to the desired output.

Starting in the output layer, the algorithm calculates how a change in a weight w_{ij}, which connects a hidden neuron j to the an output neuron i affects the total error. Done using the following -

$$\frac{\partial E_{total}}{\partial w_{ij}}$$

The chain rule is applied to this derivative to get the following -

$$\frac{\partial E_{total}}{\partial w_{ij}} = \frac{\partial E_{total}}{\partial out_{oi}} * \frac{\partial out_{oi}}{\partial net_{oi}} * \frac{\partial net_{oi}}{\partial w_{ij}}$$

Calculate how the total error changes with respect to the total output -

$$\frac{\partial E_{total}}{\partial out_{oi}} = -(target_{oi} - out_{oi})$$

Calculate how the output of the neuron oi changes with respect to its total input -

$$\frac{\partial out_{oi}}{\partial net_{oi}} = out_{oi}(1 - out_{oi})$$

Finally, calculate how the total net input of oi changes with respect to w_{ij}.

$$\frac{\partial out_{oi}}{\partial w_{ij}} = out_{oj}$$

The results of these calculations are they multiplied together using the following equation obtained earlier to get the final value.

$$\frac{\partial E_{total}}{\partial w_{ij}} = \frac{\partial E_{total}}{\partial out_{oi}} * \frac{\partial out_{oi}}{\partial net_{oi}} * \frac{\partial net_{oi}}{\partial w_{ij}}$$

This resulting value is then subtracted from the weight w_{ij} to decrease the error of the neuron oi. This process is repeated with the weights of each output neuron.

The algorithm then backtracks to the previous hidden layer where the process is slightly different to account for the fact that the output of each hidden neuron contributes to the output of multiple other outputs in the next layer. For a single hidden neuron hi that is connected to output neurons $o1$ and $o2$ the following is calculated -

$$\frac{\partial E_{total}}{\partial out_{hi}} = \frac{\partial E_{o1}}{\partial out_{hi}} + \frac{\partial E_{o2}}{\partial out_{hi}}$$

$$\frac{\partial E_{o1}}{\partial out_{hi}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{hi}}$$

$$\frac{\partial E_{o2}}{\partial out_{hi}} = \frac{\partial E_{o2}}{\partial net_{o2}} * \frac{\partial net_{o2}}{\partial out_{hi}}$$

For each weight the following is calculated -

$$out_{hi} = \frac{1}{1 + e^{-net_{hi}}}$$

$$\frac{\partial out_{hi}}{\partial net_{hi}} = out_{hi}(1 - out_{hi})$$

Calculate the partial derivative of the net input to hi with respect to w_{ij} -

$$\frac{\partial net_{hi}}{\partial w_{ij}} = i_i$$

The final value is then calculated using the following calculation -

$$\frac{\partial E_{total}}{\partial w_{ij}} = \frac{\partial E_{total}}{\partial out_{hi}} * \frac{\partial out_{hi}}{\partial net_{hi}} * \frac{\partial net_{hi}}{\partial w_{ij}}$$

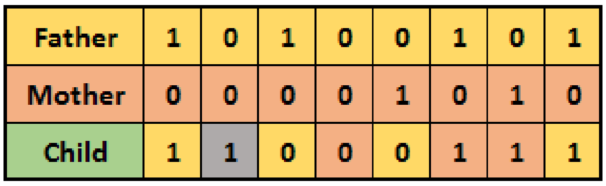
This resulting value is then subtracted from the weight w_{ij} to decrease the error of the neuron hi. This process is repeated with the weights of each hidden neuron.

Although the backpropagation algorithm looks overwhelming at first, it is actually relatively simple to implement as documented in the implementation section of it this paper. It just requires a lot of research to get to grips with and understand the underlying concepts involved.

Informally, the algorithm starts in the output layer and calculates the error of each output neuron based on the actual vs expected value. These errors are then used to calculate the error that is contributed by every hidden neuron in the network. The weights connected to each neuron are then adjusted based on the error of the neuron the weight is associated with in an effort to reduce the overall error of the network. It takes multiple iterations of the algorithm in order to fully minimise the networks error. It is of vital importance when using backpropagation to run it for each possible input/output combination once every training cycle, as opposed to training for one input/output combination 100 times and then the next combination another 100 times. This is to prevent the network 'forgetting' the adjustments made by the backpropagation algorithm for the previous combination and ensures the average error for the entire training set is reduced.

3.6 Genetic Algorithm

Inspired by the process of natural selection, the genetic algorithm that applies the evolutionary concepts of mutation, crossover and selection in an attempt to solve optimisation problems. The basic idea behind a genetic algorithm is to generate an initial population of individuals, each of which contain useful information in the form of 'genes'. Each individual's fitness is then calculated using some fitness function and the best individuals are selected then bred with each other to generate an entirely new population that should on average be fitter than the last. This process continues until an individual with the most ideal genes has been generated that is fit for purpose. This process is similar to the concept of survival of the fittest in nature, where the animals best adapted to their environment get to breed and the others eventually die off. A visual representation of the breeding process is displayed in Figure 5



Father	1	0	1	0	0	1	0	1
Mother	0	0	0	0	1	0	1	0
Child	1	1	0	0	0	1	1	1

Figure 5: Breeding process between two individuals as part of the genetic algorithm, blue represents mutation.

When individuals are bred with each other the concepts of mutation and crossover and both applied. Crossover is where the the genes of the new individual generated from the 'mother' and the 'father' are a random combination of the parent's genes. The other concept is mutation which is important to add some random exploration into the mix. During crossover when genes are selected from the mother or father, there is also a random chance at mutation where an entirely new value for the gene is generated, instead of using one of the parent's genes.

The genes are the useful information generated by the genetic algorithm and can be used as required. One example of their use is setting the weights of a neural network. The fitness of the individual would then be the calculated as the performance of the neural network and in theory over time the genetic algorithm should generate an individual that has the ideal weights for the neural network. This is an alternative method of adjusting the network's weights over the typical backpropagation algorithm and has the benefit of also training the network at the same time.

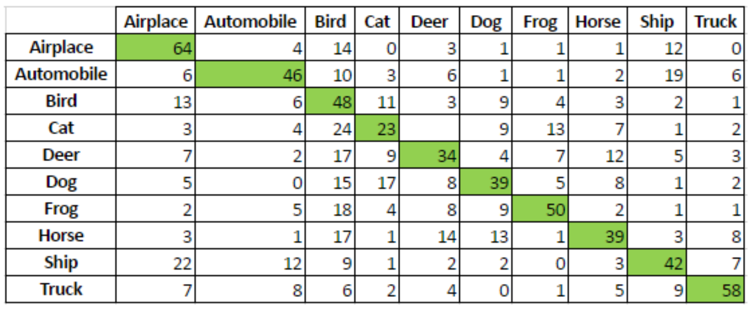
An interesting implementation of the genetic algorithm is in the generation of an evolved antenna. An evolved antenna is an antenna designed by a genetic algorithm, starting with simple shapes and modifying elements in a semi-random manner. Many models are generated and their fitnesses calculated. The worst performers are discarded while the others are used to generate a new population to be judged. Eventually the highest scoring design is chosen once certain criteria are met, with the resulting antenna typically outperforming manual designs due to the generation of random complicated shapes that which would not have been found using traditional methods. Although unrelated to this project it still shows the potential that genetic algorithms have if properly applied. An example of an evolved antenna is the ST5 antenna [16] created for use in a 2006 NASA mission which launched launched in March, 2006. This represented the first artificially evolved object to fly in space.

3.7 Supervised Learning

Supervised learning is a learning technique where the learner is trained on a labelled training set ”The learner receives a set of labeled[sic] examples as training data” [17]. The learner is fed each element of the training set as input and the labels are used as targets to train the learner towards. A testing set which is separate from the training set is then typically used to gauge the performance of the learner after training is complete.

An example of this is training a neural network to recognise images of tanks. A training set containing different images each labelled as to if they are tanks or not is used to train the neural network by feeding in each image as input and adjusting the network's weights to adjust the output depending on the label's value. A separate testing set containing both tank and non-tank images would then be used to gauge the trained network's performance by calculating the percentage of correct predictions. For classification problems a confusion matrix is typically used to gauge a learner's performance, an example of which is in Figure 6.

Supervised learning is unsuitable for use in training an agent to play more complex games such as RISK or Chess. This is due to not knowing what the target output should be for each state beforehand as the goal of the agent is to work it out for itself. Although supervised learning could be used for solved games such as TicTacToe, as it is possible to calculate a perfect move for each state which could be used to train a neural network.



	Airplace	Automobile	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck
Airplace	64	4	14	0	3	1	1	1	12	0
Automobile	6	46	10	3	6	1	1	2	19	6
Bird	13	6	48	11	3	9	4	3	2	1
Cat	3	4	24	23		9	13	7	1	2
Deer	7	2	17	9	34	4	7	12	5	3
Dog	5	0	15	17	8	39	5	8	1	2
Frog	2	5	18	4	8	9	50	2	1	1
Horse	3	1	17	1	14	13	1	39	3	8
Ship	22	12	9	1	2	2	0	3	42	7
Truck	7	8	6	2	4	0	1	5	9	58

Figure 6: Ten category classification confusion matrix.

3.8 Reinforcement Learning

Reinforcement learning is an aspect of machine learning in which the agent learns through the acquirement of perceived rewards with the end goal of maximising its total accumulated reward. This area of research is inspired by the psychology of animal behaviour, the idea of a dog being trained using treats being applied to a machine to train it to better fulfil a certain task through being given rewards. Unlike supervised learning the agent is never provided the correct answer in the form of input/output pairs and is instead usually given a large positive reward in the case of correct performance and a large negative reward in the case of incorrect performance. These rewards are then usually propagated back on a decaying scale through the action state pairs that led the agent to the state in which it received the reward so it has an idea of future reward it will receive performing a certain action in a certain state.

3.8.1 Credit Assignment Problem

Assume you have a maze with an agent that can explore the maze at random by doing one of four actions 'up', 'down', 'left', 'right'. The agent proceeds to move randomly through the maze and eventually exits the maze, receiving a reward for doing so.

Logically the state-action pair that let to the agent immediately exiting the maze is responsible for this action and so credit is assigned to that action in that state. The issue comes from assigning credit to each state-action pair preceding the final state-action. The agent needs to know how to get into the state in which it can exit the maze and so it needs to know which action to take in each state in order to maximise its final reward. But to do this a value needs to be given to each action in each state so the agent can pick the action with the highest value. A simple solution would be to record the path the agent originally took to exit the maze and give each action a positive value but it is not guaranteed that the agent took the optimal path and it is not known as to which actions helped the agent and which impacted it. So the problem is working out which of the preceding actions were responsible for improving the performance of the agent and allowing it to get a reward.

Multiple solutions exist to solve this problem. One in particular which is looked at in the following section is known as Q-Learning, where the agent would traverse the maze multiple times and the values of the actions taken are refined repeatedly until the optimum value of each action is each state is found. The agent would then simply take the value with the highest value in each state to exit the maze.

3.8.2 Q-Learning

The Q-Learning algorithm is a reinforcement learning technique which is used to calculate the optimal action for each state in an environment based on some perceived future reward. Introduced in 1989 as a means of learning from delayed rewards [18]. Q-Learning works by "incrementally updating the expected values of actions in states. For every possible state, every possible action is assigned a value which is a function of both the immediate reward for taking that action and the expected reward in the future based on the new state that is the result of taking that action." [19]. In other words the algorithm traverses through the environment and attempts to find the "Q-Value" of every possible action for each possible state.

So for an action 'u' in state 'x' the Q-value is calculated using the following function [19] unless the resulting state is a terminal state, in which case the Q-Value is equal to the reward of executing action 'u'.

$$Q(x,u) := (1-\alpha)Q(x,u) + \alpha(R + \gamma\, max[Q(x_{t+1}, u_{t+1})])$$

- Q is the expected value of performing action u in state x.
- x is the current state.
- u is an action from state x.
- R is the reward obtained for performing action u.

- α is the learning rate which controls convergence.
- γ is the discount factor, which makes earlier rewards more valuable than later rewards.
- $Q(x, u)$ is the current Q value of action u from state x.
- $max[Q(x_{t+1}, u_{t+1})]$ is the maximum Q value of all possible actions from the state reached by action u.

