Playing a Game of Cards using Neuroevolution

Training players to play the card game Golf using Coevolution and NEAT

April 30, 2019

Abstract

Games are commonly used as problems for machine learning due to the human-like decisions that need to be made to be successful. Golf is a card game where players aim to reduce their score by matching cards and favouring cards that score lower.

This paper reports the process of training players to play golf using two neuroevolution algorithms, coevolution and NEAT. The players use a function approximator to evaluate the many states of the game, that have been represented in a variety of ways, and make moves based on the resulting outputs. This results of this project show that players trained under both algorithms had gained an understanding of the game by scoring less and making more matches. This project concludes that the performances of the players trained under coevolution and NEAT are comparable, however their behaviours differ. The NEAT trained players developed a greedier play-style compared to those trained by coevolution, taking longer to play rounds due to waiting for better cards to appear. Some coevolution trained players also displayed this, however not to the same extent.

It was also found that the amount of information that the player is given and how it is represented has a drastic effect on the player's performance. Players trained with minimal information outperformed those with a greater amount of information, likely due to the resulting size of its function approximator.

I certify that all material in this dissertation which is not my own work has been identified.

Contents

1 Introduction

Machine learning is an area of Artificial Intelligence which is focused around computer systems improving their performance on some task without being given any specific instructions. The first major machine learning paper was written by Arthur Samuel in 1959 which discussed the training of systems to play the game of checkers, concluding that "such learning schemes may eventually be economically feasible as applied to real-life problems" [1]. Earlier machine learning research focused heavily on the symbolic representation of knowledge, through the use of game trees and production rules. Since then, research has expanded to include newer methods focussing more on statistical models, such as neural networks, and incorporating probability theory into learning [2]. Currently, machine learning has a wide variety of applications and research fields ranging from facial recognition [3] to speech recognition [4], natural language processing [5], forecasting [6] and marketing [7].

Games are a common problem that machine learning techniques aim to solve due to the human-like decisions that are made during play. In addition, game playing provides a variety of complex challenges for a system to solve besides simply learning to play the game, including modelling opponents and capturing and replicating player/human behaviour [8]. Due to the well-defined rules of games, a consistent environment can easily be developed for the system to interact with, enabling for these games to be played via simulation. Finally, as games are commonly played by human players, comparisons can be made between the performance of the trained system and that of human players.

"A game provides a convenient vehicle for such study as contrasted with a problem taken from life, since many of the complications of detail are removed." - Arthur L. Samuel [1]

Some famous game playing systems trained through machine learning include DeepMind's AlphaGo and Atari systems. AlphaGo [9] is a system trained to play the game of Go and was able to beat the world champion in 2016. The Atari system [10] was trained to play a collection of games from the Atari console using only the pixel information on screen. Of the 43 games in the collection, the system played at or above a human level in 29 of them.

This project aims to expand upon the existing research to be able to train a player to learn to play a card game, more specifically Golf. The player will be trained under two differing neuroevolution algorithms to not only increase the likelihood of success, but to also compare the results and investigate the potential impact that the different algorithms have.
This report will first summarise the key features of the literature in this area of research. This report will then discuss the design choices made when developing this project as well as the methodology supporting the training and descriptions of the implemented algorithms. It will also discuss the experimentation process and how these initial results impacted the development of this project. This report will finally display the results collected from the training and evaluation of the players before analysing and discussing the patterns and trends of the data, followed by an evaluation of the development process and the conclusions from this project.

2 Summary of Literature Review

There exists a large catalogue of literature surrounding the concept of machine learning for game playing, ranging from board games [11] to computer games [10]. This section summarises the key features of this literature.

Card games

Whilst there exists a wide variety of different card games, each with many variations, the literature that covers machine learning and card games focuses on two types of games in particular. The first are trick-taking games, such as Hearts, where players aim to win tricks by playing the best card in the round. These provide the challenge of determining when the best time to play a card is based on the game history. The second are draw and discard games, such as Gin Rummy and Golf, where players aim to improve their hand by drawing cards and exchanging them with the cards in their hand. These provide the challenge of having to make multiple decisions in a single turn.

