NEATStone: Helping a Neuroevolved Agent to Learn Hearthstone

by Elliott Bowman

Submitted to The University of Nottingham
in January 2023
in partial fulfilment of the conditions for the award of the degree of
Master of Science in Computer Science (AI)

I declare that this dissertation is all my own work, except as indicated in the text

Contents

Abstract

Strategy and playstyles in AI agents is still an unanswered research question. Many AI agents in commercial games are specifically designed to fufill certain roles, and with varying degrees of complexity in their behaviours, often unable to perform actions that are outside of it's constraints. Academic attempts have shown that agents can be trained to play in certain ways, but are typically trained via pre-defined training data or heuristics. This thesis attempts to answer this question via Neuroevolution in the digital card game Hearthstone, to see whether evolved neural network agents can both be successful agents while also forming a strategy that fits the deck they use. Making usage of a Python NEAT framework and Hearthstone Simulator, the final evolved agents are shown to both be consistently successful at beating pre-designed greedy scoring agents, and show distinct differences across a range of in-game metrics, that clearly define that each agent has differing playstyles.

Keywords: Neuroevolution, Hearthstone, NEAT, Card Games

Acknowledgements

I would firstly like to thank my supervisor, Alexander Turner, who provided me with the opportunity of coming under your wing, even when you could've declined. I'm grateful for the opportunity to take on such an interesting problem in an area I have great interest in. I would also like to thank my family for being around for when things got extremely out of hand !

1: Introduction

1.1 Motivation

Artificial Intelligence (AI) in the context of video games has been a major area of contention for both developers and users. Users who want engaging and believable AI characters and opponents, and developers who aim to meet those expectations, and the technical and design challenges that come along with them. Many commercial implementations of AI vary wildly depending on the genre, ranging from basic state machines, to tree searching and neural networks. How a video game AI agent acts within a game also varies, depending on the type of behaviours that the developer wants it to have: some games may have agents that fit specific roles, such as ranged attackers, or ones that flee from danger. Whereas other games may have agents that score their current situation (and next action) on a number of heuristics, providing emergent responses to the player.

Promoting strategy in an AI agent is a unique developer challenge. There are a variety of reasons for why this is the case, such as performance constraints, development time, and how engaging it would be for the player. Other factors include the game itself, such as actions to take, and how complex a state within the game can be.

The video game Hearthstone: Heroes of Warcraft (Hearthstone) is a discrete, imperfect information game where the state in a match becomes increasingly large in size, approximated at around 2.85×10^{51} (da Silva and Goes, 2018). Hearthstone has been a popular platform for artificial intelligence research, predominantly due to its very wide state space, combined with a consistent level of stochasticity to player actions. There is also a large open source community providing a range of simulation software that gives a way for researchers to create their own agents and test them in low amounts of time.

There are a number of different agent implementations for Hearthstone, with many different approaches. As seen in the Hearthstone AI Competition (Dockhorn and Mostaghim, 2019), a majority of the participants across a number of years make use of various state/value evaluation algorithms, such as Min-Max or Monte Carlo Tree Search (MCTS), or Evolutionary Algorithms such as Genetic Algorithms. Many of these algorithms are specialised, such that they are designed around a particular deck or player strategy, typically through hand-designed heuristics that encourage certain actions. What has not been seen here (or across the literature after a preliminary glance) is an agent that makes use of Neuroevolution to control its actions and overall strategy.

1.2 Aims and Objectives

With the introductory glance into the problem area and the motivation, we can define the aim of this project to be to research, design and implement a Neuroevolution controlled AI agent for Hearthstone. This agent should be able to learn to play both the game itself, and an ‘intended’ (i.e. as defined by the cards in the deck, and the developer of the game) strategy based on the deck it is given.

To achieve this aim, the following objectives have been identified:

- Conduct an extensive literature review of Neuroevolution and its underlying components, AI in games and how strategy is formed, as well as Hearthstone itself.
- Develop a program that will handle the connection between a neuroevolution agent and the simulator, alongside a number of agents to test the solution against.
- Conduct testing and evaluation of the agent after the entire generation process.
- Evaluate the outcome of training, to determine if the agent can play well.
- Evaluate the outcome of testing, to determine whether the agent can implement the intended strategy.
- Discuss any issues or shortcomings with the implementation, and outline future pathways for improvement.

1.3 Overview

This dissertation will be broken down into 6 chapters; firstly, we will provide a brief introduction into the problem space and how it will be tackled. Second, will be a literature review of the various intersecting topics that make up the previously described problem, with the goal of understanding the underlying technologies, algorithms and problem space. We will also look at AI strategy in games, and how we can attempt to measure it.

Chapter 3 will detail the main methodology chosen. Here, we will outline the choices made in regards to Neuroevolution, such as network configurations, parameters, and how the state is represented. It will also discuss the implementation of Fitness, and how it transforms into a playable strategy.

Chapter 4 will show the results of the agents after having being trained after a number of generations, and will also detail the results of testing the agent against a number of different AI opponents, analysing various in game statistics that should determine whether the agent picks up on a strategy.

We will then summarise the project in Chapter 5, before discussing future prospects in Chapter 6.

2: Literature Review

2.1 Aim of Review

This chapter will present an introductory view into the problem spaces of this thesis, in order to give context to the work being undertaken. Initially some background on Hearthstone will be given, providing context for the problem space itself, how strategy forms within it, and how it can provide an interesting use-case for this thesis' problem area. Then, an exploration of Artificial Intelligence implementations in video games and how they can promote and/or form strategies, with further exploration into AI agents in Hearthstone itself. Following this, an overview of Neuroevolution and its underlying fundamental components: Genetic Algorithms and Artifical Neural Networks, before delving into Neuroevolution in more detail. Finally, we will move onto Neuroevolution implementations in other games, to compare how others function.

2.2 Hearthstone: Heroes of Warcraft

Hearthstone: Heroes of Warcraft is a Digital Collectible Card Game (DCCG) developed and published by Blizzard Entertainment (Blizzard Entertainment, 2014). The game is a turn based card game played between two players, with the end goal of reducing the opposing player's health to zero, by way of using various cards. Players can choose between ten distinct 'Heroes' which each have their own sets of cards and abilities. These cards are added to a deck of the player's creation, allowing for an extremely large amount of gameplay strategies.

2.2.1 Cards

Each card within Hearthstone is categorised into two distinct types: *Class* and *Neutral*. Class cards are cards that are unique to each hero, and cannot be inserted into decks for other heroes. Neutral cards are class agnostic, and can be used by any hero. Cards come with varying metrics, with some depending on the type of card. A metric that exists across all cards is *Cost*, which determines how much of the resource, *Mana*, the player has to pay in order to use the card. Cost is typically static, although some cards may be of n cost, where n is changed depending on certain conditions. Mana increases by one per turn, maximising at 10, which is typically the maximum cost a card can have.

Cards are then further categorised into four more sub-categories, which detail how they function within the game, and the types of metrics they have. *Minions* are the playable units within the game, and after being played, are capable of attacking enemy minions and the hero. Minions all share the values of *Attack* and *Health*, being how much damage the minion can deal and take, respectively. Some minions have certain special effects that occur during the game, known as *Effects*, although we will refer to them as *Keywords*. A keyword is an action within the game that occurs when the trigger

for that keyword is met. For example, the *Battlecry* keyword activates when the card is played from the hand. See Appendix A.1 for a full list of card keywords and their actions. Minion cards may also have a tag which determines a grouping that it exists in, as seen in Figure 2.1.

Figure 2.1: An example Minion card. 1: Mana Cost. 2: Card Name and Rarity. 3: Attack. 4: Health. 5: Keywords and/or description of card.

The next category is *Spell*, which have unique effects, ranging from dealing damage to creating new minions on the board, and may or may not have targets and one-time uses. Much like minions, spells can also have keywords attached to them, such as *Echo*, which allows the card to be played repeatedly for one turn. There are also further subcategories of spell cards; *Secret* and *Quest*, which function as 'traps' and a list of pre-conditions to activate the final action, respectively. However, these cards are not going to be used with our agent.

The third category is *Weapon*, which function similarly to minion cards, in that they have damage and durability (health) values. Weapon cards are attached to the Hero when played, allowing it to attack other enemies much like minions, with the durability decreasing by one, until it reaches zero, where it is destroyed.

The final category is *Hero*, which are a very small set of cards that when played will replace the player's hero with a new one, with a new set of health and potentially new abilities. These cards however are also not included with our agent.

2.2.2 Decks

Players in Hearthstone make usage of a deck of 30 cards, with a maximum two duplicates per card, with exceptions to legendary cards, which are restricted to one. Decks are the cornerstone of a player's strategy, which is affected by different factors.

The cost of cards in the deck determine the minimum turn that the player can play that card. For example, a deck filled with cards that cost one guarantees that the player can act on the first turn, however they will eventually run out of cards in the hand. This is a delicate balancing act, as the player needs to have a consistent increase of cost in the deck, to maximise the chances of being able to play a card each turn. This is known as the *Mana Curve*, a histogram of cards and their costs.

The makeup of cards of various types and a consistent mana curve pushes the deck towards a number of *Deck Archetypes*. These archetypes are generalised, intended playstyles that a deck follows. As defined by the developers, there are three main archetypes that a deck can fall under, with various

sub-archetypes that can form over time. These archetypes fit into a ‘rock-paper-scissors’ pattern, where decks under an archetype are more suited for beating others, and vice versa.

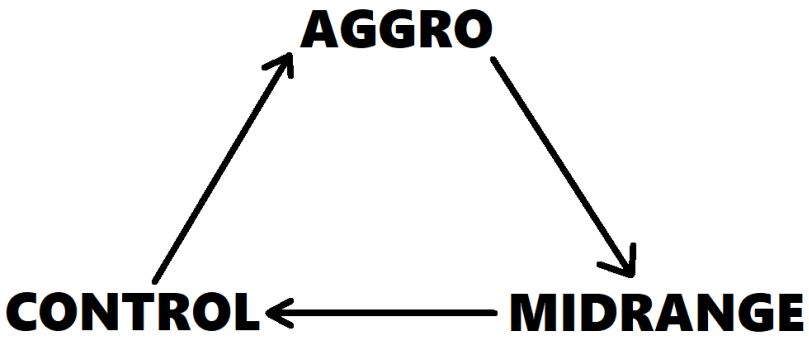


Figure 2.2: The main three archetypes in Hearthstone.

Figure 2.2 shows the three main archetypes in Hearthstone: *Aggro*, which has a focus on aggressive play, such as attacking the opposing hero directly. Decks under these archetypes typically have a low average cost, which promotes an intended strategy of attacking before the enemy can catch up.

Control is the polar opposite archetype to aggro, with the main focus being to ‘take control’ of the game and win towards the end of the match. This is typically done via taking control of the board by destroying enemy minions, and a goal of playing defensively. Decks under these archetypes usually have a mana curve that can vary between being uniformly distributed, or not at all, although typically including cards of higher costs than aggro decks. This is mainly due to the wide range of cards that can be used, and game balancing by the developer.

Midrange is the middle ground between aggro and control. Midrange decks typically focus on a Control-like defensive playstyle in the early game, before moving into an Aggro-like aggressive playstyle in the midgame. Decks under this archetype usually have a mana curve that peaks around the middle of the range of card costs.

Hearthstone’s set of cards has evolved over time, with the release of new card sets and card keywords. These ‘expansions’ bring a number of new sub-archetypes that fall under the main archetype umbrellas. These playstyles are usually created by the developer themselves through these newly added cards, although some archetypes are discovered and popularised by theorycrafting within the community. See Appendix A.2 for a full list of current sub-archetypes.

2.3 Gameplay Strategy in Artificial Intelligence

Strategy in game agents is a hard metric to define. As initially covered in Chapter 1, what constitutes as ‘strategy’ in an AI agent solely rests on what the agent use case is. An agent for a commercial game will be created for a specific purpose, whereas the state of the art for game AI research continues to expand. Togelius et al presents this as an open research question in the context of Hearthstone, suggesting this as “*a challenge that would seem to go above and beyond that of creating agents that ‘simply’ play the game to win*” (Hoover et al., 2019*).*

2.3.1 Identifying Strategy in Game AI

Video game AI has steadily evolved over the years due to various factors, such as improvements in processing power allowing for more sophisticated algorithms, and new methods devised in the scientific community.

However, much like what was mentioned in chapter 1, commercial game AI tends to focus on a particular overarching goal, an engaging experience. An AI that would have complete knowledge and accuracy of its environment would dominate even extremely skilled players, so many implementations are ‘dumbed down’, through methods such as selectable difficulty options, with many AI agents having specific roles and design decisions that intertwine with the mechanics of the

game. A well-known classical example of this being the case in commercial video game AI is in the Halo series of games, which makes usage of *Behaviour Trees* to handle agent decision making in the moment to moment to allow for 'emergent' behaviours. This is determined by a large number of interchangeable states and 'stimulus behaviours', triggers that flag certain behaviours to be executed for a short span of time after some criteria is met (Isla, D. 2005).