Of particular interest is the element of the function which takes into account future rewards ' $max[Q(x_{t+1}, u_{t+1})]$'. This simply looks at the Q value of every action that can be executed in the state that would result of doing action 'x' in state 'u' and gets the highest Q-Value which is then used to calculate the current Q-Value. The benefit of this is that it allows for future rewards to be propagated back through actions that may have led to that reward but themselves received no reward. Once the maximum Q-Value of the next state is found it is multiplied by the discount factor which determines how to value future vs later rewards. The current Q-Value is also used in the calculation of the new Q-Value so that over time the current Q-Value moves towards its optimum value.

Once Q-Values have been calculated the optimum policy is simply the action with the highest Q-Value. Although an issue with this is that once the Q-Value of one action for a state has been calculated it will always execute this action as it has the highest Q-Value and will never explore other possibilities. To resolve this the algorithm follows an "epsilon-greedy" policy where each time it picks an action it has an epsilon chance to instead do a random action and learn from that instead. This is called exploration vs exploitation where the environment is explored and exploitation delayed so that every possibility can be explored. The value of epsilon is typically decreased as the algorithm iterates over the environment's states so that the algorithm states to exploit its knowledge at a higher rate than exploring at random. Another issue this approach resolves is a situation where the agent continuously exploits a smaller reward when a much larger reward lies a few more steps away "it is necessary to sometimes pick a random action to not get stuck in local reward maxima" [10].

Q-Learning is of particular interest due to its value in calculating an optimal policy for exploiting and environment through the use of future rewards. This is due to the nature of games where a reward is typically not received until a game is over, so the use of Q-Learning would allow an AI to be trained to play a game and receive adequate feedback for both early actions and those which lead into a terminal game state.

3.8.3 Temporal Difference Learning

Also referred to as TD(λ), the temporal difference algorithm is a learning algorithm which attempts to improve future predictions based on the difference between temporally successive predictions. Or, "In other words, the goal of learning is to make the learner's current prediction for the current input pattern more closely match the next prediction at the next time step." [20]. An example of its algorithms use could be weather prediction "A TS approach, on the other hand, is to compare each day's prediction with that made on the following day. If a 50% change of rain is predicted on Monday, and a 75% change on Tuesday, then a TD method increases predictions for days similar to Monday, whereas a conventional method might either increase of decrease them depending on Saturday's actual outcome." [21].

The algorithm is used for the calculation of perceived reward based upon past results and is as follows -

$$\tilde{V_t} = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- $\tilde{V_t}$ is the correct prediction that is equal to the discounted sum of all future reward.
- r_t is the reward on time step t.
- Discounting done by powers of a factor of γ.

This algorithm could be applied to the area of playing games by using it to predict the score that would result from making a certain move. In fact, this algorithm has already been used to create an AI in the paper "A Learning AI for the game Risk using the TD(λ)-Algorithm" [22] capable of playing the game of RISK, a major aim of this project.

3.9 Game of Risk

3.9.1 Overview

The game of Risk is a strategic turn based board game, the aim of which is to eliminate all other players on the board. The game is played on a map split into multiple continents, each of which is further split into territories, an example of which can be seen in Figure 7. A player's goal during the game is to conquer and occupy every territory on the board, hereby eliminating all other players and winning the game. This is achieved through the use of armies spread across the board which the player can use to move to friendly territories or attack hostile ones. Each continent gives bonuses to the player who controls all incorporated territories, making control of these continents a crucial part of any strategy.

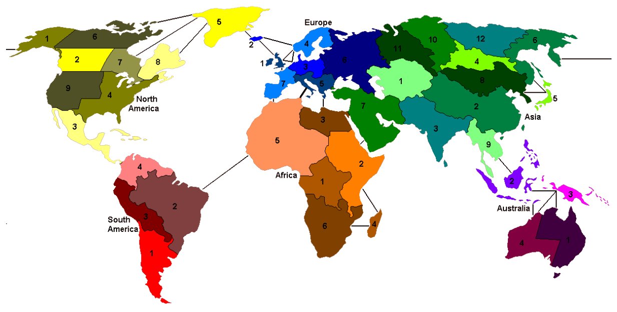


Figure 7: Typical Risk game board. [23]

The game of Risk is currently published by Hasbro and all rules as explained here originate from the official manual. [24]

3.9.2 Placement Phase

The game begins with the placement phase, where each player is given a set number of armies depending on the number of players present. Each player then takes turns to deploy a single army unit in any unoccupied territories until all territories have been claimed, the remaining army units are then placed on any friendly territory until all army units have been placed. A die roll is typically used to determine who goes first in this process, a second die roll is then used to determine who goes first when the actual game turn loop begins. This phase replaces the reinforcement phase on the first turn.

3.9.3 Reinforcement Phase

If the game has started and the placement phase has been concluded, the turn starts with the reinforcement phase. In this phase the player receives a number of new army units depending on the number and nature of territories owned. The player then deploys these new units to any friendly territory in any fashion they desire. The number of units received during this turn is calculated based on the following conditions -

- One unit is obtained for every three territories under player control. For example, if 9 territories are under player control then the player receives 3 army units.
- Each continent also has a bonus associated with it depending on its size and difficulty to conquer. For example on the typical RISK board controlling Africa will give the player 3 extra army units, while controlling Asia will give them 7.
- The player receives a minimum of three army units during this stage if they do not exceed this amount through other means.

3.9.4 Expansion Phase

The second and final phase of each turn is the expansion phase, where the player is given the opportunity to reinforce friendly territories and attack adjacent hostile ones. For each region the player owns they may choose to attack hostile territories adjacent to that territory. There is no limit to the number of regions a player may use to attack, assuming they have the resources to do so. The player may also make the choice to not attack. When a player launches an attack they must decide if that are going to use 1, 2 or 3 army units in the attack. The defending territory must defend with up to two units. A number of dice corresponding to the number of attackers and defenders is then rolls. The dice of the opposing sides are then matched up highest to lowest and for each comparison the player with the lower dice roll loses a unit. In the case of an unequal number of dice the lower values are discarded, as a result the attacker can never lose more than 2 units in an attack.

If the result of an attack reduces the number of units in a defending territory to 0, then the attacker gains control and occupies that territory with the attacking forces.

The player may also choose to move units from any region to an adjacent friendly region to reinforce that region, with no battle taking place. The act of moving armies is known as an "AttackMove" and are typically queued up and executed as a batch, as opposed to executing one action, giving an order, doing that action, etc.

4 Technology Background

4.1 Unity Engine

A powerful, free to use tool. Unity is a free game engine and editor developed by Unity Technologies [25] used in both independent and professional game development for a variety of platforms. The editor combines a simple component based drag and drop interface with either C# or Java scripting to allow for the rapid development of game prototypes, while also providing higher level control for more ambitious and larger game projects. A image of the editor is provided in Figure 8.

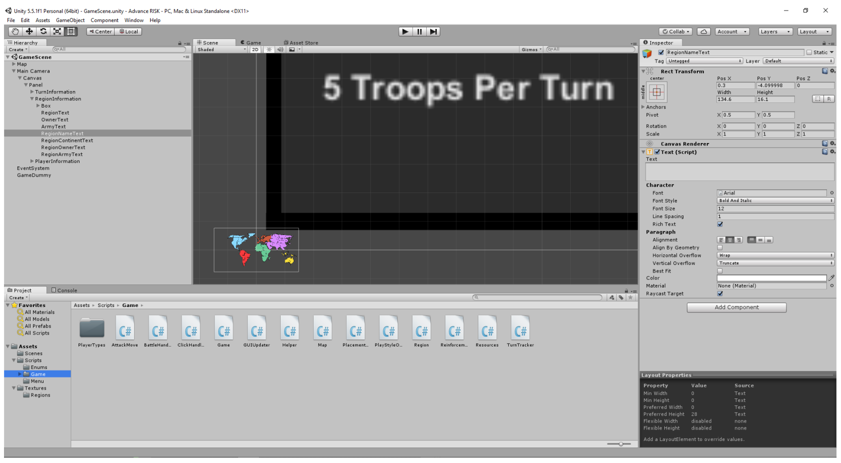


Figure 8: Unity Editor

Unity is used in the development of games to be integrated with the machine learning components of this project. It allows for the rapid development of interactive environments for the machine learning agents to interact with and learn from. The use of unity also enables a more interesting graphical interface without having to do something extreme like develop a custom game engine or disengaging like using a console interface. A console interface for this task would also be a bit overwhelming due to the amount of information involved, and as the saying goes a 'picture is worth 1000 words'. Visual Studio is used in the writing of C# scripts which compose the game logic and is described in more depth in an upcoming sub-section.

4.2 Visual Studio

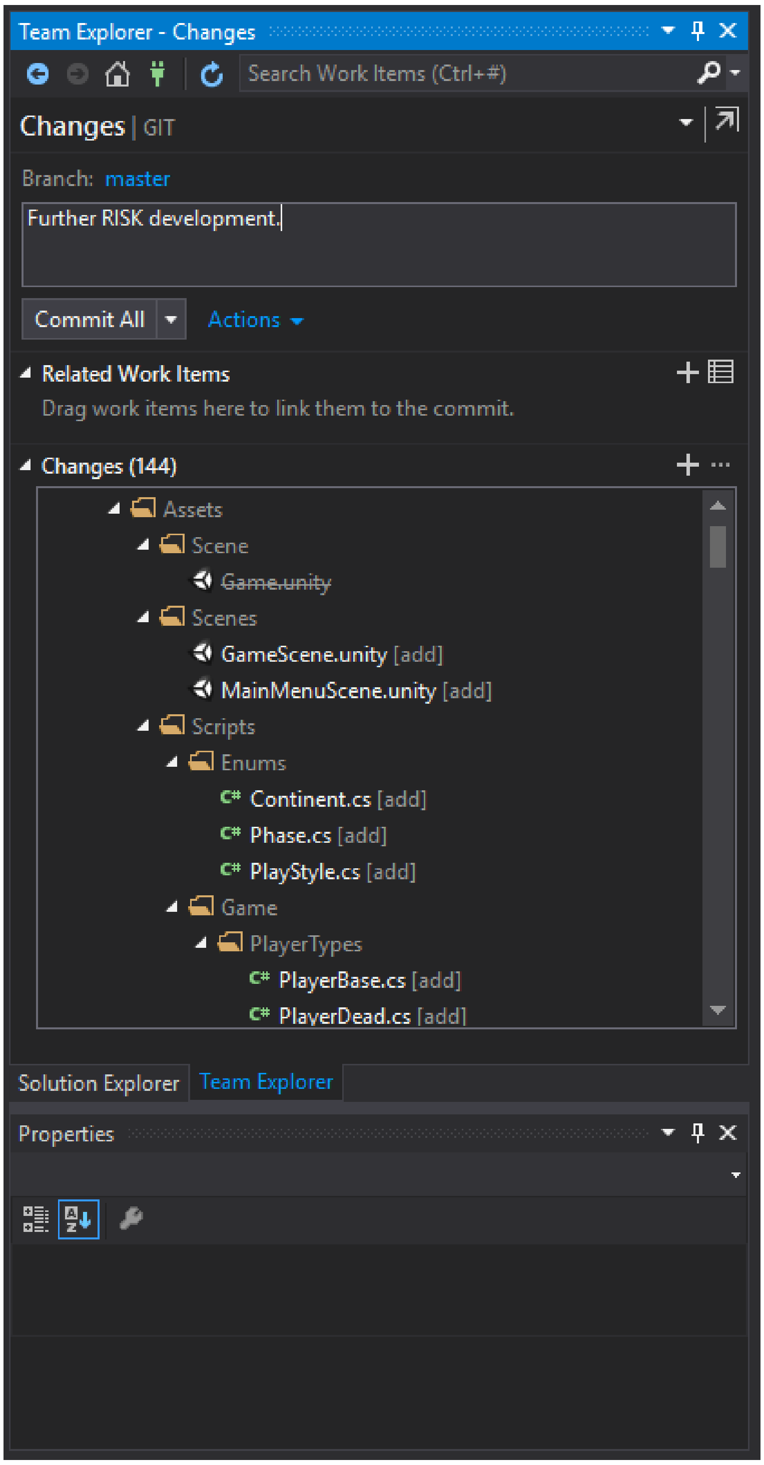
Visual Studio [26] is an integrated development environment developed and maintained by Microsoft. Although its primary use is for the development of projects for use on the windows operating system, it is flexible enough to develop projects for a wide range of different platforms. The IDE was used in conjunction with the .NET framework to develop all software associated with this project, with everything being written in the C# programming language.