Methods used

The majority of the literature makes use of a neural network to evaluate the many states of the game, known as a function approximator. This prevents the need to store game state information in tables, which would be infeasible due to the vast number of states that games can be in. Most of the machine learning methods used in the literature fall under one of three categories.

The first category is reinforcement learning, where an agent interacts with an environment and receives some feedback on how "good" that action was in the form of a reward. The aim is for the agent to learn, based off the rewards it receives, what the good and bad actions are in the environment and to make the good actions more often. Types of reinforcement learning algorithms used are temporal-difference learning [12] and Q-learning [10].

The second category is game tree learning, which uses a tree to map out all the game states and the legal moves between them. A player then makes moves by searching this tree for the ideal path from its current state to an end state. A commonly used algorithm for both generating and searching a game tree is the Monte Carlo Tree Search algorithm, used famously to play the game Go [13].

Evolutionary Algorithms

The final category concerns evolutionary algorithms, which generate solutions in a manner that closely resembles biological evolution [14]. In these algorithms, new "offspring" solutions are created by inheriting and mutating aspects (genes) from existing parent solutions through genetic operators such as crossover. These newer solutions then replace the older, usually worse, solutions which results in an increase in the quality of the solutions over time.

There are many different ways of implementing these concepts for a variety of solution types, those concerning neural networks are referred to as neuroevolution algorithms [14]. These neuroevolution algorithms can be either fixed topology, where only the weights of the neural network connections are updated, or augmented topology, where the entire structure of the network can also be updated.

An example of a fixed topology approach is a hill-climbing coevolution algorithm where two solutions constantly compete against each other to improve themselves. At every update, the algorithm moves the weights of the worst solution towards those of the better solution. This approach has been used to train players to play Gin Rummy [15] as well as Backgammon [16], in the process showing it to be as good or better than some of the previously mentioned techniques.

An example of an augmented topology approach is NEAT (Neuroevolution of augmenting topologies) [17]. NEAT provides the genetic encoding that allows for the crossover of solutions, enabling for the structure of the neural networks to change. NEAT is also a population-based algorithm, meaning that it maintains and updates many solutions at the same time. Additional functionality provided by NEAT includes the dividing of solutions into species to encourage evolution and more complex structural genetic operators.

Project Specification

The main aim of this project is to be able to train players to play the game of Golf. Given the vast number of game states in any card game, players will make use of a function approximator to help make their moves. This function approximator will be updated through training based on the results of games played by the player against an opponent.
Players will be trained using both a coevolution algorithm and NEAT, which allows for a comparison of the performances of the different neuroevolution algorithms and their resulting trained players. Further, the performance and results of the coevolution algorithm can be compared with the results of the existing literature.
The performances of all trained players will need to be evaluated in order to determine how well they have learned, if at all. To do so, they will play games against opponents of varying ability, including players with no/little understanding of the game and players that are more "human-like" (i.e. heuristic players). By extracting and analysing data from these games, the performances of the trained players can be evaluated, using the aforementioned opponents as comparison points, and any behavioural patterns can be identified. All trained players will also compete against each other in a round robin tournament to not only expand the variety of opponents but also determine the best overall player.
As such, this project aims to answer these three questions:

1. Is it possible for a player to learn to play the card game Golf by evaluating game states?
2. How well have the players learned? Are neuroevolution methods useful in this scenario?
3. How do the different neuroevolution algorithms affect the trained players?

3 Golf

Golf is a card game that is centred around a draw and discard mechanic, where players improve their hand by exchanging cards from the deck or a discard pile. In the case of Golf, players aim to build a hand that scores the fewest points possible. There are many different variations of Golf, the rules below describe the version that is being used in this project.