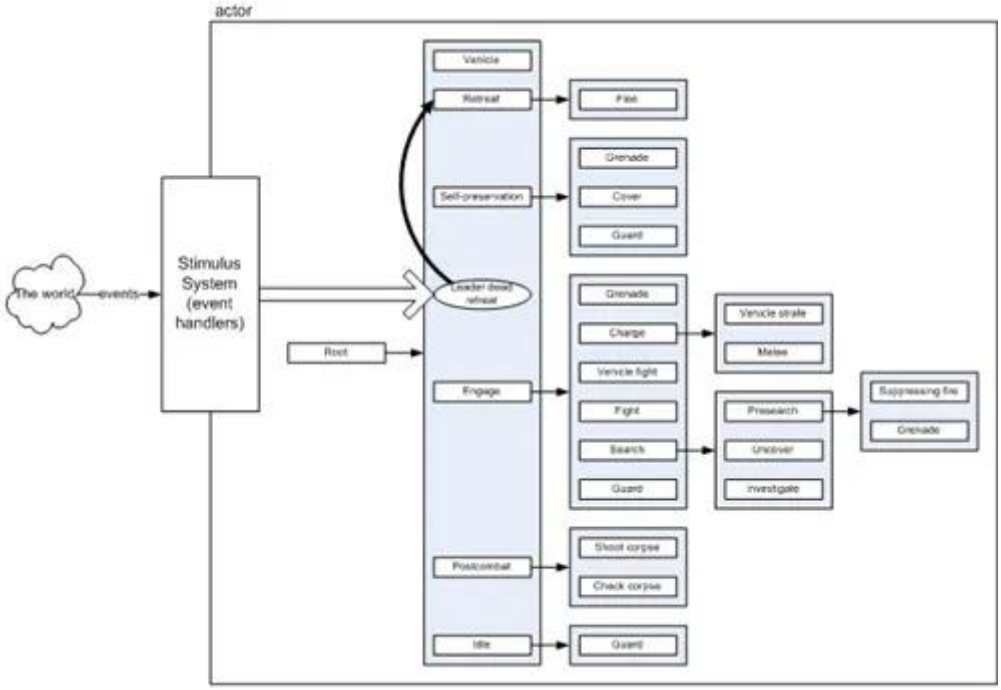


Figure 2.3: An example of how behaviour trees functioned in the game *Halo 2* (Isla, D. 2005).

Figure 2.3 shows the visualisation of this process, where a stimulus trigger, 'Leader Dead', is injected into the top level of the tree. This typically occurs when 'Grunt' characters witness leader 'Elite' characters being killed, causing them to flee for a short time. These behaviour trees and interchanging states allowed for differing enemy and ally characters to have noticeable and visible behaviours, such that players can account for them. However, they are unable to act outside of these constraints.

An example of more interesting attempts at employing strategy was in the Real-Time Strategy game Supreme Commander 2, which included agents that made usage of neural networks to control the movement and actions of its units, more specifically, when and what to attack, depending on the number of alive units. Their implementation was extremely successful, producing opponents that delivered engaging behaviours with significantly less development time required.

To allow for strategy even after networks were trained, various extra features were implemented, such as an aggression value that would inform the network that the agents units were much stronger, which would encourage more aggressive actions. Although they don't generate a discernable playstyle that may be identified and categorised easily, it allowed for a much greater variety of emergent responses to players' actions (Rabin, 2014).

Outside of commercial AI implementations, there are many instances of AI agents being developed to play games in place of a human, which presents a much harder goal to achieve over a typical game AI use case. The further challenge is to implement the capability of an agent to play in a specific way, without being pre-designed.

Holmgard et al presented a solution to generate AI agents with specific playstyles without being pre-designed in advance, coined as *Procedural Personas* (Holmgard et al., 2018). Their intended goal was to create a tool usable by human designers as a form of low cost playtesting, to identify what types of playstyles human players may form. These personas are agents that are influenced by a number of positive and negative values that are constructed in advance, and encourage evolved agents into particular actions that can then be aggregated and seperated into distinct playstyles.

2.3.2 Artificial Intelligence Agents in Hearthstone

Although the game is primarily played between two real players, there are a number of offline-only modes against AI controlled opponents. These opponents are known throughout the Hearthstone community as being very poor, typically employing basic strategies, or the same strategy repeatedly, with most of the difficulty being derived from the opponent being given 'cheats' such as powerful cards or specific bonuses, or from adding handicaps to the player. Hearthstone developers described their implementation in detail, noting that the AI makes use of a simple scoring system, where cards are either scored by its keyword(s), or by its performance in combat. This scoring system does not look ahead at future states, due to the AI being ran on a server instead of the client, meaning performance is the main priority. This means that the complexity of their AI is limited in what strategies they are able to apply, with any specific, more robust playstyles being hand crafted through adjusting various heuristics, such as choosing targets or holding onto cards (Schwab, 2014).

There is one in-game capability for an AI opponent to use a custom deck of the player's choosing, through a 'Boss' encounter in one of the offline-only gamemodes. However, there are no usable statistics to be able to judge how well this AI plays, with only anecdotal evidence of the AI not playing certain cards at all.

As explained in Chapter 1, there are a wide range of agent implementations within the Hearthstone community, with the majority using user created simulators of the game. There are also a number of different use cases within the context of the game, ranging from game-playing, deck creation and player modelling.

2.4 Genetic Algorithms

Genetic Algorithms (GA) falls under the umbrella of 'Evolutionary Computation', a grouping of optimisation algorithms that are inspired by biological evolution. First introduced by John Henry Holland (Holland, 1992), GA attempt to mimic natural selection from nature, guiding a population of solutions towards a desired target.

A typical GA starts with an initial number of solutions, known as *Individuals*, and a pre-defined function that determines how well that individual meets the desired criteria, known as *Fitness*. After each individual in the population has had its quality determined by its fitness, a new population of individuals is created.

2.4.1 Selection, Crossover, Mutation

After a population has been evaluated, individuals are recombined through various methods, known as *Selection, Crossover, and Mutation,* which fall under an overall process known as *Reproduction*. Selection methods firstly determine how an individual is chosen. *Fitness Proportionate Selection* is a subtype of methods where each individual's fitness is assigned a probability value, which is usually calculated as a percentage of the generations fitness total. One such example is through *Roulette Wheel Selection*, as shown in Figure 2.4.

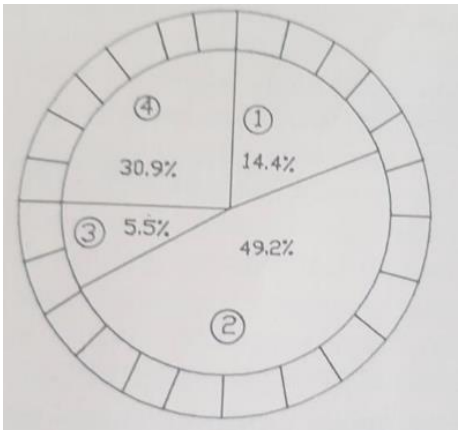


Figure 2.4: A visualisation of Roulette Wheel selection, which sorts probability of parent selection based on fitness percentages (Goldberg, 1989, p. 11).

Selection methods that do not make use of proportionate values are ones such as *Tournament Selection*, where individuals are compared against each other in 'tournaments' of arbitrary size, and an arbitrary number of winners are selected, typically via the highest fitness.

Crossover is performed after selection according to some arbitrary probability, where 'parents' are recombined with eachother, producing 'offspring' individuals that inherit some aspects of each parent, with the intented goal of pushing the evolution process into further exploration of better performing individuals.

Mutation is typically performed based upon a 'mutation rate', a user defined real number that determines whether an individual should have an arbitrary number of aspects of it changed in some way, which is selected via a probability.

Both Mutation and Crossover are a crucial part of the evolution process, as they help combat the potential problems of hitting local maxima or minima, the highest and lowest points within a range of values, respectively. As Figure 2.5 shows, mutation allows a new individual to 'break' out of the local optimum.

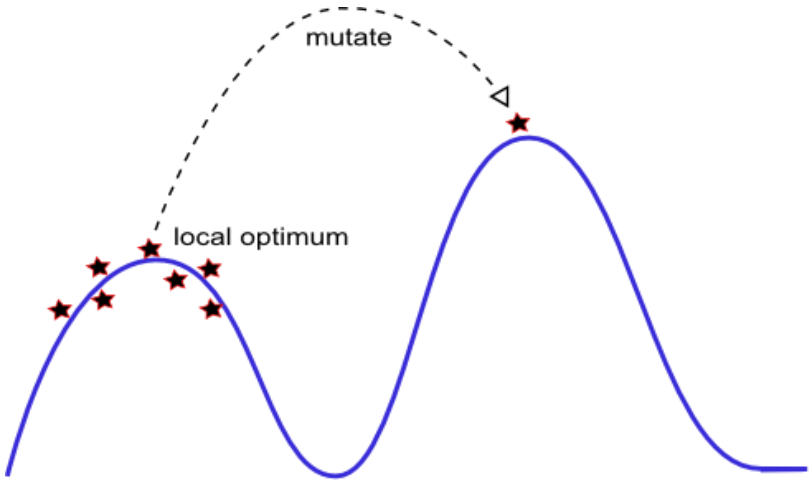


Figure 2.5: Example visualisation of local maxima, with mutation helping to escape this premature convergence (Li, 2016).

These local stopping points are what is also known as *Stagnation*, and typically occurs when the implemented genetic operators are not functioning correctly, or when a desired fitness criterion does not have the potential to reach an optimal solution, and it believes it is converging on an optimal

solution, when it is instead suboptimal. There are a number of ways to combat this, such as *fitness scaling*, where individual fitness is slowly scaled up or down over time, making poor individuals be invalidated completely, and encouraging further exploration from early individuals that may have high fitness that worsens over time (Goldberg, 1989, p. 74).

The size of the population as well as the number of total generations can also have an effect on the outcome of the generation process. For problems with large search spaces, the total population size can become a detriment into finding the global optimum solution, noting that when fitness is calculated that could lead to a local minima or maxima, a larger population size will give a greater chance of these stagnant individuals being chosen for reproduction, increasing generation times or ruining results completely (Chen et al. 2012).

2.5 Artificial Neural Networks

Artifical Neural Networks (ANN) are a subset of machine learning models that "*are inspired by the early models of sensory processing by the brain*" (Krogh, 2008).

2.5.1 Network Architecture Configuration

ANN are comprised of a number of *neurons*, a function that recieves one or more inputs x and scales each input with a weight w. There may also be an input that has no weight, called a *bias* b. The summation of inputs is passed to a non-linear function called an *activation function*. This activation function outputs a computed value, which if passes an arbitrary threshold, 'fires' the value to another neuron. Neurons are able to be represented mathematically, as shown in Equation 2.1.

$$y = f(\sum_{i=1}^{n} w_i x_i + b)$$

Equation 2.1: The generalised neuron equation for activation.

A full network is typically a number of interconnected neurons, seperated into layers – Input, Hidden and Output. The direction in which data flows through these layers is typically from Input to Output, meaning it is *Feedforward*, where data propagates forward only. Input nodes are 'sensors' that are connected to all subsequent nodes in the next layer and receive values from the task the network is performing. Hidden nodes perform further calculations with the activation function on the received input, before eventually reaching the Output nodes, which outputs a number of values.

2.5.2 Activation Functions

As described above, values that move between the input layer to hidden layers are inserted into an activation function, that determines whether that node propagates forward. There are a number of differing activation functions, that are more suited to certain tasks than others, and which scale their thresholds in different ranges. *Sigmoid* is a common function that scales its threshold between 0 and 1, forming an S shaped curve. Sigmoid has seen wide usage in ANN contexts over the years, but has steadily been phased out by other functions due to issues such as Sigmoids computationally intensive exponential operations.

Another activation function is *tanh*, also known as hyperbolic tangent, which functions similarly to sigmoid in that it forms an S shaped curve, however tanh scales its threshold from -1 to 1, allowing for more tight groups of activation thresholds.

Both sigmoid and tanh suffer from the *Vanishing Gradient* problem, which occurs with sufficiently large or small inputs that create small derivatives during *Backpropagation*. After receiving an ouput, there is usually a measure of minimizing the error of the output by evaluating its accuracy, known as the *Mean Squared Error* (MSE). This backpropagation to the input layer is performed to allow the network to eventually converge on the most accurate weights for its nodes, increasing its accuracy. However, if these weights become too small, initial layers will be updated less effectively, impacting

overall accuracy. This issue commonly occurs in large networks with multiple hidden layers, however.