4.3 Git Source Control

Git is a free, open source version control system and was used for both version control and code backup during the development of this project. 'Visual studio team services' was used as the git repository as it included integration with visual studio which allowed for seamless use during development. This system works by having a local repository stored on a personal computer where a local copy of the on-line repository is stored and can be changed freely. Any changes made to these local files can then be checked in and synced with the on-line repository and checked out elsewhere. In the event that old code needs to be retrieved Git provides a file history system in which the history of a specific file can be viewed and reverted to as needed. In the event of catastrophic file loss the on-line repository can simply be re-cloned to prevent any events that may have negatively affected this project's development. An example of Git's integration with visual studio can be seen in Figure 9.

4.4 .NET Framework

The .NET Framework is a powerful software framework developed by Microsoft, primarily used in projects developed for the Windows operating system. Its primary advantage is the inclusion of extensive class libraries which contain large amounts of reusable code for a wide range of applications. This allows a developer to quickly write code while using these libraries to provide common functionality such as interfacing with the console or generating random numbers which are usually found to a large number of programs. This means that the developer can spend their time writing code specific to their project instead of having to reinvent the wheel every time they want to write a new program.



```
Further RISK development.
```

```
Assets
  Scene
    ~~Game.unity~~
  Scenes
    GameScene.unity [add]
    MainMenuScene.unity [add]
  Scripts
    Enums
      Continent.cs [add]
      Phase.cs [add]
      PlayStyle.cs [add]
    Game
      PlayerTypes
        PlayerBase.cs [add]
        PlayerDead.cs [add]
```

Figure 9: Visual Studio Git Integration

5 Project Management

5.1 Version Control

GIT was used as version control during the development of this project. As detailed in the technology background section files are held in an on-line repository and cloned to a local machine when needed. Local files can then be freely edited and uploaded to the on-line repository to be stored and used elsewhere. The advantages of this are that it allows for an off-site backup which allows for recovery in the event of catastrophic data loss. It also acts as a version control system allowing files to be reverted if older code needs to be recovered. The process for editing a file is to check it out, make the desired changes and then check it back in and sync with the on-line repository so that everything is kept up to date.

5.2 Development Life-cycle

A first choice of development model to use during this project's development would be the waterfall model. Despite the drawbacks that come with using this method it would be ideal for this project due to the concurrent development of multiple pieces of software that will need to interact with each other. The software being the machine learning and game implementations which will need to work together. If waterfall model was used it would simplify development as it would allow for the requirements and structure of the two systems to be set in stone, including as to how the systems would interact with each other.

Despite this, the agile development model was used during the project's development. This is due to the fact that machine learning involves a learning process which cannot reliably be predicted before implementation has been carried out. This is due to the fact that effective learning requires constant tweaking and changes made to the implementation as potential avenues of learning are attempted and discarded. Because of this factor it is infeasible to see the finalised design of the system before complementation begins as it will inevitably result in constant design changes and revisions when it comes to the learning process.

Evolutionary development, early delivery and continuous improvements are the basis of the agile development model. This proved to be a suitable model for this project as it allowed for constant refinements when it came to the final parts of the implementation stage, the learning phase. Due to the constant refinements it allowed for a more robust and effective manner of teaching the AI as it allowed for experimentation into a variety of types of implementation to figure out the most effective method of teaching.

The agile development model is broken down into the following steps -

- **Define** - The system is designed and requirements are identified.
- **Develop** - The system is developed according to the design and requirements identified in the 'define' stage.
- **Release** - Although this project does not involve a customer release, this stage applies as it also involves testing of the system.
- **Evaluate** - The system is evaluated against its original design and potential improvements are identified and incorporated into a new designed.

These steps are then repeated as the development of the system evolves until a finalised version is produced.

5.3 Project Timeline

The time-line for this project deviated greatly from the original time-line devised as in Figure 10. This was mainly due to one previously unknown simple fact, that machine learning is not a simple cut and dry process.

Namely, the training part of the machine learning is not a cut and dry process. You cannot simply implement a machine learning algorithm such as a neural network, throw a problem at it and expect it to solve it with only basic training. In fact the training process is actually one of the most challenging aspects and requires constant tweaking and development in order to work correctly. Because of this the original predicted time-line way off compared to how the project's development actually went. In the original time-line neat little sections of time were cut out for training, testing and tweaking while in reality the development was done pretty rapidly and a much larger amount of time spent constantly tweaking and retraining the artificial intelligence. In fact a more accurate time-line would be a constant block of time from the start of February to the present spent on the development and training sections.

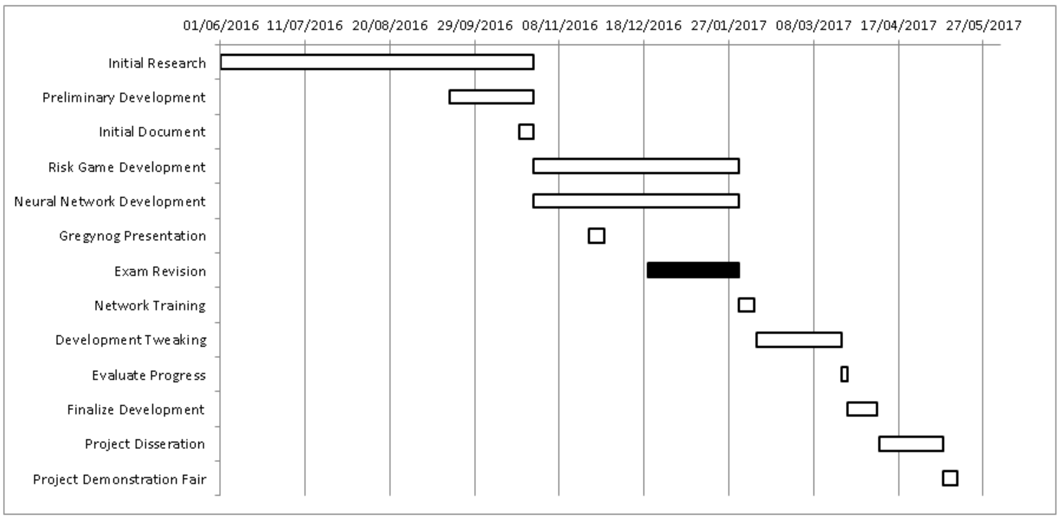


Figure 10: Original time-line for the project.

5.4 Project Risks

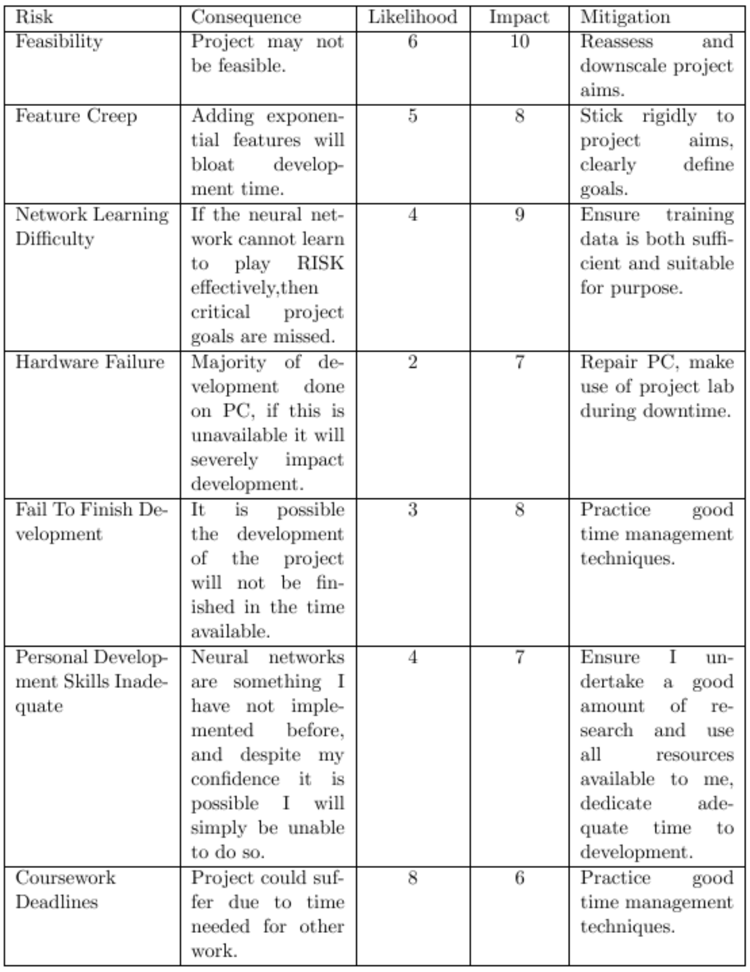
While writing the initial document for this project I identified a number of risks that had the potential to negatively impact development. This section will be used to analyse each of these risks as to any impacts that they had as well as any actions taken to mitigate these impacts.

Feasibility

Obviously one of the bigger risks to the implementation, if the implementation is not feasible then it simply cannot be done. I did find that some attempts at training were infeasible and did not work effectively given my current level of knowledge. For example attempting to train a neural network using a genetic algorithm as the network's weights proved to be impractical based on the relatively simple implementations and methods of training I was using. To rectify this particular instance I quickly abandoned this approach and looked into other methods of training.

Feature Creep

This risk was avoided by sticking to the algorithms noted in the initial document, although originally deep Q Learning was not something I



Risk	Consequence	Likelihood	Impact	Mitigation
Feasibility	Project may not be feasible.	6	10	Reassess and downscale project aims.
Feature Creep	Adding exponential features will bloat development time.	5	8	Stick rigidly to project aims, clearly define goals.
Network Learning Difficulty	If the neural network cannot learn to play RISK effectively,then critical project goals are missed.	4	9	Ensure training data is both sufficient and suitable for purpose.
Hardware Failure	Majority of development done on PC, if this is unavailable it will severely impact development.	2	7	Repair PC, make use of project lab during downtime.
Fail To Finish Development	It is possible the development of the project will not be finished in the time available.	3	8	Practice good time management techniques.
Personal Development Skills Inadequate	Neural networks are something I have not implemented before, and despite my confidence it is possible I will simply be unable to do so.	4	7	Ensure I undertake a good amount of research and use all resources available to me, dedicate adequate time to development.
Coursework Deadlines	Project could suffer due to time needed for other work.	8	6	Practice good time management techniques.

Figure 11: Risks identified in the initial document.

planned to use it was implemented on discovering just how effective a method it could be. So although this could be considered as feature creep I would personally consider it as the exploration of new approaches as more research into the task is carried out.

Network Learning Difficulty

This risk had a large impact on the development of the project. I went into this project relatively blind on the intricacies of training and so was unaware of just how challenging it could be. Training is not a simple cut and dry process, you need ensure you use training data that suit your purpose and constantly need to tweak both the implementation and training process as things that do and do not word are identified. A particularly large impact of this is when it held up the project during training of the TicTacToe AI, a lot of different processes were tried as detailed in the implementation section and some worked better than others. Although this did negatively affect the project it was still a valuable learning experience as I now know what will not work for future reference and was crucial to the development of the RISK bot.

Hardware Failure

Although no hardware failures occurred during development, I did realise my current laptop was impractical for demonstrations during the project fair. To rectify this I have invested into getting a new, vastly superior laptop which should serve this purpose well.

Fail to Finish Development

This risk did somewhat come to fruition. I was originally going hto detail the implementation of an AI for the game of connect four but was unable to due to issues with the implementation of Q-Learning in Tic-Tac-Toe. It would have been possible to implement before the deadline but not without putting the implementation of a RISK AI at risk. So the choice was made to not implement and train an AI for connect four and instead focus on RISK. This was actually very wasteful as I already implemented a fully working game using unity before this decision was made but did not want to dedicate time to training and tweaking an AI.