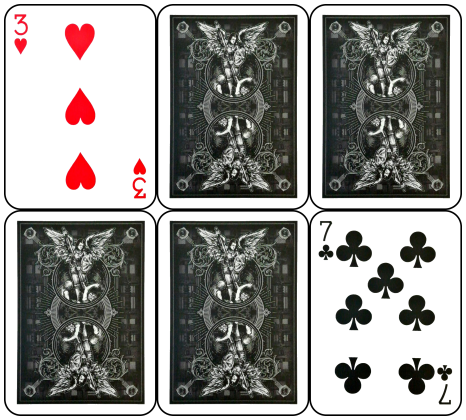

Figure 1: A starting hand of Golf

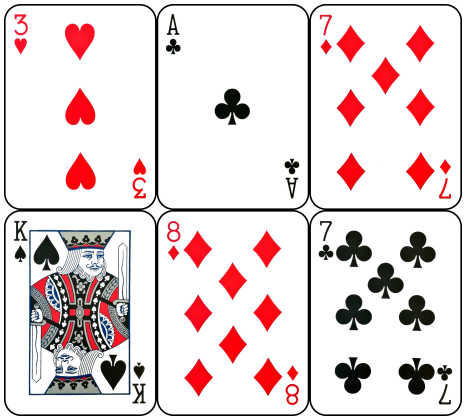

Figure 2: A final hand of Golf

3.1 Rules

Each player is dealt a hand of six cards, face-down, arranged into three columns of two. At the start of the round, each player turns two of those cards face-up, see Figure 1. An additional card is also dealt face-up to start the discard pile. Players then take turns drawing a single card either from the top of the discard pile or from the top of the deck and may either discard it, by placing it on top of the discard pile, or exchange it with a card in their hand, leaving it face-up. The exchanged card is then placed face-up

on the discard pile. Once a player has all six cards face-up (Figure 2), every other player is given a final turn before the round ends. At the end of the round, all remaining face-down cards are turned face-up.

3.2 Scoring

The hands are then scored according to the value of the cards. There are many different scoring systems that can be used for Golf, this project will use the following system:

Card	Score
Ace - 10	1 - 10 (Face Value)
Jack & Queen	10
King	0
Joker	-2

Table 1: Scoring system of Golf

In addition, if two cards in the same column have the same value (i.e. both 9s) then they both score 0 points, even if they are Jokers. Therefore, the positioning of the cards is as important to the player as obtaining low scoring cards. For example, consider the hand shown in Figure 2.
In column one, the 3 of Hearts scores 3 points and the King of Spades scores 0 points.
In column two, the Ace of Clubs scores 1 point and the 8 of Diamonds scores 8 points.
In column three, since both cards are 7s neither scores any points. Therefore, in total, this hand scores 12 points.
A game of Golf consists of nine of these rounds, with the player scoring the lowest across these rounds being declared the winner.

4 Design

4.1 Deck of Cards

The game Golf is a card game, thus one of the key aspects of the implementation will be the deck of cards. The deck has been implemented as its own class with an inner class to represent the cards.
A card has attributes to represent its suit (Clubs, Hearts, Spades, Diamonds) and value (Ace-King) numerically, where suits are represented by the numbers 1-4 and values by the numbers 1-13 (11-Jack, 12-Queen, 13-King). Additionally, Jokers can be represented by cards with a value of -1 and suits of -1 and -2, this allows for some distinction between the two Jokers. Cards also have "get" methods to retrieve their value and/or suit.
A card also has a Boolean attribute denoting whether it is hidden or visible to the player. Should one of the "get" methods be called on a "hidden" card, they will return 0 for both value and suit. This helps the player to distinguish between cards that are face-down (hidden) and those that are face-up (visible).
A deck contains a list of these cards, the order of which can be changed by using the shuffle function from python's random module.
Drawing from the deck consists simply of removing the top card from the list of cards, with an added option to keep the card hidden or make it visible, done by modifying the "hidden" Boolean attribute. This allows for cards to be drawn visibly or remain hidden from players.