There are a number of ways to combat this problem, with one such solution being the *ReLU* activation function. ReLU, or Rectified Linear Activation Unit, functions linearly for any input greater than zero, while functions non-linearly when inputs are below zero, as any output below zero is outputted as zero. ReLU has effectively become the standard activation function in recent years, mainly due to its implementation simplicity and its computational efficiency.

```
if input > 0:
    return input
else:
    return 0
```

Figure 2.10: Pseudocode Implementation of ReLU

2.5.3 Recurrent Networks

Recurrent networks are networks where outputs from nodes can form cycles, propogating its output value into itself or other nodes. This structure allows features from previous timesteps to further influence the next output.

Recurrent networks have historically had issues during backpropagation, suffering from Vanishing Gradient problems as well as *Exploding Gradient* problems, which is the opposite effect of vanishing gradients, where weight updates are large and uncontrolled. An example of how this can occur is with the ReLU function, which is able to output large input values directly.

2.6 Neuroevolution

Neuroevolution is the "*artificial evolution of neural networks using genetic algorithms"* (Stanley and Miikkulainen, 2002), and, much like GA, neuroevolution is inspired by natural processes, this time being the evolution of biological nervous systems. Unlike other machine learning methods such as supervised learning, which require a set of data for the network to initially work from, Neuroevolution allows for learning even when training data isn't available. The intention of neuroevolution is to be as generalised as possible, making it suitable for noisy problem spaces where the reward landscape from performing a set of actions is sparse, such as game playing and robotics (Lehman and Miikulainen, 2013).

The typical neuroevolution process follows the same general methodology as GA, where there is a population of individuals that are evolved over time to fit some desired fitness criteria. These individuals are then repopulated into a new population through the previously described Selection, Crossover and Mutation methods, where the weights of nodes act as individuals to be recombined. Each genotype is the encoded representation of a network, i.e. their nodes, connections, weights etc. Depending on the implementation and desired outcome, Neuroevolution can modify any desired number of features in a network, such as the weights of each node in any layer, and also the structure of the network itself, such as removing or adding new nodes and connections. The latter fall under a sub-category of algorithms known as TWEANNs (Topology and Weight Evolving Artificial Neural Networks).

TWEANN based algorithms have seen a growing popularity in recent years, as there are many instances of dynamically shaped network architectures performing at or even greater than typical reinforcement learning methods.

2.6.1 NeuroEvolution of Augmenting Topologies

Neuroevolution of Augmenting Topologies (NEAT) is a particular algorithm under the Neuroevolution banner, that was formed due to at the time issues with speed and accuracy of learning, and human time spent designing the most efficient network structures (Stanley and Miikkulainen, 2002). How NEAT differed from the state-of-the-art methods at the time was the proposal of evolving

networks via their weights and structure in small increments, starting from a minimally sized network that slowly expands into a more efficient solution, without having a contrived fitness function to measure complexity (Stanley and Miikkulainen, 2002).

NEAT also came with additional evolutionary-based functions, *Speciation*, the process of splitting populations into 'species', which is achieved via explicit *fitness sharing*, defined as the distance in fitness values between individuals, which is then subtracted from each nearby individuals' fitness total (Goldberg, 1989, p. 191-192). This sharing groups individuals together that share 'genetic similarity' and can be represented by the following equation:

$$f_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} s(d(x_i, x_j))}$$

Equation 2.2: The generalised neuron equation for activation.

Where the shared fitness of an individual within a group of individuals is its own fitness value $f(x_i)$ divided by the summation of the distances from fitness value $f(x_j)$.

This grouping of solutions allows for protection of innovative individuals from being removed from the overall process too early by competing against stronger, older individuals, allowing for increased explorative measures. Another important function in the Speciation process is *Historical Marking*, in which mutated nodes are categorised based on when they were created, allowing for a trackable structure that stays compatible even when new offspring are created. Any nodes that aren't compatible in an offspring structure are declared as *disjointed* or *excess*, which are used to calculate whether a new species needs to be created (Stanley and Miikkulainen, 2002).

2.7 Neuroevolution in Games

There is a distinct lack of research that delve into the usage of Neuroevolution in Hearthstone. However, there are a wide range of other problem spaces where neuroevolution has been used many times, and for great effect.

Since its inception in 2002, there have been many usages of neuroevolution in a variety of different video game contexts, with various different use cases. Togelius and Risi were able to define the state of the art of neuroevolution in games, including the grouping of potential 'roles' a particular developed algorithm can take (Risi and Togelius, 2017).

One such use case is procedural content generation, the '*algorithmic creation of game content with limited or indirect user input*' (Togelius, Shaker and Nelson, 2016, p1), where NEAT was used to control the creation of weapons available to the player in a science-fiction, space shooter game called Galactic Arms Race. Each new weapon a player would acquire would be created based upon the playstyle of the player, where the fitness of the current weapon being used, while unused ones would decrease over time. This fitness decay would reduce the probability of new weapons generating with the same qualities of those unused weapons, allowing for better generation of preferred ones (Hastings, Guha, and Stanley, 2009).

Another use case, which is possibly the most popular, is State/Action Evaluation, the estimation of the 'state' (the game and all of its features) and how they can affect the next action to be taken. Typical historical usage of neuroevolution in this context is by being used alongside some other algorithm such as MCTS, where the evolved network is used to evaluate the quality of the current state, and any future states before scoring an action.

Direct Action Selection is the next identified use case, where an evolved network chooses an action to take directly. This is essentially a modified version of State/Action evaluation, where the network is used indirectly to choose an option. Direct Action Selection is typically used when looking at future states is either infeasible or impossible, such as being too performance intensive, or when states become much too large (Risi and Togelius, 2017).

NERO (NeuroEvolving Robotic Operatives) was a successful attempt at combining Neuroevolution with a real-time strategy game environment, to explore ways in which to make AI agents more immersive and emergent in their behaviours. To perform this, they developed a derivation of NEAT called rtNEAT (real-time), which uses a continuous evaluation of fitness of each individual, which in this case is each member of a group of robots that fight against each other.

Because there is no fitness calculation after some set amount of time, fitness values are adjusted via fitness sharing and speciation, and are calculated after a dynamically generated amount of time which varies depending on the size of the population. Any poor performing units are simply removed from the game and replaced with a newly generated offspring that is created from reproduction.

To be able to allow the units to form a particular strategy, in-game sliders are used by human controllers to determine how their units operate, such as when to seek cover, or when to move in closer to attack.

3: Methodology

3.1 Preliminary Work

There are a number of prerequisite tasks to perform before an agent can be evolved. As we are dealing with two separate pieces of software, they both must be analysed in their functionality in order to facilitate an efficient method of cooperation with eachother. The initial representation of our network architecture, and various evolution parameters must also be decided upon. We must also choose a particular archetype that our agent will learn, as discussed in Chapter 2.2.2.

3.1.1 Fireplace

To help save development time on having to create a reimplementation of a complicated game, we are going to make use of an existing simulator that is designed for research problems such as this thesis.

Fireplace is an open-source re-implementation of Hearthstone (Fireplace git) written in Python. Fireplace is intended as a testing ground for AI agents to make usage of specific cards or strategies. It is not the only one in existence, but it does provide a high performance framework of the game.

Simulation of a game follows the rules of the original, whereby players are given a deck and hero, and take alternating turns against eachother. Fireplace makes use of a Domain-Specific Language (DSL) that define sequences of actions and the order they are played in. Cards are made up of an ID which is parsed from XML files that store card functionalities.

There are a number of identified problems that we will need to address before the entire process can begin:

- The software itself does not come with any pre-created AI agents, except for a random agent, meaning some simple score based agents will need to be developed.
- There is no way to gather game-related statistics built-in to the software (in an easy to use way), so this will also have to be implemented.
- Finally, Fireplace uses exceptions to handle important in-game events, such as when a card chooses an invalid target, or when an action is attempted when the game is over. These exceptions typically interrupt the running of the program completely, so we will want to ignore these wherever possible.

3.1.2 Hero and Deck Choice

As discussed in Chapter 2.3, there are a wide range of hero and card combinations to choose from, each with a certain strategy behind them. In order to accurately judge whether a completed agent can adhere to a certain playstyle, we will be using two different agents with two different decks and Heroes. With this in mind, we will be using simple decks that follows the *Aggro* archetype for the Warrior Hero character, alongside the *Midrange* archetype for the Hunter Hero.

The Aggro deck focusses on flooding the board with a high number of low cost minions. Some cards are included that have keywords that activate from the presence of other minions of the *Pirate* type, having equipped weapons, or when damage is taken.

Figure 3.1: Axe Flinger

Figure 3.1 shows the *Axe Flinger* card, which is a high threat card to the opponent. Its high health, alongside its activated effect, means that unless the opponent has a means to remove it in one action, they will receive a constant amount of damage. Figure 3.2 below shows the *Frothing Beserker* card, which functions in a similar way, but activates from any minion on the board being damaged, making it a priority target, as it can quickly scale to high damage values.

Figure 3.2: Frothing Beserker

The Midrange deck centers around *Beast* cards, with cards gaining or activating keywords when Beast cards are in play. There is also a higher amount of spell cards that allow for targeted damage or removal of high threat minions, and a larger number of high costing minions.

Figure 3.3: Scavenging Hyena

A main component is the ‘Scavenging Hyena’ card, which gains 2 attack and 1 health after a friendly beast minion is destroyed. Essentially, this deck has a playstyle centered around early board control to allow for the ‘setup’ of the Scavenging Hyena minion, with other fallback options if it gets destroyed. Examples include the inclusion of a number of deathrattle cards that summon more minions when destroyed, to help combat a potential aggressive opponent.

Aggro was chosen others due to the relative simplicity in overall strategy compared to the other two archetypes. Likewise, Midrange was chosen due to its playstyle having combined features of both Aggro and Control, while not leaning too heavily into the other. Each deck must also contain cards that are implemented within Fireplace, as not every card in Hearthstone is implemented. Therefore, each deck will be constructed manually. See Appendix B for their mana curves.

3.1.3 Data Analysis Methods

We will also need other software to allow us to transform and visualise our data after testing and training. As we are using Python, we will make usage of Scikit-learn (), which contains a number of feature engineering methods as well as visualisation capability, alongside Pandas () and Matplotlib ().

3.1.4 NEAT Implementation

To save time on development of a Neuroevolution agent, we will be using the Python framework NEAT-Python () to handle the evolution and network creation process. This is mainly due to the shared language between both Fireplace and NEAT-python, making development easier. The original implementation of NEAT and its various descendants have a range of parameters that determine how the evolution process behaves, with a number of considerations to take into account when choosing the values of the parameters available.

Table 3.1: Parameters within a NEAT-Python Configuration File within the NEAT section.

Parameter	***Value***
Pop_size	50
Fitness_criterion	Max
Fitness_threshold	100000

NEAT-Python networks use a configuration file that determine various behaviours in evolution and network calculation, with groups of parameters split into sections. Table 3.1 shows the most important parameters from the [NEAT] section. *Pop_size* determines the size of our population for each generation. *Fitness_criterion* and *fitness_threshold* determine how fitness in a network is calculated overall, with the threshold determining the desired criteria value. The fitness threshold is set to 100000 mainly due to the fact that it is hard to define when the agent is truly optimal, so we will instead have the evolution process last for the maximum number of generations.

Table 3.2: Parameters within the DefaultGenome Section.

Parameter	***Value***
Num_inputs	282
Num_hidden	0
Num_outputs	31
Initial_connection	Full_nodirect
Feed_forward	False
Activation_default	Tanh
Activation_options	Tanh
Activation_mutate_rate	0.0
Aggregation_default	Product
Aggregation_options	Sum, product, min, max, mean, median, maxabs
Aggregation_mutate_rate	0.1

Table 3.2 shows the most important parameters from the [DefaultGenome] section. This section holds many parameters about how a genome functions, such as how weights and biases are changed, and by how much or how often. *Num_inputs*, *num_hidden* and *num_outputs* determine the initial structure of a network, with *initial_connection* determining the initial layout of connection between nodes. As pointed out in Chapter 2.4, NEAT works well with initial network configurations that are smaller and simpler, and that are fully connected. The reasoning behind the chosen number of input and output nodes will be discussed in Chapters 3.3 and 3.4. The initial connection type chosen is *full_nodirect*, which connects each input node to each output node, with recurrent connections to itself.

The parameter *feed_forward* is a boolean determining whether the network is feedforward or recurrent. As discussed in Chapter 2.3, the choice to use a feedforward or recurrent network typically depends on the use case of the network. In our case, we have gone with a recurrent network, simply because Hearthstone works with sequential actions, with future actions being determined by past ones.

Activation_default and *activation_options* determine the starting chosen activation functions and the list of potential activation functions that can be assigned to nodes, with *activation_mutate_rate* determining the probability of it being changed. During preliminary testing of different activation functions on agent outputs, *tanh* was chosen as the sole function. Testing with the ReLU function often lead to extremely large outputs that would become unusable.