Personal Development Skills Inadequate

This project was originally meant as a learning experience and a learning experience it turned out to be. I found that for the implementation of back propagation I needed to do a lot of reading to understand it due to the new concepts involved, in particular gradients which I was never the best with. Through reading and analysis of implementations of the algorithm in other languages I was able to overcome my lack of understanding and develop a working implementation of the backpropagation algorithm.

Coursework Deadlines

This project was an immense amount of work and so needed a lot of time dedicated to it. This risk did not end up negatively affecting development of the project as I made sure to practice good working techniques. Getting other work such as coursework out of the way quickly in order to not let it drag on and dig into time that was needed for this project.

6 Implementation

6.1 Q-Learning

Q-Learning is a machine learning algorithm that at its core explores an environment and attempts to assign a value to every possible action in each state. This results in an optimal policy that can be followed by simply choosing the action with the highest value in each state. This subsection will cover the object oriented implementation of the Q-Learning algorithm using C#.

The finalised implementation uses four classes, the names and purposes of which are as follows -

Class	**Access**	**Function**	**Static?**
Experience	Internal	Holds information on a single state-action pair encountered by the algorithm.	No
Memory	Internal	Contains all the experiences encountered by the algorithm provides functionality to add to and access experiences it holds.	No
Learner	Public	The algorithms 'brain', this holds all functionality regarding calculating what action to take for certain state. It also provides functionality for learning.	No
Utility	Internal	A helper class that holds a single 'Random' object and provides functionality for generating random integers and doubles.	Yes

The full class diagram for this implementation is available in Figure 12.

The implementation is compiled to a dll named 'QLearning.dll' which can be freely imported into a user's project. Once they have achieved this they need to create a new 'Learner' instance, specifying their desired learning rate, discount factor, epsilon value and file name to save the learner's knowledge to. They then use the learner's 'ChooseAction' function, passing in the current state and possible actions to get the action chosen by the learner. The user then uses that action in their chosen environment and calculates a reward for doing that action. The user should then call the 'Learn' method, passing in the new state as a result of doing the chosen action, the possible actions of the new state, the reward and a boolean determining if the new state is a terminal state or not. The learner will then use this information to learn and will now be just that little bit smarter. The user repeats this process multiple times until the algorithm performs to their satisfaction, the learner's knowledge can then be loaded for future use.

Two of the more interesting sections of code are available in the appendix, the snippet of code used for choosing an action is in appendix A.1. The snippet of code responsible for calculating the value of a state-action pair is available in appendix A.2.

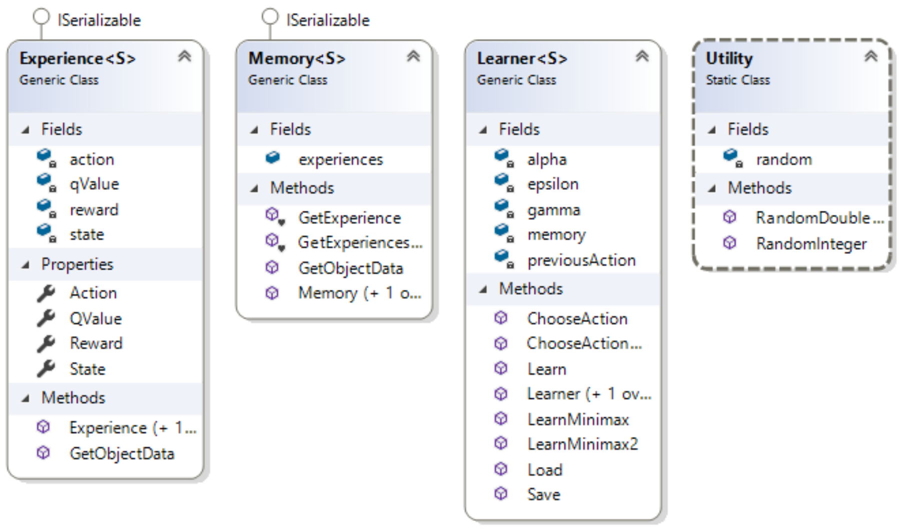


Figure 12: Full class diagram for this project's implementation of the Q-Learning algorithm.

6.2 Genetic Algorithm

A genetic algorithm is an evolution inspired algorithm specialised for solving optimization problems. It works by initialisation a population of individuals which each contain useful information in the form of 'genes'.

The fitness of each individual is then according to how close it is to the ideal solution, the fittest individuals are then used to generate a new population that on average is fitter than the last. In this implementation the fitness function is defined externally by the user and passed in as a 'func¡double[], int¿' which is a function that takes in a double array and returns an integer. So for each individual the individual's genes are passed into this function and the resulting integer is that individual's fitness rating. An example of a fitness function is available in appendix A.3.

The finalised implementation uses 5 classes, the names and purposes of which are as follows -

Class	Access	Function	Static?
Individual	Public	Holds the gene's and fitness of a single individual in the genetic algorithm.	No
Population	Public	Contains all the individuals in a single population, provides functionality to generate a new population and access the individuals contained within.	No
GeneInfo	Public	The gene's of a single individual, holds information on the gene's limits and type of data.	No
Breeder	Public	Holds the breeder settings such as mutation rate and uniform rate. Provides functionality for generating new populations using mutation, crossover and tournament selected.	No
Population Builder	Public	Provides functionality for the generation of an initial population and initialisation of the fitness function.	Yes

The solution compiled to a 'GeneticAlgorithm.dll' which can be freely used by third party projects. To use the implementation the user needs to initialise a new 'GeneInfo' object, specifying the type, size, minimum values and maximum values of the genes in each individual A function to determine the fitness of an individual needs to be written by the user which takes in a double array and returns an integer. The user then creates an initial population by calling the static 'CreatePopulation' function of the 'PopulationBuilder' class, passing in the size of the population, the instantiated 'GeneInfo' object and the defined fitness function. This returns a single 'Population' object which represents the initial population.

The user then instantiates a new 'Breeder' object, passing in the mutation rate and specifying whether the breeder should be elitist. Elitist in this case meaning the fittest individual in one population will always carry on to the next. The user then follows the following steps -

- Evolve the population by calling the 'Evolve' function of the breeder object.
- Calculate the fitness of each individual in the new population.
- Get the fittest individual.

This process repeats until the fittest individual with a suitable fitness rating has been generated. The genes of which can then be used for future solutions relating to the problem being solved.

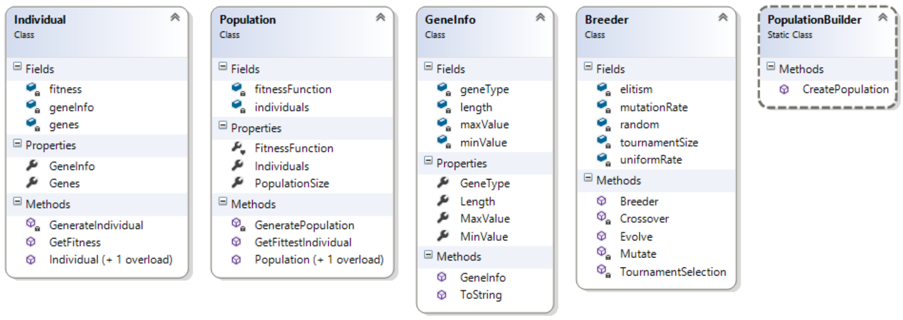


Figure 13: Full class diagram for this project's implementation of the Genetic algorithm.

6.3 Neural Network & Backpropagation

A neural network is a machine learning technique built up of layered interconnected neurons. The network must have at least a single input layer and a single output layer, and can have any number of hidden layers. At its core an output layer is functionally identical to a hidden layer but typically with a different number of neurons. The neurons in the input layer have their value's set to the desired input, the input is then fed forward through the network with each successive layer's neuron being the sum of the previous layer's values multiplied by the accompanying weights. Each neuron has an activation function that determines the output of the neuron. Typically the sigmoid function which has an output in the range of '0' to '1' is used for this purpose, refer to appendix A.4 for its implementation in code. Another commonly used function is the hyperbolic tangent function which has an output in the range of '-1' to '1', refer to appendix A.6 for its implementation in code.

For its implementation as part of this project an object oriented approach was taken, with a single neural network object being composed of multiple layer objects which in turn are composed of multiple neuron objects. Polymorphism is used with the abstract classes 'Layer' and 'Neuron' to prevent the need for multiple casts to refer to the multiple types of layers and neurons that make up a neural network. The 'NetworkBuilder' class is used to create 'NeuralNetwork' objects and is passed two functions which take a double and return a double, to be used as the network's neuron's activation function and derivative.

The full class list of the implementation is as follows -

Class	Access	Function	Static?
Neuron	Public	Abstract class providing the base functionality for the different types of neuron.	No
InputNeuron	Public	Provides functionality to set the value, no activation function.	No
HiddenNeuron	Public	The workhorse of the neural network, provides functionality for weights, an activation function and the ability to 'fire' the neuron.	No
BiasNeuron	Public	Holds a constant value of one, used to implement the network's bias.	No
Layer	Public	Abstract class providing base layer functionality, forces implementation of abstract functions for child classes.	No
InputLayer	Public	Provides functionality to set the values of the input neurons in holds, does nothing when 'Fire' is called.	No
HiddenLayer	Public	Also acts as the output layer, provides logic for the 'Fire' method which calculates the values for each hidden neuron it holds.	No
Synapse	Public	Acts as the weighted connection between two neurons, holds the originating neuron and the value of the weight.	No
NetworkInput InvalidException	Public	Custom exception thrown when user attempts to set a number of inputs that does not match the number of input neurons. Provides a custom message and the input/input neuron count.	No
NetworkOutput InvalidException	Public	Custom exception thrown when user attempts train the network with a number of outputs that does not match the number of output neurons. Provides a custom message and the output/output neuron count.	No
Back Propagation	Internal	Provides functionality to adjust the network's weights through the implementation of the back propagation algorithm.	Yes
NeuralNetwork	Public	Provides functionality to allow for the use of the neural network in other projects. Also provides saving, loading and accessors to retrieve data.	No
NetworkBuilder	Public	Static class which provides methods to initialise, set up and return a NerualNetwork object. Used to control external access to NeuralNetwork creation.	Yes

Although backpropagation is distinctively different concept to that of a neural network. Backpropagation is used in this project as a method of training neural networks and so was implemented as part of the neural network implementation. This provides a built in method for training the neural network by calling the 'Train' method of the 'NeuralNetwork' class. The algorithm is implemented as follows -

- Calculate the error of each output neuron in the output layer, refer to appendix A.8 for code.
- Moving backwards, for each hidden layer calculate the error of every hidden neuron. The code to calculate the error of a hidden neuron can be found in appendix A.9.
- Once the error of every hidden and output neuron has been calculated, adjust the weights of each neuron depending on its error. Review appendix A.10 for the code to carry out neuron error adjustment.

For training to be successful the backpropagation needs to be repeated multiple times, typically in the range of the 10000s. It is important to ensure the training data is spread out so that similar data is not used in large chunks. This acts to prevent the network being adjusted for a particular range of values and then essentially forgetting what it learned when it is trained for extremely different data. This is avoided by using all unique training data to train the network before using the same data again.

The derivatives of the activation functions are used to get the original values of the neurons before the activation function was used. The code for the derivative of the sigmoid function is in appendix A.5 and the derivative for the hyperbolic tangent function is found in appendix A.7.

6.4 Deep Q-Learning

Deep Q-Learning is the use of a neural network to approximate the Q-Values of the Q-Learning algorithm, instead of using the traditional Q-Learning method of storing every experience in memory. This allows the Q-Learning algorithm to be applied to problems with state spaces that would be much too large to store using traditional methods. The concept is rather simple with the state/action pair being fed to the neural network as input and the output being interpreted as the value of doing that action in that state.

Informally, the deep Q-Learning process to choose an action is as follows -

- Agent is in some state S.
- Agent can do action a or b to move to a new state.
- The Q-Value of each action is approximated by feeding each state-action pair as input into the neural network, running it and getting the output.
- The action with the highest Q-Value is then chosen.