4.2 Golf

A game of Golf will require a deck of cards, represented by the aforementioned class, and a discard pile, represented by a list. These are stored as variables inside the Golf class.
The project also only focusses on a two-player version of the game, this helps to simplify the development

and also reduce development time, discussed further in Section 5.3.1.
Before the start of each round, each player is assigned an empty hand, a NumPy array, and cards are dealt into these hands, hidden, via the aforementioned draw method. One additional card is also drawn, visible, and immediately added to the discard pile. At the start of a round, each player turns over two of the cards in their hand. In this implementation, this is done randomly for all players, removing the need for a player to make decisions outside of play and thus overall simplifying the game. From here, the game plays as described in the rules.
If at any point the deck becomes empty, the discard pile is recycled by creating a new shuffled deck using all cards in the discard pile bar the top card. If at the start of a player's turn all cards in their hand are visible, then the round ends. A full game consists of nine of these rounds, each round starting with a new deck.
Whilst the game is being played, game details and events are being stored in a running game history allowing for it to be saved to file at a later point. The format of this is discussed in Section 4.5. In some cases, the game may enter a loop where the same sequence of moves is constantly repeated. Using the game history, the number of times that the same turn has been made can be calculated and the game will end if that turn has appeared five times. This prevents games from continuing indefinitely.

4.3 Player

Each player has an attribute for a hand and a function approximator and methods for determining what card to draw and what to discard given a game state. How these decisions are made depends on the type of player implemented. Three different types of player have been designed and are as follows:

- **Random Player** - Each legal move is equally likely to be made and randomly selected via the python random library. This simulates a player with no understanding of the game and sets a benchmark where any improvements from it can be seen as learning.
- **Greedier Random Player** - When drawing, this player will draw from the deck if the card on top of the discard pile scores more than 6, otherwise they draw from the discard pile. When discarding, the player first attempts to swap the drawn card with the first face-up card in its hand that scores higher. If all cards score lower, then it randomly chooses between the face-down cards or to discard it. This therefore simulates a player with a much better understanding of the game (a heuristic player), thus serving as a better benchmark when evaluating player performance.
- **Golf Player** - Each move made is determined by a policy, discussed in Section 5.1, which determines moves based on the evaluation of game states. This evaluation is done by the players function approximator, by passing in the game state and returning some value. It is this that it updated during training.

4.4 Function Approximator & Neural Networks

The function approximator of a player will be used to evaluate the current game state and assign it a numerical value. The function approximator will be some form of neural network depending on the training algorithm used. The coevolution training algorithm will use a standard feed-forward neural network as a function approximator, the NEAT algorithm will use a more "abstract" feed-forward network, with no distinct hidden layers.
With regards to the standard feed-forward network, there are two options for implementing this within the system:

- **Use an existing library** - There exist many different python libraries with implementations of neural networks, such as Sci-Kit Learn [18] and TensorFlow [19]. Using such libraries would rid the need to develop a custom neural network, however they might not have the precise functionality that is needed for the training algorithm.
- **Create a bespoke implementation** - A custom neural network class can be created that has only the required functionality needed for the training algorithm. This would include access to the

weights to be able to update them by a set amount and a method to run the input data through the network. However, this will not have had the level of use and testing that existing libraries have.

After reading the documentation of some of the more common neural network libraries, the decision was made to write a custom implementation for the training algorithm as none of them had the desired functionality. This custom implementation makes use of the NumPy library [21], representing the weights and biases as matrices, allowing for weight updates and passing data through the network to be performed via matrix operations.

4.5 Writing games to file

An important part of this project is to be able to analyse the performance of the players over time, to do so requires the data to exist outside of the training environment, written to a file. Given the vast number of games that will be run, the data needs to be stored in as concise a format as possible, whilst also allowing for information to be extracted easily. Designing this format requires some understanding of what information will be needed for later analysis, which is described later in Section 5.2. During the development and training of the players, new methods of analysis may be also considered which requires further information. Therefore, the designed format will be "backwards-compatible" to be able to analyse new patterns from older games files.