Aggregation_default, *aggregation_options* and *aggregation_mutate_rate* all function similarly to the ones described above, which control how the aggregation calculation (the summation of values in a node) functions and changes.

The final three sections, [DefaultSpeciesSet], [DefaultStagnation] and [DefaultReproduction], determine how the evolution process handles stagnation and reproduction of species, as discussed in chapter 2.6.1, as well as the calculation of genomic distance between species. *Max_stagnation* is set to 1 alongside *species_elitism* set to 1. These are set as our population is initialised as 50, meaning the potential for a large number of species is not very high. *Elitism* during reproduction is set to 5.

3.2 State Representation

Any state of the game needs to be represented with enough level of detail to allow for the agent to make accurate decisions.

The state is represented as a 1-dimensional vector of 282 real numbers, containing a number of various in-game metrics, such as the type and statistics of each hero (health, mana), and whether a weapon is equipped, and that weapons statistics. For active minions on both sides, they are represented as 10 values per the following metrics: whether it exists, the attack, maximum health and current health and whether it can attack. We also include values depending on the keywords that it may have: deathrattle, divine shield, taunt, stealth or silenced.

For cards, they are stored in a similar manner to minions, with 10 values corresponding to: exists in the hand, whether it's a minion, spell or weapon, attack, health, mana cost, and whether it has the keywords divine shield, deathrattle and taunt.

As Hearthstone is an imperfect information game, players can only see the other players card when they are played. Because of this, when storing values representing the opposing player, we store the length of their hand, deck, maximum mana, and number of minions on the board. We also store whether either side has board advantage.

If the player's hand or the board isn't full, we simply set all values within those ranges to be zero, and add a buffer to the vector if it does not meet the number of input nodes.

3.3 Processing Output

After receiving a state representation, the network will output a range of probability values. As noted in Chapter 3.2.3, the number of output nodes is 29, corresponding to maximum 20 possible actions, and 9 selectable targets, as shown in Table 3.3.

Table 3.3: All actions within Hearthstone corresponding to output nodes.

Node	Action/Target
1	Card Slot 1
2	Card Slot 2
3	Card Slot 3
4	Card Slot 4
5	Card Slot 5
6	Card Slot 6
7	Card Slot 7
8	Card Slot 8
9	Card Slot 9
10	Card Slot 10
11	Minion Slot 1
12	Minion Slot 2
13	Minion Slot 3
14	Minion Slot 4
15	Minion Slot 5
16	Minion Slot 6
17	Minion Slot 7
18	Hero Power
19	Attack With Hero
20	End Turn
21	Own Hero
22	Opposing Hero
23	Minion Slot 1
24	Minion Slot 2
25	Minion Slot 3
26	Minion Slot 4
27	Minion Slot 5
28	Minion Slot 6
29	Minion Slot 7

3.3.1 Action Selection and Invalid Masking

As mentioned in Chapter 2., we can categorise our neuroevolution implementation as Direct Action Selection, as we are effectively choosing the output node that gives the highest probability value. () proposed the method of heavily penalising or ignoring actions that are invalid at any state. Their work showed that invalid masking improved network outputs and learning significantly. Therefore, we will implement this functionality into our solution, which is performed after the network has given its output.

```
rows, cols = (21,2)
        actions = [[0.0 for i in range(cols)] for j in range(rows)]
        player = instance.current_player

        if player.choice:
            # get valid choices e.g. discover, choose one
            for i, c in enumerate(player.choice.cards):
                actions[20][i] = 1.0
        else:
            # add cards in hand
            for i, card in enumerate(player.hand):
                if card.is_playable():
                    if card.requires_target():
                        actions[i][0] = 1.0
                        for t, c in enumerate(card.targets):
                            actions[i][1] += 1.0 # enumerate counter e.g board slot

                    elif card.must_choose_one:
                        for choice, c in enumerate(card.choose_cards):
                            actions[i][choice] = 1.0
                    else:
                        actions[i][0] = 1.0
```

Figure 3.4: Partial view of invalid masking for agent choices.

For example, if the network outputs a value on node 11, corresponding to the leftmost minion on the board, we first check if it exists in play. If it does, we then need to check that it hasn't already attacked. This masking of actions occurs for every action, and varies depending on whether it requires a specific type of target, or whether it has been already played. The action to end the turn is never masked. Figure 3.4 denotes a partial snippet of the algorithm for action masking, which shows when the agent attempts to play a card from the hand.

Targets are also masked, which occurs after action masking. Fireplace stores targets for actions in a list, where each player has eight maximum targets (Hero, seven minions). This list of targets varies in size for certain actions, such as spell cards, which can sometimes target any place on either side of the board. Early preliminary testing with a network output shape of 16 nodes (corresponding to each side) and no masking did not work very well for this, as spells in the deck would have a higher maximum number of targets compared to minions, meaning the simulator often crashed.

With this in mind, each of the 9 target nodes correspond to both sides of the board. The chosen action determines which sides of the board we want to check for valid targets. Minions and the Hero cannot willingly attack friendly units, meaning for spell cards, we validate each target index with the corresponding node.

3.4 Fitness Function

The fitness function is the most crucial element of our setup, as it will define what in-game metrics we want the agent to make a push towards.

The fitness of an individual will be determined through a number of components, based around retrieved in-game metrics after all games have been played. The fitness function is as follows:

$$F(x) = TW - TL + w1 * SMH + w2 * SHH + w3 * BA + w4 * SMM + w5 * SHM - w6 * T$$

The first, and most obvious metric is the total number of wins; an agent probably isn't very good if it loses every single game. The second is the total number of losses.

$$SMH, SHH$$

These metrics refer to the averaged total amount of times minions and the hero attacked the enemy hero, respectively. These are averaged because the maximum damage applicable to a hero per game is most commonly 30, meaning it has a feasible maximum bound.

$$BA$$

This metric is the 'Board Advantage' of the agent, or the average amount of turns where the agent's board was larger than the opponents (the number of active minions).

$$SMM, SHM$$

These metrics refer to the averaged total amount of times minions and the hero attacked enemy minions. These are again averaged to allievate outliers per game, such as gaining a high amount in game 1, then nothing through the rest.

$$T$$

The final metric is the averaged total number of turns taken per game. The maximum amount of turns in a game is 180, although this is an almost impossible place to reach, as a player will typically start receiving fatigue damage (drawing a card from an empty deck) at around turn 25.

Based upon expert knowledge, and a few assumptions, the metrics were determined based upon how 'general' they are. That is, they can be applied to theoretically any deck, as any deck should promote (or reduce) attacking minions, heroes, staying in control, and finishing quickly or slowly. Any other metric, such as playing cards, cards of certain types/keywords, using the hero power and others become much too specific and will make the fitness function become much less generalisable.

Another point is that these differing metrics essentially turns it into a multi-objective optimisation problem. However, due to the non-deterministic qualities of the game, it's effectively infeasible for a true optimum solution to be found, making it technically unbounded.

3.4.1 Fitness Weights

The weights shown in prior are determined by the deck that the agent is using, and are used within the fitness function to determine which component is worth more over others. Determining an intended strategy from the cards in a deck alone is a difficult task, especially in a game with such a large space of wildly differing strategies, as discussed in Chapter 2.3. As previously mentioned in Chapter 3.2.1, we are using rather 'simple' decks that don't include any cards with complex strategies that go beyond what the statistics of that particular card say.

Therefore, in an attempt to deduce an archetype, we collect a number of statistics from the deck:

- The average mana cost of the deck
- The average damage (from minions and weapons)
- The total number of minions, spells and weapon cards

```
SmhWeight = 5
    ShhWeight = 5
    SmmWeight = 5
    ShmWeight = 5
    BoardAdv = 5
    Turns = 5
    results = []

    if avgCost < 3:
        # probably aggro focused
        Turns -= 4
        SmhWeight -= 3
        BoardAdv -= 1
        # high weapons
        if weaponSum > 2:
            ShhWeight -= 3
            ShmWeight -= 2

        if (abs(totalMins) - abs(totalSpells)) >= 10:
            # probs minion heavy/zoo type
            SmhWeight -= 1
            ShhWeight -= 1
            BoardAdv -= 1
            SmmWeight -= 1


        results.append(SmhWeight) #1
        results.append(ShhWeight) #1
        results.append(Turns) #1
        results.append(BoardAdv) #3
        results.append(SmmWeight) #4
        results.append(ShmWeight) #4

    elif avgCost >= 3:
        # probs midrange
        Turns -= 3
        SmhWeight -= 1
        SmmWeight -= 1
        BoardAdv -= 3
        # high weapons
        if weaponSum >= 2:
            ShhWeight -= 2
            ShmWeight -= 3

        if (abs(totalMins) - abs(totalSpells)) >= 10:
            # minion heavy/zoo type
            SmhWeight -= 2
            ShhWeight -= 1
            SmmWeight -= 1

        elif (abs(totalMins) - abs(totalSpells)) < 10:
            # probs more focus on spell use to clear
            SmhWeight -= 1
            BoardAdv -= 1
            ShmWeight -= 1
```

Figure 3.5: Partial view of algorithm to determine deck archetype

With these metrics in mind, we can naively find an archetype from a simplistic deck by using a number of differing thresholds based upon these parameters, alongside premade changes for each ranking based on these thresholds, as shown in Figure 3.5.

After determining the archetype of the deck, the weights are assigned an ‘Importance’ ranking, ranked from ‘Very high’ to ‘Very Low’, determining the range of applicable weights that can be generated for that metric. These weights will be bounded between 0.01 and 1.

Applying this function onto our agent's deck produces the following weight ranges:

Table 3.4: Initial Fitness Weight values calculated.

Warrior (Aggro)		Hunter (Midrange)	
SMH	Very High(0.9 to 1)	SMH	Medium (0.4 to 0.6)
SHH	Very High (0.9 to 1)	SHH	Medium (0.4 to 0.6)
T	Very High (0.9 to 1)	T	High (0.8 to 0.9)
BA	Medium (0.4 to 0.6)	BA	Very High (0.9 to 1)
SMM	Low (0.1 to 0.3)	SMM	Low (0.1 to 0.3)
SHM	Medium (0.4 to 0.6)	SHM	Very High (0.9 to 1)

An initial exploration into the mutation of these weights over a number of generations lead to the following assumption: if a generation were to have its weights mutated, then its fitness calculations could potentially lead it to a playstyle that differs from the 'intended' playstyle of the deck it uses, or could lead to losses of well performing individuals. Therefore, these weights will be static throughout the entire process.

3.5 Gathering Statistics

To be able to thorougly evaluate both the results of training and testing the agent, we need to be able to gather in-game statistics about every game. Some of these statistics we gather will also be used within the fitness function to calculate an individuals' fitness. Hearthstone has a number of varied features that can exist across any particular state, which, when gathered across a whole game, will help us understand how the agent behaves. Table 3.5 shows the 15 metrics that we are gathering.

Table 3.5: Game Features

Game Feature	Description
PLAYMINIONCARD	Total number of minion cards played per turn
PLAYSPELLCARD	Total number of spell cards played per turn
PLAYWEAPONCARD	Total number of weapon cards played per turn
ATKMINTOHERO	Total minion attacks to the opposing hero per turn
ATKHEROTOHERO	Total hero attacks to opposing hero per turn
ATKMINTOMIN	Total minion attacks to opposing minions per turn
ATKHEROTOMIN	Total hero attacks to opposing minions per turn
HEROPWR	Total number of times the hero power was used per game
NUMTURNS	Total number of turns taken per game
BOARDADV	Total number of turns where player had board advantage (higher number of minions)
ENEMYMINDEATHS	Total number of opposing minion deaths per turn
SELFHERODMG	Amount of Hero health lost per turn
ENEMYHERODMG	Amount of opposing Hero health lost per turn

MANASPENTPERTURN	Total mana spent per turn
MANAREMAININGPERTURN	Total mana remaining per turn

As discussed in chapter 2.3, the composition of the deck used and the overall archetype the deck adheres to will help differentiate between strategies. For example, the *ATKMINTOHERO* and *ATKHEROTOHERO* metrics are more likely going to be higher within aggro decks that promote attacking the opposing hero more often, whereas a Control deck is more likely to have a higher number of *ATKMINTOMIN* and *PLAYSPELLCARD* values to promote playing defensively. Another facet of these metrics is that many of them affect one another. For example, the higher the *NUMTURNS* metric is, the higher the *MANASPENTPERGAME* metric will become. These metrics will help in finding if the developed agents are able to show strategies that are different from eachother.