The training process is different from the typical Q-Learning training process. Q-Learning training is done by updating the Q-Value of an action based on the stored Q-Values of a future action, but as nothing is being stored in deep Q-Learning a new approach is needed. A logical approach to training would be to get the approximated Q values of every action in the next state and use that to calculate the target to train the network towards, but neural networks perform poorly when trained with non-linear training. The solution to this is a concept called "Experience Replay", where action-state pairs and their rewards are stored as they would be in the normal Q-Learning algorithm, with older experiences being removed as the number of experiences approaches too large a number. Each time the network is trained a random sample is then taken from the stored experiences and used to train the network. This breaks up the similarity of subsequent training samples to prevent poor performance in the network as a result of it getting stuck in a local minimum. Another benefit of this approach is that as stored samples are used for training it makes debugging easier as it becomes closer to a supervised learning technique. Although, at its core it remains a reinforcement learning technique.

Hence, the training process is as follows -

- Choose an action, store the resulting reward and state as an experience.
- Once a threshold has been met, take a random sample of experiences.
- For each experience, if the new state is a terminal state then the target is equal to the reward, otherwise the target is equal to the reward + the maximum approximated q value of all actions in the next state.
- The network is trained with the expected value being the target.

The Deep Q-Learning algorithm is used to great success at Deep Mind and is very capable, although it is challenging to effectively implement as setting up rewards and training data is a difficult task.

6.5 Games

6.5.1 Tic-Tac-Toe

A turn based game known by every school child, Tic-Tac-Toe is a two player game played on a 3x3 grid where each player takes turns to mark an empty section of the grid with an 'X' or an 'O'. The goal of the game is to get 3 in a row of your shape and prevent the opposing player from getting 3 in a row of their shape. This game is simple to play and simple to implement, with a logical array based structure that makes it ideal for the implementation of machine learning algorithms.

The implementation is intended to be used as a black box. Used by instantiating and starting a new 'TicTacToeGame' object which has two event handlers 'GameOverEvent' and 'PlayerTurnEvent' which need to be hooked into by the program implementing the game. Whenever it is a players turn the 'PlayerTurnEvent' fires which passes the current player whose turn it is through the use of a custom EventArgs 'PlayerTurnEventArgs'. When the game is over the 'GameOverEvent' fires, passing a 'GameOverEventArgs' which contains the winning player. It is up to the user to provide functionality to handle these two events. The implementation comes built in with the Minimax algorithm, capable of picking a perfect move to guarantee at the very least a draw. A move is then made by calling the 'MakeMove' method, which validates and makes the move, then invokes the turn for the next player. The move should be validated before being passed to this method through the use of the boolean function 'MoveValid'.

The implementation of Tic-Tac-Toe is composed of 8 classes as follows -

Class	Access	Function	Static?
Square	Public	Single square in the grid, holds information on the current owner of the square and the current shape it contains.	No
Board	Public	Implements the game's grid as a 2d Square array, provides functionality for making and validating moves, as well as getting the board state as a 1d integer array.	No
Move	Public	Holds information for a single move to be made on the grid, used as opposed to an integer due to the 2d nature of the grid.	No
Player	Public	Holds player information such as name, shape, and game stats.	No
TicTac ToeGame	Public	Holds the game's logic, invokes events and provides functionality for external interaction with the game.	Yes
GameOver EventArgs	Public	Passed when the GameOverEvent is invoked, holds a reference to the winning player.	Yes
PlayerTurn EventArgs	Public	Passed when the PlayerTurnEvent is invoked, holds a reference to the current player.	Yes
MiniMax	Private	Implements the Minimax algorithm and functionality for getting the best move for any given board state.	Yes

6.5.2 Game Of Less Risk

The game of risk is heavily documented in the background section of this paper so it won't be repeated here. But the game implemented, affectionately dubbed 'Game Of Less Risk ' differs slightly from the typical rules of the normal game of risk.

In the game of risk battle is done by sending a number of units to attack an adjacent, hostile region. The attack then occurs with 3 troops at a time with dice being rolled to determine who the victor is. The issue with this is that it is completely possible to attack a single unit with 20 units and come out the loser in the situation. This would pose a problem during training of the RISK AI as it would get completely different results for the same action each time, for example if it attacks 3 units with 5 and wins without taking any losses, it would be rewarded accordingly and so be more likely to do that action in the future as it is beneficial. But then it does it again and manages to inflict no losses while having its attack completely wiped out. This poses an issue because the AI is not going to learn anything useful from this information as it will be trained to do an ever changing action based on its ever changing reward for the same situation.

To solve this problem it was decided that the rules around attacking would be simplified and the random aspect removes, leading to the name 'Game Of Less Risk ' as there is now less risk in making attacks. When an attack is made now an an algorithm is applied to determine how many troops the attacker destroyed, with more attacking troops leading to more damage. The defender then counter-attacks and the same logic is applied to calculate losses for the attacking army, although if the defender is victorious the same rule applies that they do not gain the attacker's territory. The code used to determine damage dealt is available in appendix A.11.

The implementation is difficult to describe due the use of Unity in the development process. Unity makes use of editor based 'GameObjects' to which scripts are attached. Each script which is attached inherits from a 'MonoBehavior' class, providing a range of useful methods such as 'onStart' and 'onUpdate' allowing for logic to be called when the script is started or every update loop which happens multiple times a second. Unity also provides a wealth of UI components from which scripts can also be called. Due to Unity's design process of lots of small, lightweight scripts, it is hard to describe the implementation as has been done for the other solutions in this project. Despite this the core principles are rather simple and so that with a selection of the key classes is detailed in the diagram shown in figure 14.

A brief explanation is that the turn timer fires every second, which triggers the tracker to execute the logic for the current phase. The phase then gets the move from the current player and increments the player, if the phase is over then the phase increments to the next phase. Of note is that the placement phase only occurs at the start of the game and so the game logic will only go between the attack and reinforcement phases. In program structure terms the logic for each phase is kept a relevantly named static class and the methods of which are called from the Tracker class, which also keeps track of the current player and current phase.

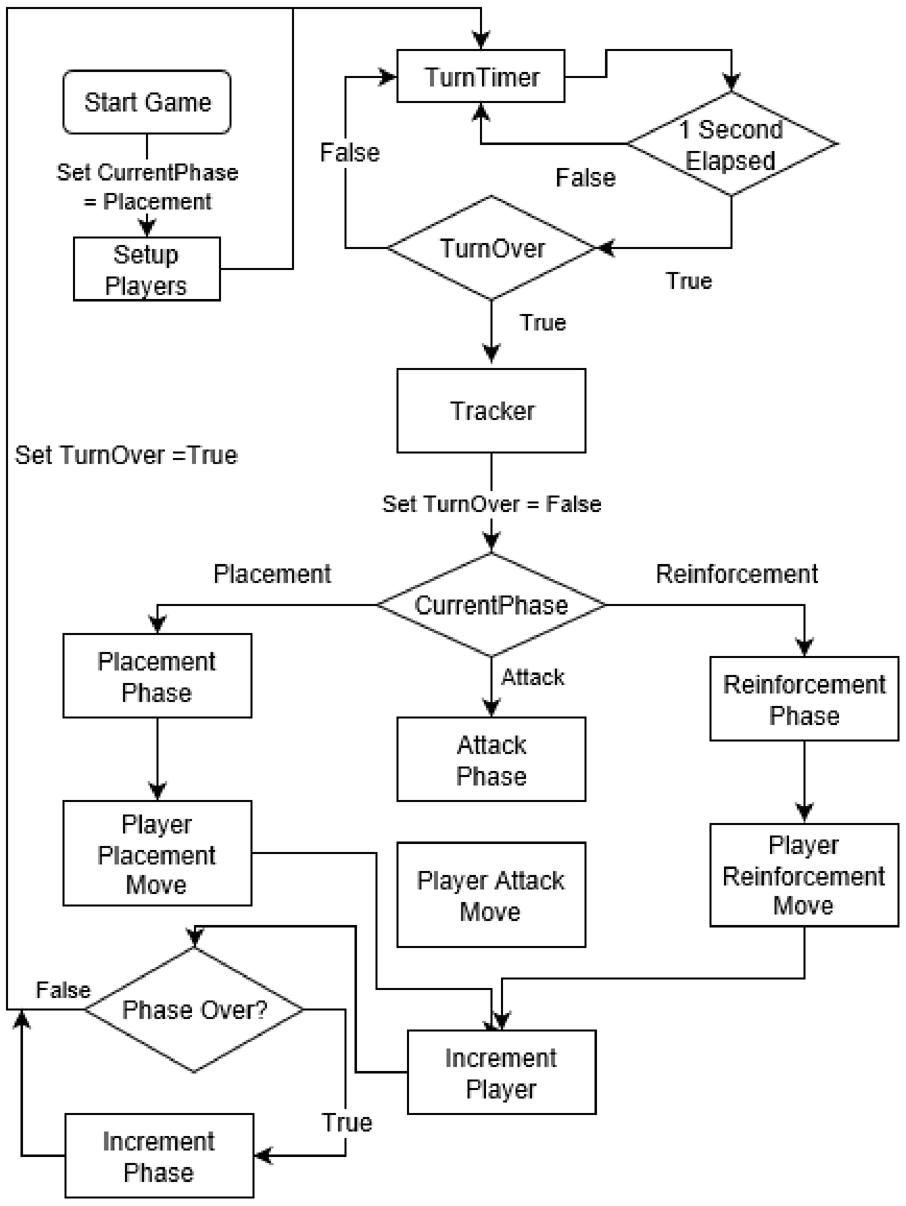


Figure 14: Game logic loop of RISK implementation.

7 Testing & Results

7.1 Genetic Algorithm

The genetic algorithm was the first concept implemented and tested during this project's development. Unfortunately, initial results were poor and the idea of implementing the algorithm in any future attempts was quickly discarded.

The main approach was to use the genetic algorithm as the weights of a neural network, with each element of an individual's genes corresponding to the weight between two neurons. This was tested using the implementation of tic tac toe as a means of training the network. The following steps were used -

- Initialise the genetic algorithm and generate an initial population.
- For each individual in the population, initialise a neural network with the individuals gene's and get the win percentage of 1000 games against a random AI, alternating between going first and going last.
- Pick the individual with the highest fitness and generate a new population.
- Repeat until an individual with a fitness over 90 was generated.

This process resulted in poor results. The network failed to improve in any real capacity and the general process was slow, as with a population of 50 it would take 50000 games to fully test the capabilities of one population. Meaning this process was both impractical and extremely slow and so it was decided that this approach would not be perused as it would need improvements and optimisation beyond the scope of this project.

Overall the genetic algorithm was a bit of a gamble, at its core it is intended mainly for optimisation problems and although training a neural network could be considered as such its not recommended If this approach were to be reconsidered a different means of judging fitness would be ideal. Further experimentation into the network's structure, learning rate and the genetic algorithms mutation and crossover rate would have to be looked into. A code snippet of the fitness function is available in appendix A.12.

7.2 Playing Tic-Tac-Toe

TicTacToe served as an excellent test bed for testing a variety of machine learning techniques. This is due to its simple, turn based nature with a logical state space that easily converts to a 2d array. This subsection will go into detail the approaches taken with attempting to create an AI capable of effectively playing TicTacToe and the results of doing so.

7.2.1 Approaches

7.2.2 Genetic Algorithm

As detailed in an earlier subsection, the first attempt was undertaken with a neural network optimised using a genetic algorithm. Results were poor and the pursuit quickly abandoned in favour of more promising techniques such as backpropagation and Q-Learning.

7.2.3 Neural Network & Backpropagation

Neural networks are excellent for use in classification problems, and it can be argued that playing RISK could be reduced to a classification problem. Due to the fact TicTacToe is a solved game, for every board state there exists a perfect move that will ensure at the very least a draw. Meaning that for every unique board state there exists a unique move, something that a neural network can be trained to find.

The network architecture used consisted of 9 input neurons for each square on the board, a single hidden layer of 5 hidden neurons and an output layer of 9 output neurons corresponding to each square of the board. The network was trained using the Minimax algorithm which is able to pick the perfect move for any board state, the idea being that the network would be trained to output the same value as the minimax algorithm. The following steps were used to train the network, after it was initialised with random weights -

- Set the input of the network to the current board state.
- Run the network.
- Get the perfect move for the current board state using the minimax algorithm.
- Train the network to output 1 for the output neuron corresponding to the minimax's chosen move, and 0 for every other output neuron.
- Make the move as chosen by the minimax algorithm, ensuring the network gets used to the ideal states.