The designed format takes inspiration from [20] who represents game events with specified characters. In order to represent each card as a unique character, the suit and value of the card can be substituted into an equation to calculate the corresponding ASCII value of its character ranging from "A" to "v". Hidden cards are represented by the "#" character. The tables below demonstrate how characters link to game events:

Game Feature/Event	**Range of Characters**	**Number of Characters**
Number of players	1 - 4	1
Starting player index	0 - 3	1
Starting hands of players	A - v and #	6 per player
Start of the game	<	1
Turns	*See Table 3*	*4 per turn*
End of the game	>	1
Ending player index	0 - 3	1
Face-down cards per player	0 - 4	1 per player
Final hands of players	A - v	6 per player
Scores of players	-4 - 60	2 per player (0 padded)

Table 2: Range of characters for each game event or phase

Turn Phase	**Range of Characters**	**Number of Characters**
Draw location	+ or -	1
Drawn card	A - v	1
Discard location	0 - 6	1
Discarded card	A - v	1

Table 3: Range of characters for each phase of a turn

This representation is for a single round of Golf. A full game consists of 9 rounds, with each round separated by a newline character, and each game separated by an additional newline character.

The data needs to be saved in a way that minimises the amount of reads and writes to files, when reading and saving the data, whilst also reducing the amount of work that would be lost in the event of a failure. As such, games will be saved to file in batches. In addition, as the files are being created automatically, the names of the files should be constructed in such a way that prevents duplicate names from occurring.

The date-time of the file creation was found to be a suitable name for this implementation, due to the time difference between these file writes. This name also includes any additional game information which may be important to analysis such as the player position or generation number.

4.6 Serialising players

To allow for the player to persist between sessions, it can be saved to a file and re-loaded at a later point. There are many different methods that exist to serialise an object, the method being used for this project is python's pickle library. Pickle provides the serialising functions needed to allow for regular backups of players to be made during training, which will coincide with the writing of games to file.
An additional use of serialising players is that they can be loaded into different environments for different purposes. One of the main examples of this use will be when evaluating each player post training in the round-robin style tournament, as mentioned in the project specification (see Section 2).

5 Methodology

5.1 Policy of deciding cards

Each turn, the Golf Player makes two decisions, where to draw and what to discard. It does so via the use of a policy which maps actions/moves to states. This is the same policy that is described in [15] with a slight modification* to better reflect human decision making.
The policy will often refer to the maximum value of a card. This is a value calculated by evaluating each game state, via the player's function approximator, that would result from exchanging a given card with each card in the player's hand and selecting the largest value. Consider the following example in Figure 3, where the given card is the Ace of Clubs;