As described in chapter 3.2.1, Fireplace does not come equipped with any verbose logging functionality that is easily parsed into a set of different values. Because of this, the metrics described above will be instead accumulated per turn and calculated after each game is played. We can easily acquire them due to how the agent functions with specific action outputs, as described in Chapter 3.3. They are then stored within a double nested Dictionary data structure, where each nested dictionary corresponds to each of the 15 metrics. Inside each dictionary is a vector of N length, where N is the number of turns that game, with each metric per turn stored within. Any per game metrics are calculated afterwards, such as the number of turns.

3.6 Agent Opponents

As mentioned briefly in Chapter 3.2.1, Fireplace also does not come with any premade AI agents to test against, only including a random agent that performs any valid actions at random, with any random target, including its own Hero.

To allow for a better training environment for our agent, we need to have opponents that are stronger than a completely random one. In order to save development time and complexity, these agents are instead modelled off of the described implementations from Hearthstone itself, as mentioned in Chapter 2.2. Each agent is a relatively simple greedy scoring agent, that is, it will rate its next action based upon several in-game factors at that state.

The warrior opponent makes use of the same deck as the Neuroevolution agent, so it will therefore have an Aggro favoured strategy. As described in previous chapters, the typical way in which an aggro deck plays are by playing very aggressively in the early turns, attacking enemy minions occasionally. However, this agent will function slightly differently.

```
for t in targs:
                curScore = -5
                tHealth = t.health
                tDmg = t.atk

                if isinstance(t, Hero):
                    curScore += 1000
                    # if board adv do that first
                    if len(p1.field) < len(p1.opponent.field):
                        curScore -= 1000

                if minDmg == tHealth:
                    curScore += 5
                if tDmg >= minHealth:
                    curScore += -2

                if isinstance(t, Minion):
                    if t.taunt:
                        curScore += 9999
                    elif t.has_deathrattle:
                        curScore += -2
                    elif t.has_inspire:
                        curScore += 1

            scores.append([curScore, t, minion])
```

Figure 3.7: Partial view of algorithm for scoring of minion attacks

As Figure 3.7 shows, the agent will first focus on 'efficient trading' of minions, the process of choosing the target with minimal loss to the player, which if unable to do so (or a win is available), resume attacking the hero. Cards are selected based on whichever minion costs the cheapest, with spell and weapon cards being played afterwards. Finally, if no other cards or minions are available, the agent will use the hero power, and attack with a weapon if necessary. This scoring does not take any card keywords into account, the reasoning being we don't want to have an opponent that dominates the neuroevolved agent in earlier generations, which would negatively affect fitness.

```
if self.currentTurnCount < self.AggroTallyMinimum:
            return

        totalDmg = 0
        manaLeft = p1.mana
        hasBeast = False

        for t in p1.field:
            totalDmg += t.atk
            if t.race == Race.BEAST:
                hasBeast = True
        for c in p1.hand:
            if c.cost <= manaLeft and c.is_playable() and
            c.type == CardType.SPELL and c.__str__() != "Multi-Shot":
                if c.__str__() == "On the Hunt":
                    totalDmg += 1
                elif c.__str__() == "Kill Command":
                    if hasBeast:
                        totalDmg += 5
                    else:
                        totalDmg += 3
        if p1.weapon is not None and p1.hero.can_attack():
            totalDmg += p1.weapon.damage
        if p1.hero.power.is_usable() and manaLeft >= 2:
            totalDmg += 2

        if p2.hero.health - totalDmg <= 0:
            self.IsAggroing = True
```

Figure 3.8: Partial view of algorithm to determine when to become aggressive

The next scoring agent again is a mirror to the Hunter agent, in that it is based upon the Midrange archetype, and focusses on using Beast cards. As shown in Figure 3.8, the hunter agent functions different to the warrior agent in that it will 'switch' into aggressive actions, by keeping a running tally of turns. After a certain number of turns, the agent will check whether it has a strong board advantage over the opponent, in which case it will begin playing more aggressively. This also affects the scoring of actions, where if playing defensively, the agent will score taunts, spells and minion trading over others. This switching can occur multiple times depending on whether the agent has board advantage or not.

The random agent will use a simple deck that only loosely fits into the midrange archetype, although it is not expected to win many games, simply being used as an extra benchmark to test the neuroevolved solution.

3.7 Evaluation Methods

We will evaluate our agents in two stages: the end state after generation is complete, and the results after going head-to-head against all handcrafted agents described previously. As mentioned in the initial objectives, the resulting agent needs to be able to play the game to a reasonable level, which could potentially have an impact on the agent being capable of implementing a specific strategy. In order to determine whether the agent can implement the strategy in question will be determined via analysis of its game features collected during its matches against the other agents.

3.7.1 Evaluation of Fitness Function

To judge the quality of our designed fitness calculation method, we will be performing three separate tests:

- The first, using the Aggro deck, will consist of a population size of 50, and a generation count of 50. Each individual will play 15 games each against the pre-designed Aggro agent. The fitness weights will be the initial values as described in chapter 3.4.2.

The first test is to determine if the generation of weights based on the composition of the deck has any impact on fitness improvements, under the assumption that a longer generation process/training games will allow for more potential of higher fitness candidates. As mentioned in Chapter 3.4.2, the generation of weights does not take into account more specific features such as card groupings or card synergies, meaning a deck based around a very specific playstyle can't be determined. However, as our decks do not follow this strategy, we can only assume whether this will impact overall fitness.

It also tests whether the weight ranges themselves will have any impact on fitness, and whether it converges on a local maxima or minima.

- The second, using the Aggro deck, will consist of a population size of 50, and a generation count of 50. Each individual will play 15 games against the pre-designed Midrange agent.
- The third, using the Midrange deck, will have a population of 50, and a generation count of 50, with individuals playing 30 games against the pre-designed Aggro agent.

The following two tests will vary depending on the outcome of the first. If the first test points towards a lack of exploration, then the weight ranges will be increased from 0.01 to 5 in the second test, and 0.01 and 10 in the third test.

All three tests will finally be performed to determine whether the agent can win games reasonably well, including against the agents it has not trained against in later testing.

3.7.2 Evaluation of Strategy

To determine whether a generated agent can implement its given strategy, we will first be pitting the best agents from the previous tests against the other scoring agents, as shown in Table 3.6.

Table 3.6: Proposed Evaluation Methods and Matchups

Player 1	Playstyle	Player 2	Playstyle	Num Games per Match
Aggro Agent (test 1)	Aggro	Aggro Score	Aggro	500
Aggro Agent (test 1)	Aggro	Random	Random	500
Aggro Agent (test 1)	Aggro	Midrange Score	Midrange	500
Aggro Agent (test 2)	Aggro	Aggro Score	Aggro	500
Aggro Agent (test 2)	Aggro	Random	Random	500
Aggro Agent (test 2)	Aggro	Midrange Score	Midrange	500
Midrange Agent (test 3)	Midrange	Aggro Score	Aggro	500
Midrange Agent (test 3)	Midrange	Random	Random	500
Midrange Agent (test 3)	Midrange	Midrange Score	Midrange	500

Each agent takes part in 3000 matches in total, with game statistics acquired after each match. Under the assumption that the agents will have evolved to play in a certain way, the statistics obtained should show differences in playstyles, particularly between the aggro and midrange agents.

We will then perform Principle Component Analysis (PCA) via the scikit-learn and pandas libraries on the retrieved statistics, to be able to visualise the variations between each agent playstyle.

4: Results and Analysis

4.1 Training Process

As shown in Chapter 3.7.1, multiple tests were performed as a part of the generation process.

4.1.1 Aggro Agent 1

The first aggro agent was completed after around 2 hours, with a sharp increase in average fitness that quickly deteriorates into a plateau after 20 generations, showing no signs of improvement.

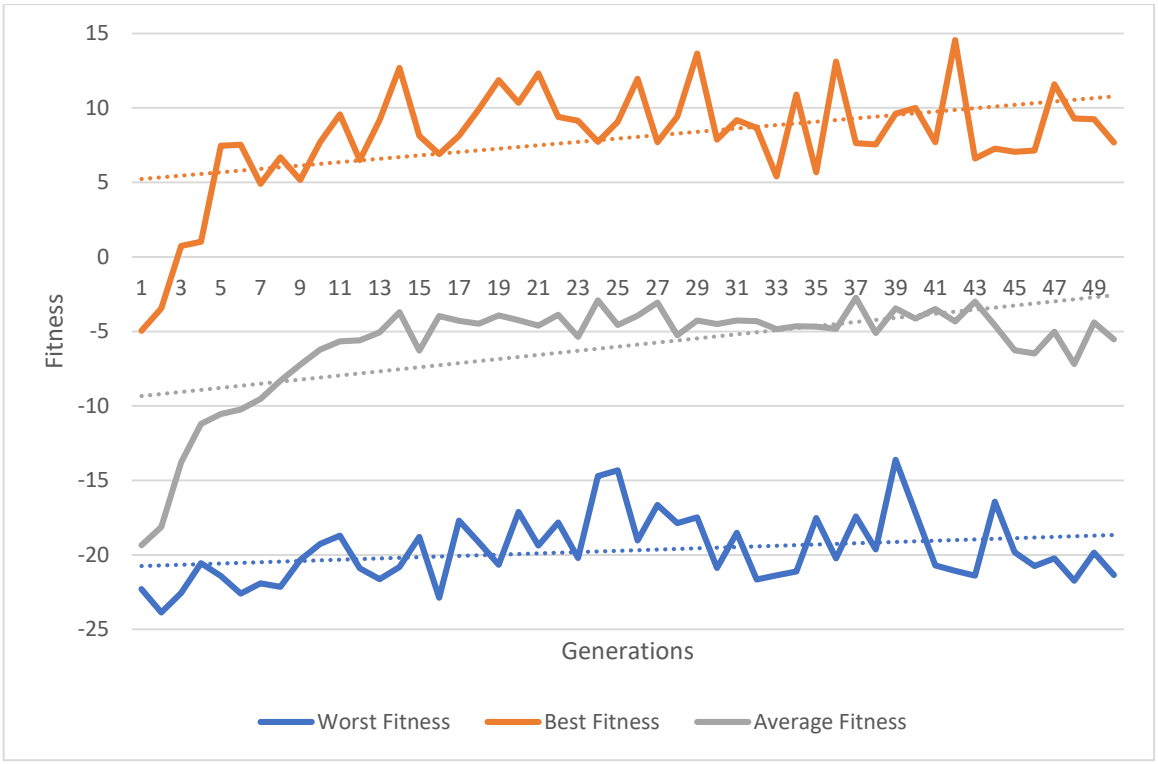


Figure 4.1: Fitness per generation after 50 generations for Aggro Agent 1.

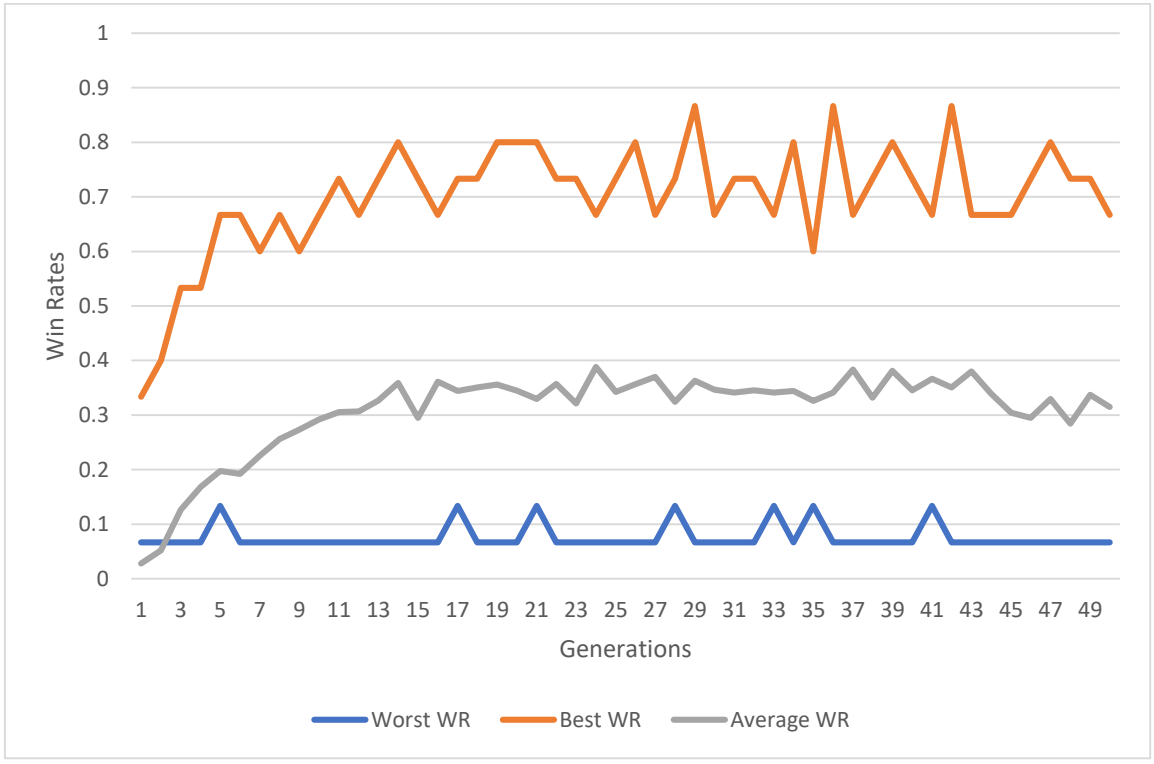


Figure 4.2: Win rates per generation after 50 generations with 15 games per individual for Aggro Agent 2.