The concept was simple, yet highly effective. This training procedure was used to train two neural networks, one playing against a perfect player and one playing against a player which would make moves at random. Each process was repeated 10000 times, the results of which are available in the next section.

7.2.4 Results

Results are meaningless without something to compare them with, so initially an untrained network with randomly initialised weights was tested by playing 1000 games against a random player and a perfect, minimax player. The results of this testing are shown in Figures 15 and 16.

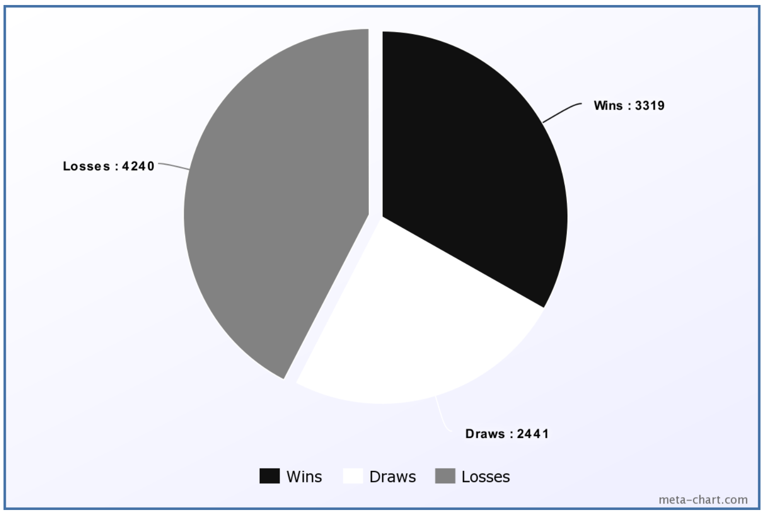


Figure 15: Results of an untrained network vs a random player.

As expected the untrained network was absolutely dominated by the minimax algorithm, with the network getting lucky only 34 times and scraping a draw. It was impossible for the network to win because the minimax algorithm is perfect and cannot lose. The results of the untrained vs random are also unsurprising, as an untrained network is making random moves it was essentially a random player vs a random player explaining the relatively evenly distributed results. Now these results have been collected they will be useful in determining the effectiveness of a trained network.

A network was then trained by playing 10000 games against the minimax algorithm and tested vs a random and perfect player. The results of which are available in Figures 17 and 18.

These results are extremely interesting and quite enlightening, in

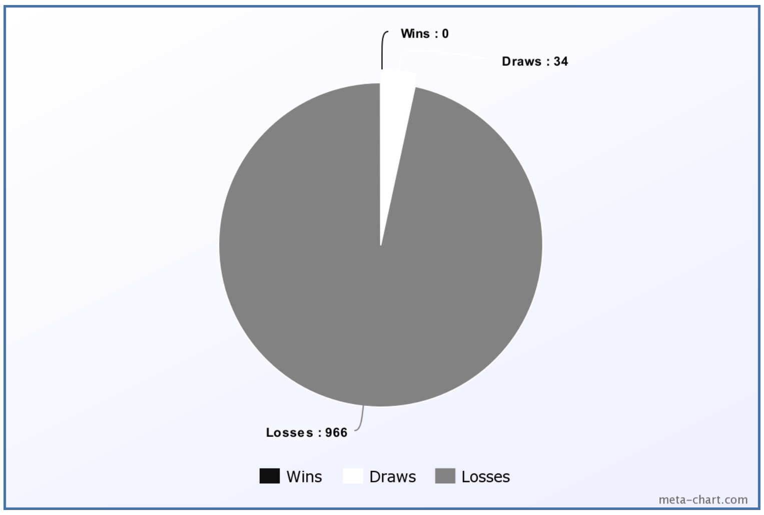


Figure 16: Results of an untrained network vs a perfect player.

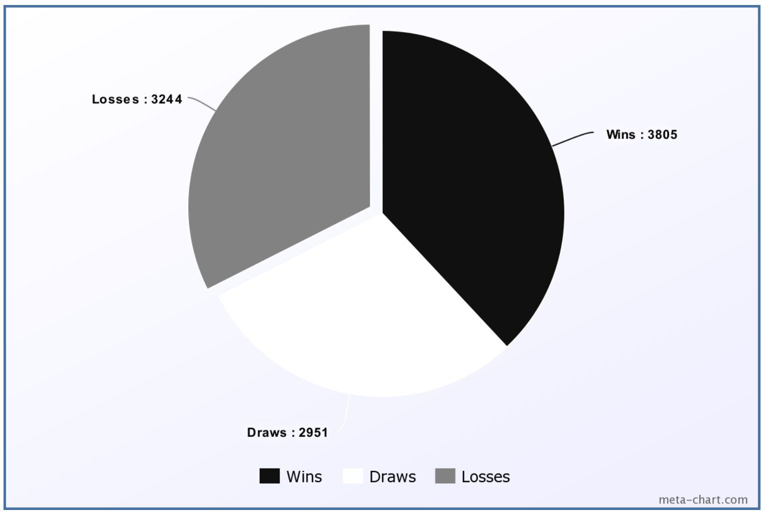


Figure 17: Results of a network trained using a perfect player vs a random player.

particular the results of the games against the perfect player. The network was capable enough to draw every game against the perfect player and the reason for this is actually quite clear. At its core the minimax algorithm has no random element and will return the same value every time for the same state, so the network simply learned the perfect moves against the exact combination the minimax algorithm would play every time. If an element of randomness was added to the minimax algorithm, such as picking a random move out of the available perfect moves as opposed to

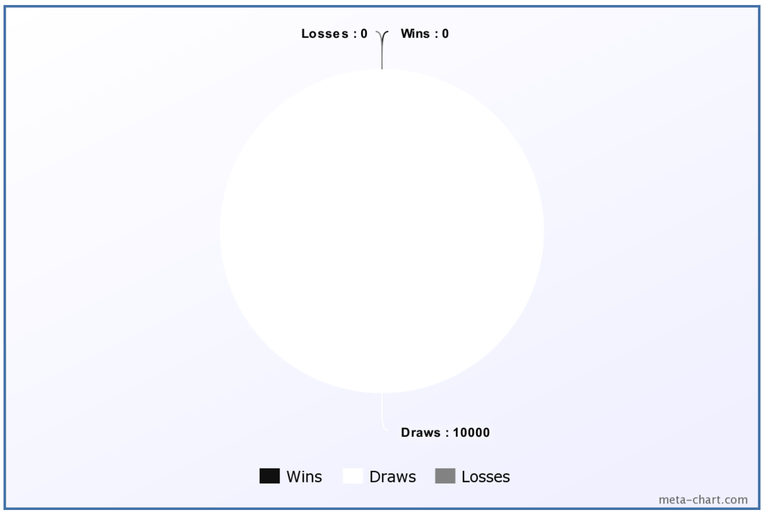


Figure 18: Results of a network trained using a perfect player vs a perfect player.

the first in the list then we would see different results.

On the other hand the results from the games against the random player were rather disappointing. Although there is some notable improvement as the number if wins and draws has increased while the number of losses has decreased, it is not to a particularly spectacular level. This is due to the fact the network was trained against the specific combination played by the minimax algorithm as evidenced by the other results, and so unless the random player played as the minimax the network was essentially playing random moves. Although this was the case the network did have a slight edge over the random player due to its training against the perfect player and so some measure of success was had.

The final stage of testing was training a neural network against a random player, achieved as it was with the perfect player but with the opponent making random moves. The trained network was then played against a random and perfect player. The results of which can be found in Figures 19 and 20.

This resulted in great success for the neural network, with massive improvements against both the random and perfect player. Against the random player the network achieved far greater results over the network trained with the perfect player. With over a 90% increase in win rate the randomly achieved network performed extremely well. This is most likely due to the random nature of its training, allowing it to slowly adjust its values to the correct result for lots of different states and situation,

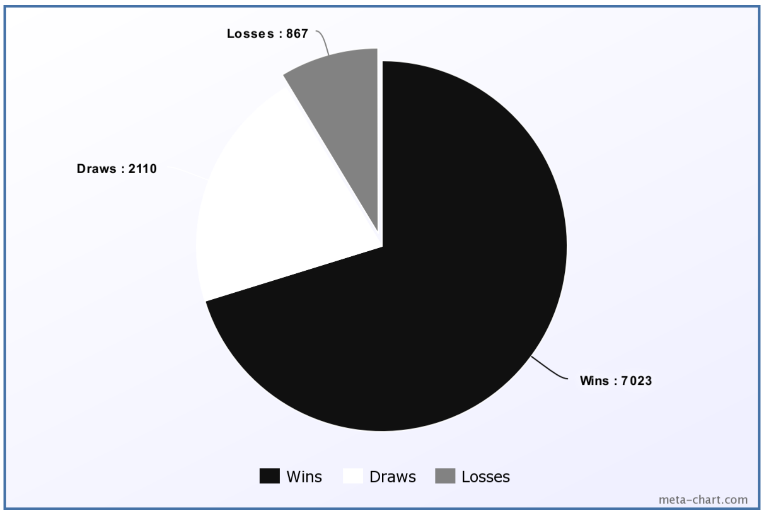


Figure 19: Results of a network trained using a random player vs a random player.

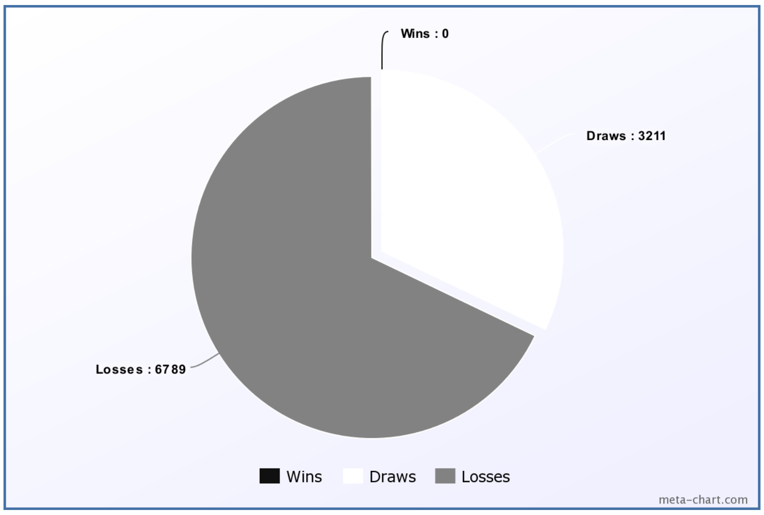


Figure 20: Results of a network trained using a random player vs a perfect player.

effectively decreasing its overall error over time for every possibility, as opposed to minimising its error for one possibility. the network also performed relatively well against the perfect player, with a smaller loss rate and by extension larger draw rate.

In conclusion, this testing has revealed that the data used in training a neural network is extremely important and a crucial part in any task involved with a neural network. Initial results are promising and this testing proves the effectiveness of the neural network and backpropagation implementations, meaning a core aim of this project to develop a custom working implementation of a trainable neural network has been reached.

7.2.5 Q-Learning

Reinforcement learning was also used in an effort to develop a TicTacToe playing AI. In this case a bot implementing the Q-Learning algorithm learned to play TicTacToe by playing against itself. This was achieved through taking the concept of 'min' and 'max' from the minimax algorithm and modifying the Q-Learning implementation to implement something called Minimax Q-Learning.

Minimax works by picking an action that maximises its own value and minimises the opponents value. It does this by recursively looping through the states and calculating the value of making each move in each state, alternating between min and max where min is an opponents action and max is the maximisers action. It makes the assumption that the opponent will also play perfectly and so picks actions leading to a high value state for itself and avoids actions which lead to a high value state for its opponent. These concepts are applied to Q-Learning to create Minimax Q-Learning.

To implement this the methods 'ChooseAction' and 'Learn' were edited, the code of which can be found in appendix A.13 and A.14.