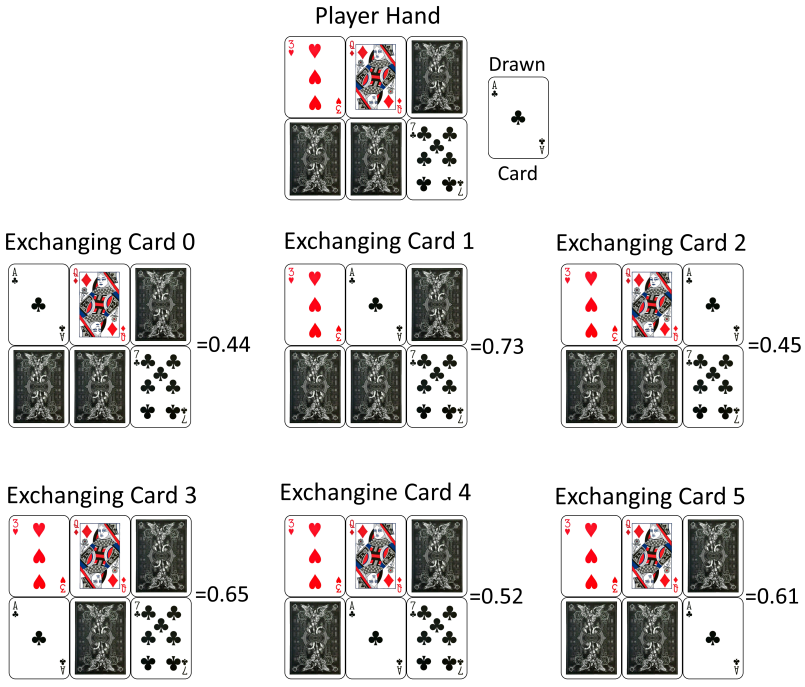

Figure 3: A visual representation of the policy

The six possible game states are evaluated by the player's function approximator, the highest value being 0.73. This would therefore be the maximum value of the Ace of Clubs.

The policy that the player uses is as follows:
When determining the drawing location, the maximum value is calculated for the top card of the discard pile and all cards whose location is unknown, these being the cards that could potentially be drawn from the deck. If the maximum value of the card on top of the discard pile is greater than 50% or more of all potential cards in the deck, then the player will draw the card on top of the discard pile. If not, then the player will draw from the deck. *In addition, if the card on top of the discard pile does not improve the player's hand, then the player draws from the deck.
When determining what to discard, the maximum value of the drawn card is calculated and the exchange which returns this value is made. If the card does not improve the value of the player's hand, then it is discarded.
Referring again to Figure 3, the maximum value of 0.73 is calculated by exchanging the Ace with the Queen. Assuming that the value of the player's hand is low, e.g. 0.50, the player would therefore exchange the Ace with the Queen and place the Queen on the discard pile.

The use of a fixed policy does introduce some human knowledge to the learning process by specifying a way of making decisions. However, the purpose of this project is not for the player to learn to play the game, rather for the player to learn to evaluate the many states of the game. Therefore, the use of a fixed policy allows for the consistent decision making of players during training.

5.2 Measuring performance

Measuring the performance of a player is a crucial aspect of this project, as this analysis is done both during and after the player is trained. Analysis during training is important as the training algorithms require an understanding of the performance of the player in order to appropriately alter and update them. Analysis after training is important as it is used for evaluating the performance of the trained player and subsequently the various training methods or parameters used. It also allows for some insight into the learned behaviours of the player (how they play the game) and how these behaviours have changed during the training process.

With regards to Golf, the biggest indicator of an improvement in performance is the score that the player achieves. As the aim of Golf is to score the fewest points, the players will be mainly assessed on how low they score in a game.
There are two ways in which a player can reduce their score, by prioritising lower scoring cards and matching cards of the same value. These behaviours can be detected by analysing the hands of a player at the end of a round, calculating how often the player makes matches and how often cards of a given value appear in their ending hand.
There are additional statistics that assist in measuring a players performance, such as:

- **Wins** - This shows how well the player performs against their opponent. Whilst this is a useful measure of performance, it is wholly dependent on the ability of the opponent and thus won't give an accurate and reliable measure of the player's performance during training. This statistic is therefore more useful for post-training analysis, as the player is fixed and no longer being updated and will be used when comparing players that have different inputs or have been trained under different algorithms.
- **Rounds ended** - This shows how often a player was the first to turn all six cards face-up and end the game. A lower number may indicate that the player has adopted a greedy play-style where, for example, it may be waiting for better cards to appear rather than progressing towards the end of the game.
- **Turns per round** - This shows how long a round of golf between two players lasts and may give further insight into player behaviour and how these behaviours interact with each other.

5.3 Randomness

5.3.1 Reducing the effect of randomness

All card games have some element of randomness in them, most commonly this is the order of the cards after the deck has been shuffled. Golf uses a 54-card deck and thus there are 54! (2.3×10^{71}) possible ways in which the deck can be ordered. Of these starting game states, some may be in a permutation that gives one player a significant advantage over the other. This has the potential to result in an inaccurate assessment of the player's performance, therefore such advantages need to be minimised.