The highest fitness achieved was 15, with the lowest being -23.8, and the highest average of -2.7. Although these fitness values are not impressive, win rates also increased rapidly before converging

into a plateau, with the highest win rates being 80% across a number of generations and averages converging to around 30% very quickly.

The notable fitness plateau could potentially be a factor of the relatively ‘short’ generation process, with only 15 games per individual and 50 generations, which would mean that an individual does not get the opportunity to improve its fitness. This is also shown with the consistent average win rates sitting around 30%, meaning that network improvements via the genetic operators were minimal or nonexistant.

In terms of being a capable agent, the highest win rates were consistently above 50% after generation 5, meaning that the agent is able to play well against what is essentially a mirror match, although not consistently. As each agent has the exact same deck and hero, the assumed outcome would be a win rate of 50%, but this could also have been affected by the scoring methods of the Aggro agent, which, as described in chapter 3.6, focusses on destroying minions before attacking the hero. This differing playstyle from what normally constitutes an aggro playstyle may explain the consistent average win rates, as Aggro playstyles typically suffer against opponents that prioritise destroying minions.

Overall, the most probable reasoning behind the lack of improving fitness is more than likely due to the small weight ranges, which will only provide small changes in fitness, slowing down exploration.

4.1.2 Aggro Agent 2

The second aggro agent test was completed after around 3 hours, and again features a similar rapid initial increase in average fitness that plateaus, although there is much less fluctuation than the previous.

Figure 4.3: Fitness values per generation for Aggro Agent 2.

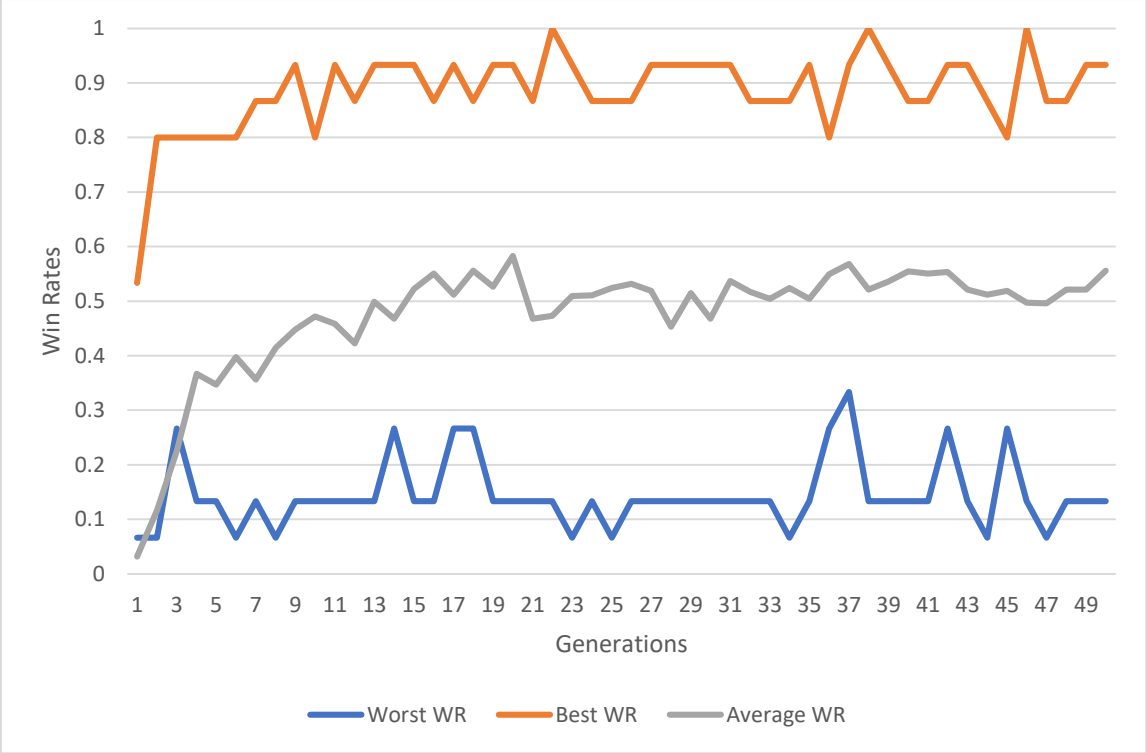


Figure 4.4: Winrates per generation with 15 games per individual for Aggro Agent 2.

The highest fitness achieved was 34.3, with the lowest being -98.7, and the highest average being 7.2. Much like the previous test, there is also a sharp increase in average win rates that eventually plateaus to an average around 55%, with the highest win rates consistently being above 80% for a large number of generations.

As mentioned in Chapter 2, Aggro playstyles are more suited to win against Midrange playstyles, which bolsters the improved win rate percentages over the previous test. This 'intended' matchup outcome is usually due to Midrange decks requiring 'setup' time for some particular strategy or set of cards, which leaves them vulnerable to aggressive play in earlier turns. Our designed deck was designed for this playstyle in mind, centering around boosting a particular minion, Scavenging Hyena, as mentioned in Chapter 3. An example way to counter this minion would be to ensure that you have early board control to remove weak enemy minions that they could easily sacrifice against your own minions.

However, it could be argued that because Aggro is an inherent 'risky' playstyle, it may be harder for individuals who prioritise being aggressive to guarantee consistent wins.

With regards to the previous test, it is clearly visible that increasing the value ranges of the fitness weights increases fitness results directly, although it is unclear whether that has any actual effect on the win rate performance of the network itself.

```
Mean genetic distance 2.486, standard deviation 0.354
Population of 50 members in 2 species:
   ID   age  size   fitness   adj fit  stag
  ====  ===  ====  =========  =======  ====
   12    4    37      5.102    0.768     0
   13    1    13      8.585    0.814     0
Total extinctions: 0
```

Figure 4.5: Final species statistics after generation 50.

Figure 4.5 above shows the final generation statistics after the 50th generation, with a total of 13 unique species generated. The mean genetic distance across species is at 2.4, with a standard deviation of 0.3, which could suggest that species are only marginally unique from one another, potentially forming local maxima. However, the species with the greatest fitness that gave us our final agent was clearly the offspring from the older species which persisted for 4 generations, meaning that there is some level of exploration of a number of network structures.

Overall, the results from both tests clearly outline that our initial network structure and configuration are capable of winning against each scoring agent consistently and with visible improvement, although not to a high level.

4.1.3 Midrange Agent

The Midrange agent completed generating after 7 hours, and unlike the previous two tests, the agent had more positive outcomes.

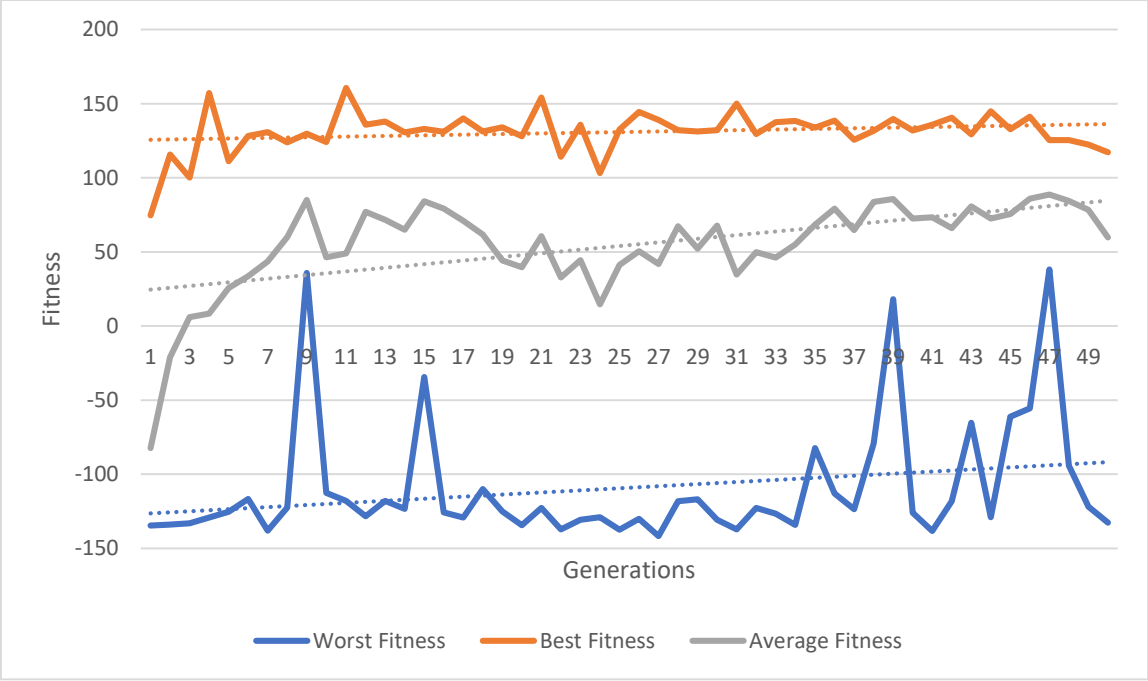


Figure 4.6: Fitness values per generation for Midrange Agent

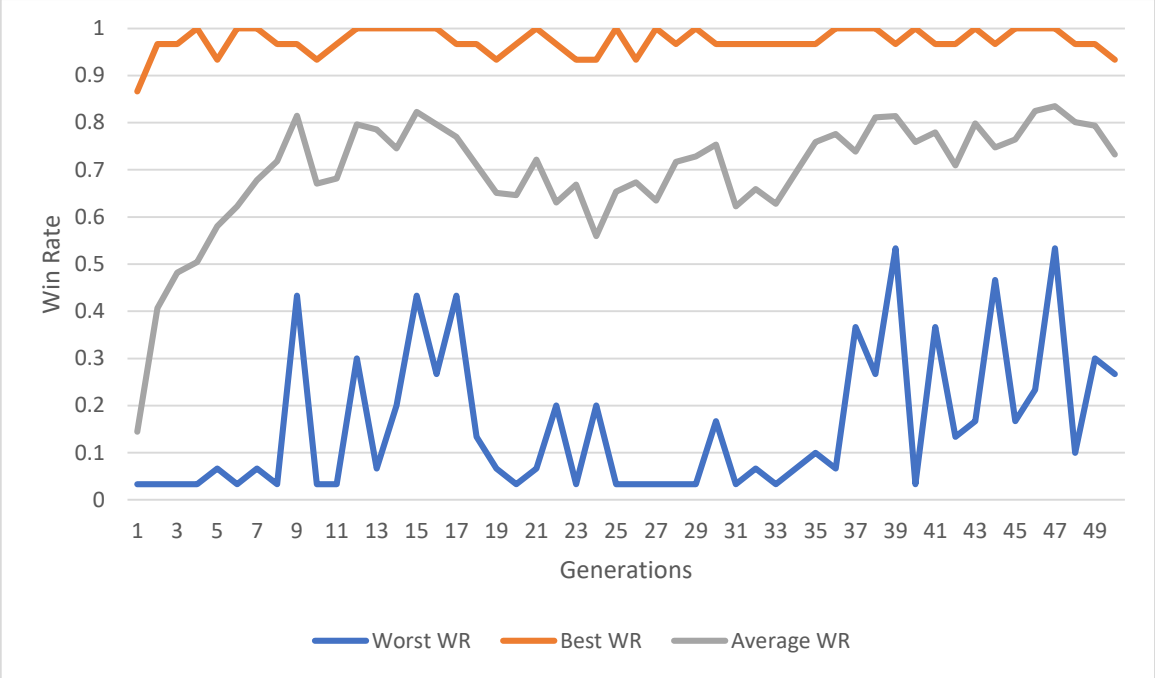


Figure 4.7: Winrates per generation with 30 games per individual for Midrange Agent

The highest fitness value achieved was 160.5, while the lowest fitness was -141.7, with the highest average being 88.6. The highest win rates barely increased at all from the initial generation, moving from above 80% to constantly above 90% to 100% for the entire process, with average win rates steadily increasing over time, with a maximum average of 83%.

It was mentioned in the previous chapter that Midrange playstyles are less suited against Aggro playstyles. Game design wise, this is clearly never a completely true statement, as it is possible for any playstyle to win against the other. This is shown in our results, where the Midrange agent has the highest average win rates against the other two agents.