Two Q-Learner bots with a shared memory were used as player one and two, with the first acting as the maximiser and the second as the minimiser, so as to emulate the minimax algorithm The following steps were used to train the Q-Learner.

- Call 'ChooseAction', passing the current state of the board and possible actions.
- Perform the chosen action, receive a reward from the game. 1 for victory, -1 for defeat, 0.5 for a draw and 0 for a non game over scenario.
- Call the 'Learn' function with the resulting state and reward to train the q learner.

The AI was trained by playing 400000 games against itself and then tested against a random and perfect player, the results of which are available in Figures 21 and 22.

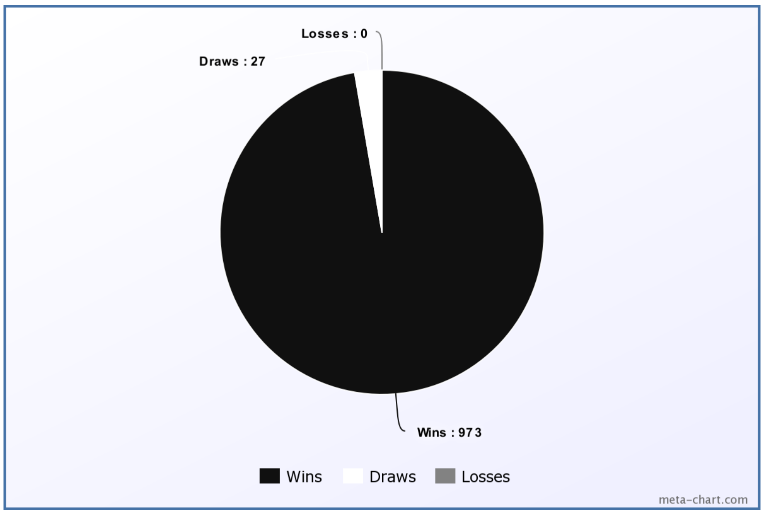


Figure 21: Results of a trained Q Learner vs a random player.

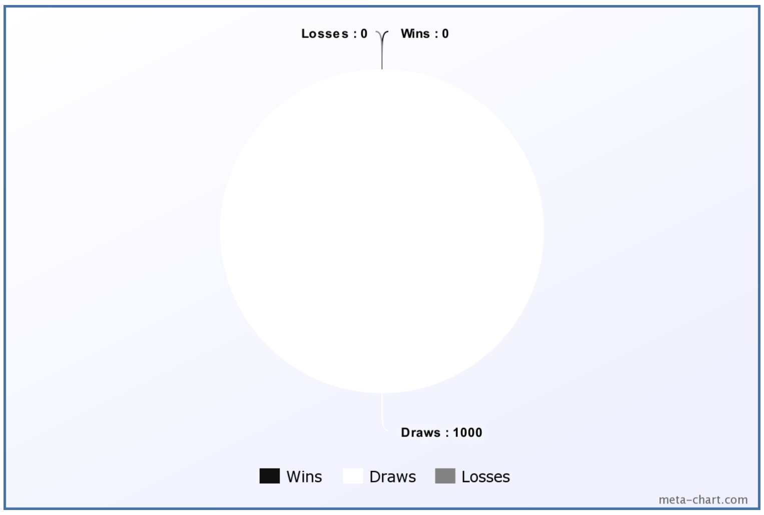


Figure 22: Results of a trained Q Learner vs a perfect player.

The results show that the Q-Learner AI is a resounding success, with results that blow that of the neural network's out of the water. The AI is unbeaten by both the random and perfect players, with the learner well suited to countering the perfect player's policy as well as being capable of beating the random player the vast majority of the time.

These results are particularly remarkable due to the fact they were obtained through the use of reinforcement learning, so even though the AI had no knowledge of how to play TicTacToe or what the best moves were, it was capable of learning just through obtaining a reward based on the games outcome. This shows that reinforcement learning is an extremely capable method of applying machine learning to a problem, even for situations where supervised learning is feasible.

A downside to this approach compared to the neural network's is the length of time it takes to train, with this AI in particular taking 9 hours to train with 400000 games. This is due to the Q-Learner needing to navigate a massive amount of stored data every time it chooses an action or learns, and is something that cannot really be avoided as it is at the core of the algorithm.

7.3 Playing Connect Four

Originally Connect Four was going to be used as a middle ground between TicTacToe and RISK implementing machine learning techniques. Due to time constraints this was abandoned in favour of focusing on RISK, unfortunate as the time invested in implementing Connect Four itself was essentially wasted effort. Although it did come with the advantage of refining development techniques for developing the implementation of RISK.

7.4 Playing Game of RISK

As discussed in the background section, changes were made to the game of risk to remove the random aspects of attacking. Apart from this the rules of risk remain unchanged and operate as directed in the game's manual.

7.4.1 Approach

Q-Learning would be the first choice as a learning technique to use in the RISK bot due to its excellent performance in the TicTacToe testing. This cannot be done though due to the extremely high number of possible states in the game of risk. With over 20 regions and the potential for an arbitrary number of troops and owners for each region the state space is simply too large for traditional Q-Learning to be applied in any manner that is at least partially effective.

The solution to this issue is to make use of Deep Q-Learning, as described in the implementation section. This algorithm combined Q-Learning and a neural network to allow Q-Learning to be applied to problems with very large state spaces, making it perfect for the task at hand.

The first stage is to recognise the distinct actions that can be undertaken while playing risk, these have been identified as placing a unit in the initial placement stage, attacking in the attack phase and reinforcing in the reinforcement phase. Because of this it was decided that the AI would be composed of three distinct 'brains' which would each serve the purpose of carrying out each action. A single brain could not realistically be used due to the different nature between the possible actions, an AI dedicated to reinforcing is going to have very different goals to an AI dedicated to attacking.

This section will now be used to detail how each 'brain' is implemented, how it is rewarded and how it trained.

Placement Brain

The simplest and least used component of the AI, the placement brain is responsible for deciding where to place a particular unit during the placement phase. The action is chosen by looking at each region in turn and calculating the predicted value of placing a unit there.

The input structure is as follows -

- Region continent ID.
- Surrounding regions owned.
- Number of regions in same continent owned.
- Continent bonus troop value.
- Surrounding continents owned.
- Max regions in continent owned by single opponent.

The logic behind this choice is that when initially placing units you are looking to form blocks of regions, ideally of the same continent and preferably with a higher troop bonus. So, this data is fed as input into the brain so the AI knows the conditions of the continent it is currently considering and calculates the value of placing a unit there. The player should also consider what would most harm the opponent, so if they can block the opponent from capturing a continent then they should do so. For this purpose the most number of regions in the continent owned by an enemy player is passed in. As with Q-Learning the action with the highest value is chosen.

In terms of training, RISK is not a solved game and so there are no 'wrong' moves. Because of this a different means of rewarding the placement brain was devised. This took the form of the following table, where if making that move resulted in any of the following conditions that reward was granted.

Result of Action	**Reward**	**Reasoning**
Continent Captured	+ 1	Capturing a continent is the highest priority due to the troop bonus.
Continent Denied	+ 0.75	Not as good as capturing a continent, but denying the opponent troops is crucial to victory.
Region Captured	+ 0.5	Always good to encourage region capture!
Region Invalid	+ 0.0	Discourage the bot from making invalid moves.

The reward gained as a result of the chosen action is then used to store an experience for future learning.

Attack Brain

The second component of the AI is the attack brain. Responsible for deciding when and where to attack. The action is chosen by looking at each region and each region that neighbours that region, using the features of both such as number of troops, continent to decide on the value of the action.

The input structure is as follows -

- Region continent ID.
- Neighbour continent ID .
- Number of troops in region.

- Number of troops in neighbouring region.
- Neighbour region ID.

The logic behind this choice of features is that you typically want to attack regions with which contain a low number of troops, or only attack when you have some numerical advantage. It is possible this data will need tweaking though as testing develops as the AI may need more information to make a logical choice. For example a region threat level function could be devised which calculates the threat a particular region is under from enemy troops, or a value function to determine how attractive a certain region is depending on its continent and the player's continent.

Again, a reward scheme has been devised to encourage the bot to make moves which will benefit it in the long run.

Result of Action	**Reward**	**Reasoning**
Region Captured	+ 1	Capturing regions is always good.
Battle Won	+ 0.5	A 'won' battle is defined as a battle where the enemy ends up with less units then you after damage calculation.
Battle Lost	+ 0	A 'lost' battle is defined as a battle where the enemy ends up with more units than you after damage calculation.
Action Invalid	+ 0.0	Discourage the bot from making invalid moves.

Reinforcement Brain

The final component of the AI's brain, the reinforcement brain is responsible for deciding which regions to reinforce with troops gained at the end of each turn. It decides this based on the number of enemy troops surrounding a particular region, as well as if the player owns the entire continent the region belongs to. The logic behind this being that regions at higher risk should be reinforced but regions which belong to an owned continent should take a higher priority, as they are more valuable than a single isolated region.

The input structure is as follows -

- Region continent ID.
- Neighbouring enemy troops.
- Region continent owned.

Another reward scheme has been devised, this one is the most subject to change as it is hard to judge a good reward outside of feedback in the form of enemy attacks. As of now the following stands -

Result of Action	**Reward**	**Reasoning**
Region Reinforced	+ 0.2	Encourage the bot to make correct moves.
Owned Continent Reinforced	+ 0.4	Encourage the bot to reinforce valuable regions.
Action Invalid	+ 0.0	Discourage the bot from making invalid moves.

Now that the structure of the AI has been devised and implemented, it is time to train the AI. This process will take a long time due to a combination of how long a game of RISK takes, the large amounts of data involved and the relatively slower learning process of Deep Q-Learning as opposed to traditional Q-Learning which in turn is already slower to train than a neural network.

7.4.2 Results

The bot is still in the middle of learning and so no results can be collected at this stage. If it were finished then the process would be to play around 100-1000 games depending on the amount of time available and calculate the average score over the games played. This would then be compared to the random AI too see how it compared to a player making purely random moves.

The current aim is to complete the training process in time for the project fair, so that way something more interesting can be presented and meaningful results can be collected for display and reference. Failing this the underlying concepts, structure and processes can still be explained. As well as the trained TicTacToe bots which perform competently.

8 Conclusion

8.1 Reflection of Results

The main goal of this project was to develop a deeper understanding of the concepts surrounding machine learning, a goal which has definitely been met as a result of this project. During the course of development a large variety of concepts were explored, researched and implemented in an effort to further understand them, typically to great success. In particular the flawless implementation of a neural network and back propagation are the crowning achievements of this project, something I was essentially clueless about a year ago and now much more knowledgeable in.

The genetic algorithm and Q-Learning were also successfully implemented as per the aims of the project. Although the genetic algorithm performed poorly during its testing with TicTacToe this is not due to any particular fault with the implementation itself. More due to the fact it was not well suited for the task it was being applied too and due to my inexperience with its use. Q-Learning was also a resounding achievement, resulting in an AI capable of playing TicTacToe to a high level as a result of playing against only itself with no outside interference. The implementation of which also allowed for the development of Deep-Q-Learning, suitable for use in the RISK AI.

The TicTacToe testing results exceeded my expectations. I assumed I would receive poor results in my attempts due to my lack of experience with the application of machine learning. Although this was initially true, I did not decide to just accept it and move on, instead I worked at the problem and refined my solution to get the results seen in this paper.

In regards to RISK, it is a pity that a trained AI was not produced in time for the submission of this paper. This was mainly due to the late discovery of Deep-Q-Learning which proved a breakthrough in this projects development. Which I was able to rapidly implement due to it being a combination of two techniques I had already thoroughly researched an successfully implemented, making the implementation of Deep-Q-Learning a trivial task. The only issue with the AI thus far is the extremely slow rate of training, but this is due to the fact it needs to play a full game of RISK many times over using massive amounts of data, as opposed to any obvious flaws in the implementation.

In regards to my personal performance in this project. One critique I would have is my tendency to not focus on one concept for long. For example the implementations of the neural network and Q-Learning were developed simultaneously, as I would flit back and forth between them as they took my fancy. This arguably impacted my project negatively as it means I had to do training for multiple implementations at the same time, as opposed to training for one implementation as I developed the other. For future projects of mine this is something I will have to take care in avoiding.

The main point to take away from this paper is not to underestimate the task that is training the AI. Training takes a long time with data that needs to be very carefully curated and applied to the learning technique used. A main drawback of training is the time that it takes to actually develop something that works, but in the end it is usually worth it. When you train an AI incorrectly it massively impacts on the duration of your project, resulting in the need to go back, identify the issues and retrain the AI, with no particular guarantee that it will work this time. The simple fact is that training is as difficult, if not more so, than the implementation of the actual machine learning algorithms themselves. Implementing a neural network was easy compared to how long it took to refine the Q-Learning process with TicTacToe.

In conclusion, I would say my project succeeded in its goal to provide a substantial background on the field of machine learning. I certainly learned a lot during its development and believe that machine learning has a lot to offer humanity.

8.2 Potential For Further Development

There are numerous options for future development of this project, a major example being putting the implementation of Connect Four to use. Due to time constraints no machine learning techniques were applied to this game meaning the development of which was a massive waste of time, contributing to the current situation of being unable to test the trained RISK AI. Another option which would also help to alleviate the slow training issue is to make use of the GPU to accelerate the learning process, as the GPU is a much more capable mathematical processor than the CPU. The CPU used during this project was a rather old model of an I5 while the GPU is a much newer GTX 970, meaning GPU acceleration would of benefited the project greatly.

Further refinement of the Deep Q Learning implementation would be another goal for future development, there are additional concepts not explored in this paper such as error and reward clipping. Both of which are a great benefit to optimising the experience replay learning process and so would also help in alleviate the slow learning issue. Finally, development of the implementations will continue past the submission of this document in preparation for the project fair, with a particular focus on RISK and Deep Q Learning. It is likely that RISK will be ready by the time of the project fair, but further development never hurts.

References

[1] BBC News. *Google AI wins second Go game against top player.* News article. http://www.bbc.co.uk/news/technology-35771705 (as of 28/04/2017)

[2] BBC News. *Ford's self-driving car 'coming in 2021'.* News article. http://www.bbc.co.uk/news/technology-37103159 (as of 28/04/2017)

[3] BBC News. *Legal breakthrough for Google's self-driving car.* News article. http://www.bbc.co.uk/news/technology-35539028 (as of 28/04/2017)

[4] Marketwatch. *Elon Musk: Self-driving Teslas will go between LA and NYC by the end of the year* News article. http://www.marketwatch.com/story/elon-musk-self-driving-teslas-will-go-between-la-and-nyc-by-the-end-of-the-year-2017-04-28 (as of 30/04/2017)

[5] The Guardian. *Self-driving trucks: what's the future for America's 3.5 million truckers?.* News article. https://www.theguardian.com/technology/2016/jun/17/self-driving-trucks-impact-on-drivers-jobs-us (as of 28/04/2017)

[6] IBM. *IBM Watson Health.* https://www.ibm.com/watson/health/ (as of 28/04/2016)

[7] Microsoft. *Cortana - Meet your personal assistant - Microsoft.* https://www.microsoft.com/en/mobile/experiences/cortana/ (as of 28/04/2016)

[8] J. S. Esmaail. *Playing video games using neural networks and reinforcement learning.* Bachelor Dissertation. Swansea University. 2016.

[9] G. Tesauro. *Temporal Difference Learning and TD-Gammon.* Communications of the ACM 38.3. 1995. 58-68.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, I. Antonoglou, D. Wierstra, M. A. Riedmillier. *Playing Atari with Deep Reinforcement Learning.* CoRR 1312.5602. 2013.

[11] C. E. Shannon. *Programming a Computer for Playing Chess.* Springer New York. 1988. 2-13.

[12] R. Myerson. *Game Theory: Analysis of Conflict* Harvard University Press. Cambridge. 1991.

[13] V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* Ponsen & Looijen. 1994.

[14] E. W. Weisstein. *Sigmoid Function* http://mathworld.wolfram.com/SigmoidFunction.html (as of 26/04/2017)

[15] M. Mazur. *A Step by Step Backpropagation Example* https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/ (as of 30/04/2017)

[16] G. S. Hornby, A. Globus, D. S. Linden, J. D. Lohn. *Automated Antenna Design with Evolutionary Algorithms* American Institute of Aeronautics and Astronautics. Space 2006. 7242. 2015.

[17] M. Mohri, A Rostamizadeh, A Talwalkar. *Foundations of Machine Learning* MIT Press. 2012.

[18] C. J. C. H. Watkins. *Learning From Delayed Rewards.* PHD Dissertation. University of Cambridge. 1989.

[19] C. Gaskett, D. Wettergreen, A. Zelinsky. *Q-learning in Continuous State and Action Spaces* Australasian Joint Conference on Artificial Intelligence. 1999 Dec 6 (pp. 417-428). Springer Berlin Heidelberg.

[20] G. Tesauro. *Temporal Difference Learning and TD-Gammon.* Communications of the ACM 38.3. 1995. 58-68.

[21] R. S. Sutton. *Learning to predict by the methods of temporal differences.* Machine learning 3.1. 1988. 9-44.

[22] M. Lütolf. *A Learning AI for the game Risk using the TD(λ)-Algorithm*, Bachelor Thesis, University of Basel, 2013

[23] No Author Attributed, under Creative Commons. *Risk Board Map.* $https://en.wikipedia.org/wiki/Risk_(game)$ (As of 04/05/2017)

[24] Hasbro. *Risk: The World Conquest Game.* 1993. http://www.hasbro.com/common/instruct/risk.pdf (as of 04/05/2017)

[25] Unity Technologies. *Unity - Game Engine*, https://unity3d.com/ (as of 28/04/2017)

[26] Microsoft Corporation. *Any Developer, Any App, Any Platform — Visual Studio*, https://www.visualstudio.com/ (as of 28/04/2017)

[27] Microsoft Corporation. *.NET - Powerful Open Source Cross Platform Development*, http://www.microsoft.com/net/ (as of 28/04/2017)

[28] Author Name. *Reference Name*, Version, Publisher, Year.

A Appendix

A.1 Q-Learning Choose Action

```
//Get the experiences for this state.
List<Experience<S>> experiences = memory.GetExperiencesForState(state, possibleActions);

Get the experiences with the highest Q Value, there may be more than one with identical values.
experiences = experiences.Where(x => x.QValue == experiences.Max(m => m.QValue)).ToList();

//If we have more than one "best" action pick one at random.
//Otherwise random max is exclsuive so it will pick the best action.
action = experiences[Utility.RandomInteger(0, experiences.Count)].Action;
```

A.2 Q-Learning Learn

```
//If the action results in a terminal state, then the Q Value is equal to the reward.
if (!terminalState)
{
        //Get Q values of every action in the new state.
        List<Experience<S>> qValues = memory.GetExperiencesForState(newState, possibleActions);

        double maxQ = 0;

        //Get the maximum Q value of the Q values in the new state.
        if (qValues.Count > 0)
        {
                maxQ = qValues.Max(x => x.QValue);
        }

        //Calculate new QValue for this experience.
        experience.QValue = experience.QValue + alpha * ((reward + gamma * maxQ) - experience.QValue);
}
else
{
        //Q value is equal to the reward in terminal states.
        experience.QValue = reward;
}
```

A.3 Genetic Algorithm Fitness Function

```
public int FitnessSum(double[] genes)
{
        //Fitness of individual is calculated as the sum
        //of all of its genes.

        return (int)genes.Sum();
}
```

A.4 Sigmoid Function

```
private static double SigmoidFunction(double value)
{
        return 1 / (1 + Math.Exp(-value));
}
```

A.5 Sigmoid Function Derivative

```
private static double SigmoidDerivative(double value)
{
        return value * (1 - value);
}
```

A.6 Hyperbolic Tangent Function

```
private static double TanHFunction(double value)
{
        return Math.Tanh(value);
}
```

A.7 Hyperbolic Tangent Derivative

```
private static double TanHDerivative(double value)
{
        return 1 - Math.Pow(value, 2);
}
```

A.8 Output Neuron Error Calculation

Executed for every output neuron in the output layer, 'desiredOutput' is a double array that holds the target value for each output neuron.

```
HiddenNeuron outputNeuron = neuralNetwork.OutputLayer.Neurons[i] as HiddenNeuron;

//Calculate the error delta and store it in the neuron.
outputNeuron.Error = (desiredOutput[i] - outputNeuron.Value) * derivitiveFunction(outputNeuron.Value);
```

A.9 Hidden Neuron Error Calculation

Executed for every hidden neuron in each hidden layer.

```
//Sum up the weighted errors.
double weightedErrorTotals = 0.0;

//For each weight from this neuron to each neuron in the next layer, sum up the
//weighted error of the connected neuron.
foreach(HiddenNeuron connectedNeuron in neuralNetwork.Layers[i+1].Neurons)
{
        //Get all weights connected from this neuron to the next layer.
        Synapse connection = connectedNeuron.Connections.First(x => x.Origin == neuron);

        //Acumulate the weighted error total.
        weightedErrorTotals += connection.Weight * connectedNeuron.Error;
}

//Error of the hidden neuron is the total weighted error multipled by the
//activation function's derivative of the neuron's value.
neuron.Error = weightedErrorTotals * derivitiveFunction(neuron.Value);
```

A.10 Neuron Error Adjustment

```
public void AdjustForError()
{
        foreach(Synapse synapse in connections)
        {
                //The new weights value is calculated as weight + learning rate * error * input.
                synapse.Weight = synapse.Weight + NeuralNetwork.learningRate
                                * error * synapse.Origin.Value;
        }
}
```

A.11 RISK Battle Damage Calculation

```
//Subject to change.
private const double BASE_DAMAGE_PER_TROOP = 0.8;

private static int CalculateDamage(int attackingTroops, int defendingTroops)
{
        double damage = (BASE_DAMAGE_PER_TROOP * attackingTroops);

        //Only RISK deal in absolutes.
        return (int)Math.Round(damage);
}
```

A.12 Genetic Algorithm Tic Tac Toe Fitness Function

```
\begin{lstlisting}[basicstyle=\tiny]
public int FitnessSum(double[] genes)
{
        //Set the weights of the player's neural network.
        networkPlayer.SetWeights(genes);

        for(int i = 0; i < 1000; i++)
        {
                TicTacToeGame game = new TicTacToeGame(networkPlayer, randomPlayer);
                game.Start();
        }

        //Fitness is the number of wins out of 1000.
        return networkPlayer.Wins;
}
```

A.13 Minimax Q-Learning Choose Action

```
int action;

//Random epsilon for random exploration of other actions.
if (Utility.RandomDouble() < epsilon)
{
        action = possibleActions[Utility.RandomInteger(0, possibleActions.Count)];
}
else
{
        //Pick the best experience based on Q values, if no actions
        //exist in memory then just pick a random action.
        //Get the experiences for this state.
        List<Experience<S>> experiences = memory.GetExperiencesForState(state, possibleActions);

        if (maximiser)
        {
                //Get the experiences with the highest Q value.
                experiences = experiences.Where(x => x.QValue == experiences.Max(m => m.QValue)).ToList();
        }
        else
        {
                //Get the experiences with the lowest Q value.
                experiences = experiences.Where(x => x.QValue == experiences.Min(m => m.QValue)).ToList();
        }

        //Pick an action at random from the most ideal actions.
        action = experiences[Utility.RandomInteger(0, experiences.Count)].Action;
}

previousState = state;
previousAction = action;
return action;
```

A.14 Minimax Q-Learning Learn

```
//Get experience for this state, action pair.
Experience<S> experience = memory.GetExperience(state, action);

//Train the experience.
experience.Reward = reward;
double expectedQ = 0.0;


if (!terminalState)
{
List<Experience<S>> qValues = memory.GetExperiencesForState(resultState, possibleActions);

        if (maximiser)
        {
                expectedQ = reward + (gamma * qValues.Min(x => x.QValue));
        }
        else
        {
                expectedQ = reward + (gamma * qValues.Max(x => x.QValue));
        }
}
else
{
        //Q value is the reward for terminal states.
        expectedQ = reward;
}

double change = alpha * (expectedQ - experience.QValue);
experience.QValue += change;
```