4.2 Testing Process

As discussed in Chapter 3, our generated agents were pitted against the other scoring agents for a number of games, with statistics from each game gathered for analysing.

4.2.1 Discernable Strategy

One of our original objectives was to see whether an agent that is evolved over time could form a strategy that it is intended to do as according to the deck it has been given. As we have two separately generated agents that follow the Aggro playstyle, we will perform Principle Component Analysis, alongside a Biplot visualisation to determine how similar or different each agent is against the Midrange agent.

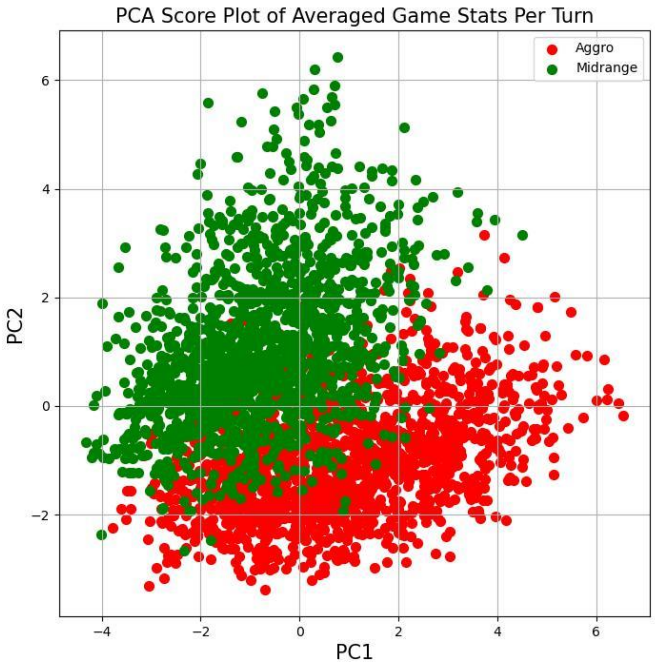


Figure 4.8: Principle Component Analysis between Aggro agent from Test 1, and Midrange agent from Test 3

As shown in figure 4.8 above, there is a clear seperation between each agent, with each dot representing the averaged 15 metrics we described in Chapter 3, across a total of 1500 games for each agent. The seperation isn't perfect as there are margins of errors visible, however, this can be attributed to a number of factors. Firstly, Midrange has facets of both Aggro and Control in their usual playstyles, such that Midrange will have Aggro-like early turns and Control-like later turns. It could also be the case of an agent having a particularly bad game, where the agent was suffering simply due to random chance of drawing cards from the deck.

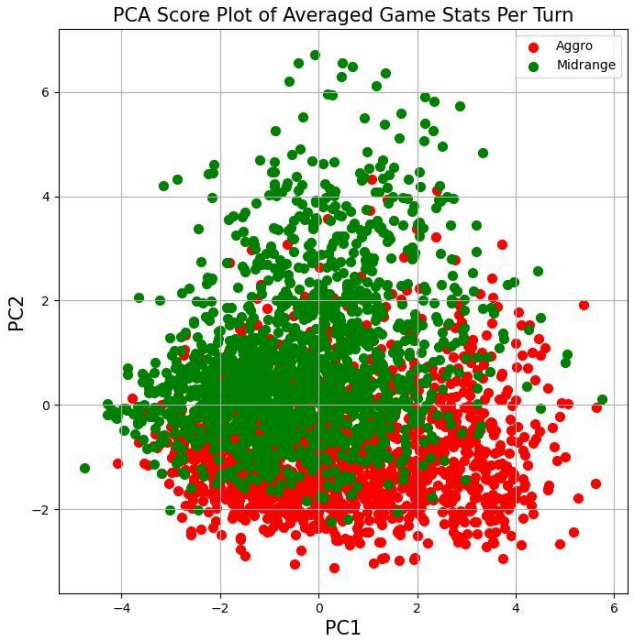


Figure 4.9: Principle Component Analysis between Aggro agent from Test 2, and Midrange agent from Test 3

Performing PCA between the agent from the second and third tests also produced similar results, however there is a lot more points that crossover between each component, which again will potentially suffer from the same problems described previously. This could potentially be due to the improved performance in win rates of the Aggro agent, alongside the high average win rates of the Midrange agent, as both were trained against eachothers playstyle.

To help visualise how these inputted features are plotted, a Biplot is also created for each PCA calculation, with each of the 15 metrics being assigned vectors which represent their load onto each of the component axes.

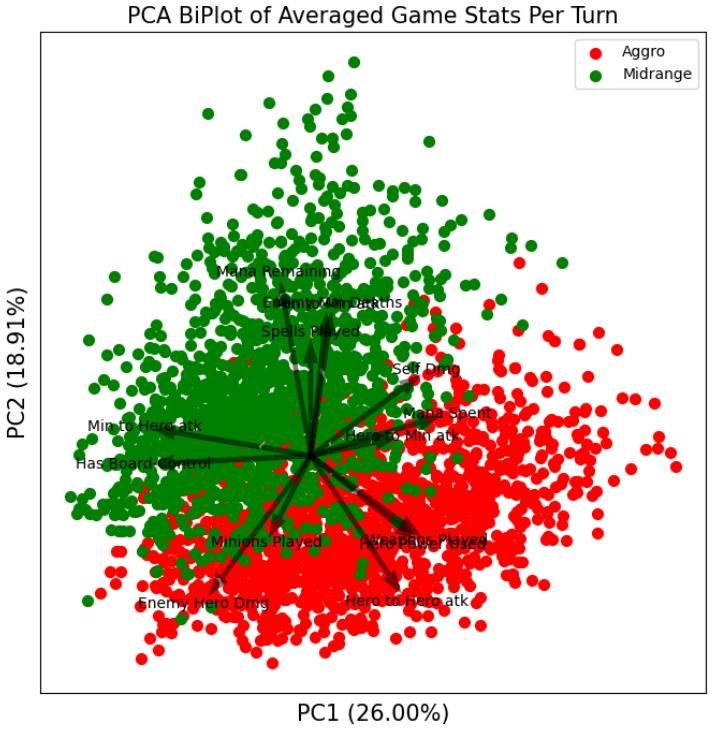


Figure 4.10: PCA with Biplot between Aggro agent from Test 1, and Midrange agent from Test 3

As viewed in Figure 4.10, the features help to show a clear seperation of actions taken that formulate a certain playstyle. Many features along the PC1 axis, which also contributes the highest to the variance in the data, are highly aggressive, with only a few instances of features from the Midrange agent heading towards this direction. Features along the PC2 axis are much less aggressive, and point more towards a defensive, Control-like playstyle.

The Hero to Hero Attacks, Weapons Played, Enemy Hero Damage, Hero to Minion Attacks, and Self Damage metrics are all mostly dominated by the Aggro playstyle, which suggests the Aggro agent prioritised using its hero to clear minions from the board instead of its own.

There are some outliers however, such as Minion to Hero Attacks, which although is an action that all playstyles would do, it is mainly dominated by the Midrange agent, with only a few instances of the Aggro agent. Overall, this potentially suggests that this agent either didn't learn that it should attack with its minions, with a few possible explanations. One is possibly due to this particular agent training against the Aggro scoring agent, which prioritised attacking other minions before attacking the hero. This would mean that the agent would learn to essentially fill the board with minions while equipping the large number of weapons it has to attack the hero directly. Another possibility is the inclusion of minions in the deck that can damage the hero without attacking, such as Leper Gnome and Axe Flinger.

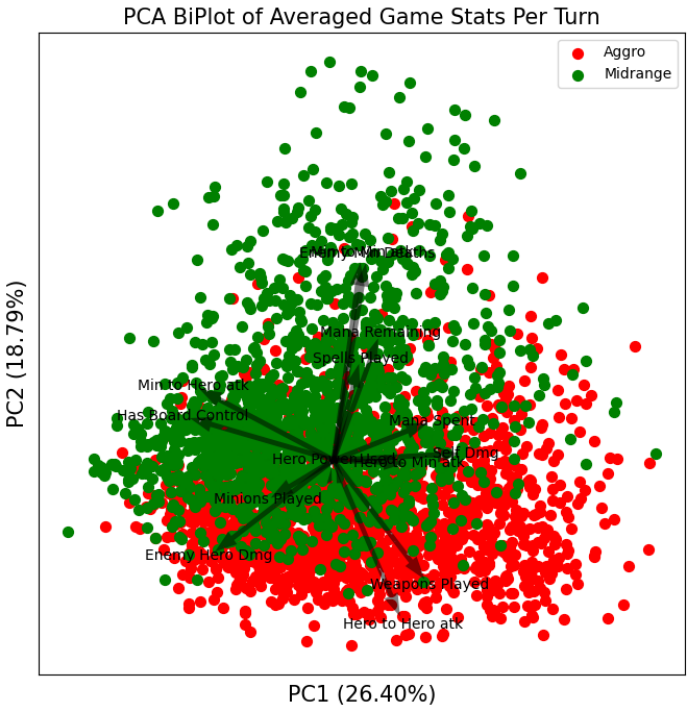


Figure 4.11: PCA with Biplot between Aggro agent from Test 2, and Midrange agent from Test 3

The Biplot between agents 2 and 3 also showed similar vector directions to the previous chart. As described earlier, there is a clear seperation between Aggro-like actions that are mostly covered by the Aggro agent. However, some outliers identified in the agent from test 1 are not as prevalent in the agent from test 2, such as Minion to Hero attacks.

The Board Control, Minion to Hero, Enemy Minion Deaths, Minion to Minion Attacks, and Spells Played are mostly represented by the Midrange agent, which makes sense when considering the deck composition itself, and how it has a focus being defensive early.

4.2.2 Game Performance

Each agent performed 500 games against the two scoring agents and the built-in random agent, for a total of 1500 games played.

Table 4.1: Win rates of trained agents vs. scripted scoring agents

Vs.	Random	Aggro Score	Midrange Score
Aggro Agent 1	97.2%	87%	74.6%
Aggro Agent 2	89.8%	83%	67%
Midrange Agent	98.2%	91.4%	86.8%

As visible in Table 4.1, the Midrange agent is clearly the best performing agent, with the lowest win rate percentage of 86%. This is more than likely a product of the increased generation time, and the deck promoting a less risky and aggressive playstyle.

Interestingly, the Aggro agent from test 2 performed the worst out of all three agents, with a lowest score of 67% against the Midrange scoring agent. This is surprising considering this agent was trained against the Midrange agent, and also had a higher average win rate than the Aggro agent from test 1, which ran for the same number of generations and training games. As seen in Figure 4.9, the metrics

gained from this agents' matches are sparser when compared to the other Aggro agent, which could suggest that that agent either lost a well performing individual and/or species during generation, or had entered a stagnant state and converged upon a local maxima solution. Another explanation could've been its training opponent, as the Midrange scoring agent is much stronger than the Aggro scoring agent, meaning genetic exploration would've taken longer. This agent may have performed better if it ran for a longer number of generations.

Overall, it is clear, alongside the results from training, that each bot can play the game well while also winning games very consisently.

5: Conclusion

5.1 Success of Objectives

Our original objectives were defined in Chapter 1.2.

- An extensive literature review was performed within Chapter 2, consisting of an overview of the problem space, alongside Neuroevolution and its underlying components, followed by an analysis of how AI in games have shown strategy, either intentionally or not, in both commercial and non-commercial contexts, and how these strategies can be identified. Finally, an exploration into implementations of Neuroevolution in academic contexts was performed, to allow for a comparison against our own.
- A Python program was developed that handles the connection between the Fireplace simulator and the NEAT-Python framework, and providing an interface for a created network to perform actions in the game. A number of agents, both generated and pre-designed, were also developed.
- Testing was performed both before and after training of each agent, with initial training outcomes revealing that the generated agents are capable of winning games consistently, and play with visibly different playstyles.

5.2 Success of Aim

The initial aim of this thesis, as outlined in Chapter 1.2, was:

To design and implement a Neuroevolution controlled AI agent for Hearthstone. This agent should be able to learn to play both the game itself, and an intended strategy based on the deck it is given.

Three separate Neuroevolution agents were successfully developed, and as outlined in Chapters 3 and 4, these agents are able to learn to play the game, and play the game with differing playstyles.

6: Future Work

6.1 A More Thorough Fitness Function

One problem area that needs improvement is the fitness function itself, and how it is calculated, and what parts of the agents' performance affects the overall fitness. Changing the values of weights seemed to not particularly have any impact on overall performance of the agents at all, with the agent from test 2 having worse playtesting performance than the agent from test 1, that made usage of smaller weight range values. Additionally, the calculation of these weights based upon the deck is too naïve, and should be much more thorough in terms of cards in the deck, alongside the types of cards and potential synergies. Additional components in the fitness calculations may also cause agents to become more refined in their playstyles.

6.2 Combination with Other Methods

Our implementation does not have any kind of looking ahead of future states at all, meaning it can only make its decisions based on its current inputted state. Although it is the case that our implementation uses Recurrent networks, it is unclear whether that would have made any

improvement over a solution that used a feedforward network. There are numerous successful examples of other implementations that make usage of search algorithms to sift through a number of future states, using the network to instead score the best next state.

6.3 Larger Number of Stronger Opponents

There is potential evidence as seen in Chapters 3 and 4, that the designed scoring agents may have affected the generated behaviours of the agents, and that there is potential for agents to deviate from an intended playstyle into one that is easily suited to hand-designed opponents. As seen in Chapter 2, agents that are more complex and have potential to show emergent behaviours act for better opponents, and there is an interest to see whether the quality of the opponents will affect the overall capability of the agents.

References

Chen, T. et al. (2012) *A Large population size can be unhelpful in evolutionary algorithms, Theoretical Computer Science 436*, pp. 54-70. Available at: doi: 10.1016/j.tcs.2011.02.016

Dockhorn, A. and Mostaghim, S. (2019) *Introducing the Hearthstone-AI Competition*, Available at: https://arxiv.org/abs/1906.04238 [Accessed 15 August 2022].

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning. 1st ed*. Boston [u.a.]: Addison-Wesley.

Hastings, E. Guha, R. Stanley, K. (2009). *Evolving content in the Galactic Arms Race video game, 2009 IEEE Symposium on Computational Intelligence and Games*, pp. 241-248, Available at: doi: 10.1109/CIG.2009.5286468.

Holland, J. (1992). *Adaptation in natural and artificial systems.* Cambridge, Mass.: MIT Press.

Holmgard, C., Green, M.C., Liapis, A. and Togelius, J. (2018). Automated Playtesting with Procedural Personas with Evolved Heuristics. *IEEE Transactions on Games*, pp.1–10. doi:10.1109/tg.2018.2808198.

Hoover, A.K., Togelius, J., Lee, S. and de Mesentier Silva, F. (2019). *The Many AI Challenges of Hearthstone*. *KI - Künstliche Intelligenz*. doi:10.1007/s13218-019-00615-z.

Isla, D. (2005). *GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI*. [online: weblog]. Available at: https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai [Accessed 01 January 2023]

Da Silva, A. and Goes, L. (2018). *HearthBot: An Autonomous Agent Based on Fuzzy ART Adaptive Neural Networks for the Digital Collectible Card Game Hearthstone,* IEEE Transactions on Games, 10(2), pp. 170-181, Available at: https://doi.org/10.1109/TCIAIG.2017.2743347 [Accessed 30 July 2022]

Krogh, A. (2008). *What are artificial neural networks*? *Nature Biotechnology*, 26(2), pp.195–197. doi:10.1038/nbt1386.

Li, S. (2016). *Local Maxima*. [Image]. Available at: http://blog.justsophie.com/teaching-cars-to-drive-with-genetic-algorithms/ [Accessed 20 August 2022]

Lehman, J. and Miikkulainen, R. (2013) *Neuroevolution*, Scholarpedia [online], 8(6), pp. 30977. Available at: http://dx.doi.org/10.4249/scholarpedia.30977 [Accessed 23 August 2022]

Schwab, B. (2014). *AI Summit, AI Postmortem: Hearthstone,* Game Developers Conference [online], 17-24 March 2014, Moscone Center, USA. [Accessed 20 August 2022]

Stanley, K. and Miikkulainen, R. (2002). *Evolving Neural Networks through Augmenting Topologies,* Evolutionary Computation [online], 10(2): pp. 99-127. Available at https://doi.org/10.1162/106365602320169811 [Accessed 14 August 2022]

Togelius, J., Shaker, N. and Nelson, M. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer. ISBN 978-3-319-42714-0.

Rabin, S. (2014). *Game AI pro: collected wisdom of game AI professionals*. Boca Raton: Crc Press/Taylor & Francis Group, pp 367.

Risi, S. and Togelius, J. (2017). *Neuroevolution in Games: State of the Art and Open Challenges*. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1), pp.25–41. doi:10.1109/tciaig.2015.2494596.

Rodrigues, N., Silva, S. and Vanneschi, L. (2020). *A Study of Generalization and Fitness Landscapes for Neuroevolution*. IEEE Access, 8, pp.108216-108234. Available at: https://doi.org/10.1109/ACCESS.2020.3001505 [Accessed 16 August 2022].

Appendix A: Hearthstone Mechanics

A.1 Card Keywords in Hearthstone

Keyword	**Description**
Adapt	Choose from one of 3 possible upgrades to X minion/self.
Battlecry	Triggers when card is played.
Can't Attack	Unable to attack, but deal damage when receiving an attack.
Can't Attack Heroes	Cannot attack enemy hero.
Cast Spell	Casts X spell for the player.
Cast When Drawn	Triggers when card is drawn from deck.
Colossal	Enters board split into X separate minions
Corrupt	Gains X new effects if card of higher cost is played when in hand.
Copy	Copies X card in hand/deck, or X minion on board.
Charge	Can attack as soon as it's played.
Choose One	Choose between 2 unique actions.
Choose Twice	Choose twice between X number of actions.
Combo	Triggers action(s) based on X amount of actions played beforehand.
Counter	Counters X card/effect/keyword/action when it occurs.
Draw Card	Draws X cards from deck.
Deathrattle	Triggers when minion dies in play.
Discover	Choose between either: 3 randomly created cards, or 3 cards of X type.
Divine Shield	Causes first instance of damage to be nullified.
Dredge	Look at bottom 3 cards and put one on top of deck.
Dormant	Minion is unusable and un-targetable until X conditions are met.
Death Knight	Generated from other keywords
Deal Damage	Deals X damage to a character.
Destroy	Destroys X number of minions, secrets or weapon.
Discard	Discards X cards from hand.
Enchant	Modifies X number of metrics on X number of cards/minions.
Enrage	Triggers when minion is currently damaged.
Equip	Equips hero with stated weapon.

Echo	Adds a temporary copy of a card that is repeatedly spawned until the end of turn.
Forgetful	50% chance for minion to attack the wrong target, including friendly targets.
Frenzy	One-time trigger after surviving one instance of damage.
Freeze	Minion is unable to attack for one turn.
Honorable Kill	Trigger when minion deals exact damage to its target.
Immune	Prevents all incoming damage.
Inspire	Triggers when the hero power is played.
Infuse	Gains additional effect after X minions die in play.
Invoke	Upgrades the card 'Galakrond' if it's in the deck.
Jade Golem	Summons a 'Jade Golem' minion that increases in attack and health for each cumulative summon.
Lackey	Generated 1 cost, 1 health 1 attack minions with Battlecries.
Lifesteal	Heals owning Hero for the amount of damage dealt.
Magnetic	Combines 2 'Mech' cards together if played on the left of other minion.
Mega-Windfury	Allows 4 attacks in the same turn.
Overload	Temporarily reduces the player's total Mana by X on the next turn.
Outcast	Gains an effect when card is in right-most or left-most slot in hand.
Overkill	Triggers when dealing damage that is more than the target's health.
Poisonous	Destroys any minion that it attacks.
Quest	A set of sequential conditions that trigger an action when completed.
Reborn	Will resurrect the minion with 1 health, once.
Recruit	Summons X minions from a deck.
Rush	Allows a minion to attack only other minions when played.
Secret	A 'trap' that cannot be seen by the opponent, and triggers when a certain condition is met.
Set Attribute	Sets X attribute to X value.
Sidequest	See Quest.
Silence	Removes all keywords and attached effects from a minion.
Start of Game	Triggers action at the start of the game.

Stealth	Character cannot be targeted until it attacks.
Spellburst	Triggers action after a spell has been played.
Spell Damage	Cumulatively increases all Spell card damage by X.
Taunt	Causes all opponents to target this first.
Tradeable	Card can be shuffled and swapped with a deck for 1 Mana.
Transform	Changes target card/minion into a new one permanently.
Twinspell	Creates a single copy of the card when played.
Windfury	Allows 2 attacks in the same turn.

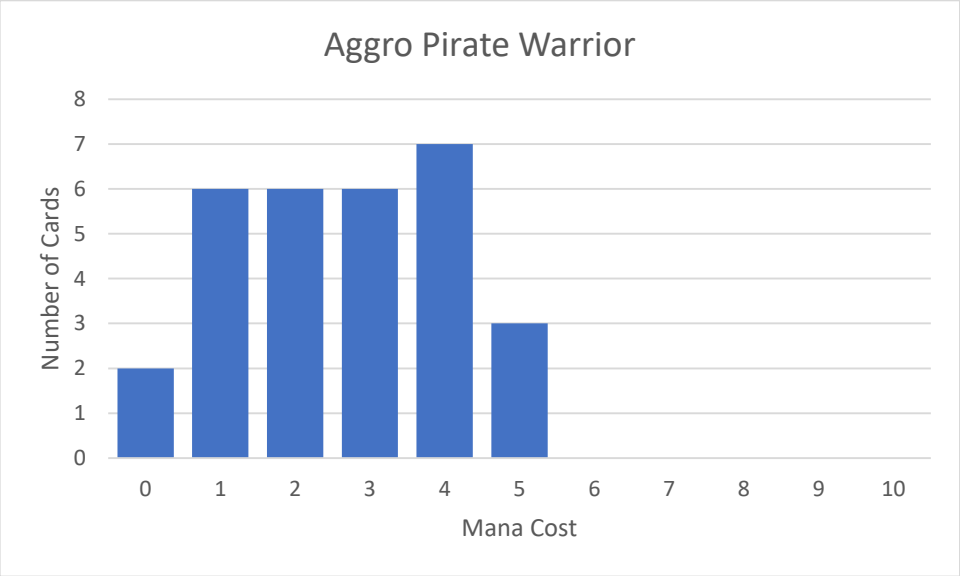
A.2 Common Deck Archetypes in Hearthstone

Archetype	Overview
Aggro/Aggressive	High focus on playing aggressively in early turns by dealing high amounts of damage as quickly as possible with low cost cards.
Control	Focus on keeping control of the game by playing defensively in earlier turns before ramping up aggressiveness in later turns with high costing cards.
Combo	Revolves around playing specific combinations of cards to make dangerous plays.
Midrange	A combination of Aggro and Control, with a strategy of early defense, midgame offense.
Minion/Card Based	A deck that has a goal of upgrading/playing certain types of cards that provide strong benefits.
Tempo	Often merges with Midrange archetype decks, has a focus on playing 'on tempo' by playing cards on every turn.
Token	Aggro-centric deck that focuses on large numbers of low-cost generated minions (tokens).
Zoo	Aggro-centric deck based upon high numbers of low/medium cost

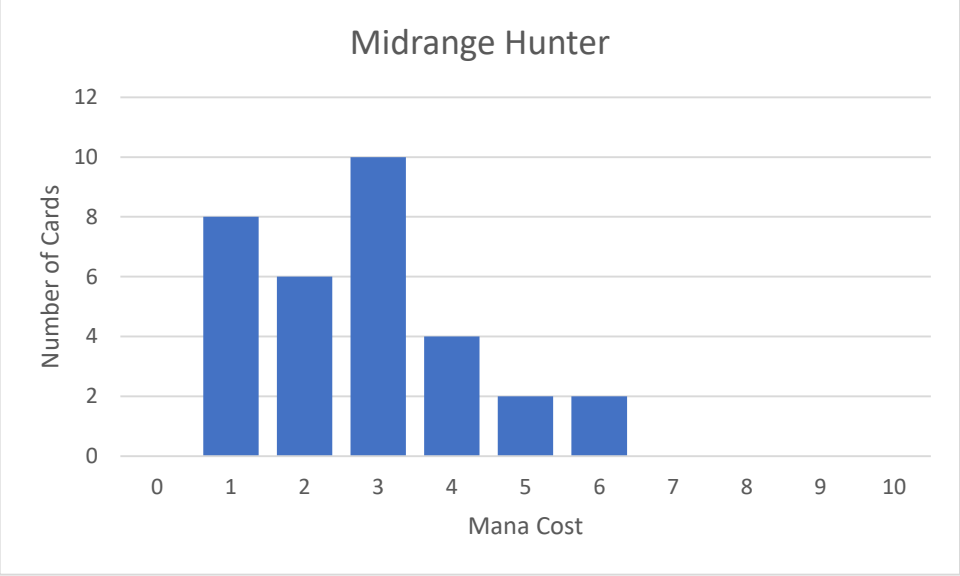
	minions, with a focus on board control.

Appendix B: Decklist Mana Curves

B.1 Mana Curve for Neurevolved and Aggro Score based Agents



B.2 Mana Curve for Midrange Score based Agent



B.3 Mana Curve for Random Agent