The main way in which this project aims to reduce the impact of the randomness is to play games in pairs, as implemented in [15] and [16]. This means that the same game will be played twice, but with the initial positions of the players reversed. Doing this helps to evenly distribute any advantages, or opportunities at these advantages, received from the starting state of the game amongst all players.
In addition, whilst it is impossible to play even a fraction of the possible games, increasing the number of games played will ensure that players are exposed to a wider variety of games during training. Further, when evaluating the performance of the player, playing more games allows for more accurate averages to be made and thus more reliable conclusions to be drawn. In this project, players play 10 games (5 pairs of games) per epoch/generation, compared to the 4 games (2 pairs of games) played in [15]. The reasoning for this will be discussed further in Section 6.1.
It should be noted that playing pairs of games only allows for all permutations of positions for a two-player game. The same game would therefore need to be played more times when using more than two players, more specifically n! times where n = number of players, in order for all permutations of positions to be covered.

5.3.2 Repeating randomness

Ensuring that a game can be played a second time requires that the same sequence of random events occurs, including the game initialisation. This is achieved by playing the pair of games in "parallel" rather than sequentially. This means that the first game plays round 1 followed by the second game also playing round 1. This continues for each round in the game. Playing the games in this way allows for greater control over the random events that would occur within a game.
To ensure that rounds are initialised identically, a deck can be generated beforehand and a copy of it can be used for the initialisation of both rounds. Copying the deck prevents the object reference from being passed, allowing for the master deck to remain constant. To control any further randomness in the round, the seed of a random module can be set, allowing for the same random sequence to be generated. The main use of this is to ensure that the cards that are turned over at the start of each round are the same for both games. Here an array of seeds (integer values) are generated beforehand, with one seed for each round of the game. Before each round is played, the corresponding seed from the array is set as the seed of the random module, allowing for all random events to be replicated.
However, there are other elements of this project that rely on randomness, such as the random player, and therefore precautions must be taken to ensure that this process does not affect them. This is accomplished by using a separate and independent random module for the areas where randomness needs to be repeated. The NumPy [21] random module is used for this purpose, with all other random elements using python's random module.

5.4 Input to the Function Approximator

The function approximator is fed data about the current game state, more specifically what cards are in what locations, and this data can vary in the way it is represented.
This input data can be represented numerically, known as Integer Encoding, as implemented in [15] where cards were assigned values based on their location in game. These values represented how accessible cards in each location are to the player, with more accessible cards being assigned positive values and less accessible cards being assigned negative values.

There are several problems with the approach. Firstly, it requires some initial human knowledge about the accessibility of locations when assigning these values, meaning that the player is not fully learning independently. Secondly, it is very difficult for this representation to distinguish between the positions of the cards in the hand without introducing unintended biases or relations where they do not exist. For example, position 2 may be valued less than position 5, as 2 is smaller than 5. This is problematic for Golf as the positioning of the cards in the hand can significantly reduce or increase the player's score. However, the main benefit of this representation is the significantly reduced input size, which results in a simpler network with far fewer weights. The number of weights that a network has is important as more weights can negatively affect the performance of the training algorithms due to there being more attributes that need to be optimised.

The input data can also be represented via one-hot-encoding, which is another encoding method to represent categorical data. In this, each data item is represented by a series of bits, with one for each potential category, where only a single bit is active (1, all others are 0). This was also suggested by [15] as an area of further research. This approach not only allows for the data to be represented without any pre-defined weightings, but also provides a better abstraction of data for machine learning algorithms, due to the prevention of any unintended relations. However, this will likely result in a significantly larger input size, depending on the number of categories, thus resulting in a more complicated network due to the number of weights it contains.

The input can also vary by the amount of information that it represents. The input could contain as little as the cards in the player's hand, effectively restricting the access of information about other game features to the player, such as the cards in the discard pile. Conversely, the data could be completely unrestrictive and contain all information about the game state. However, the more information that is given, the harder it becomes to represent this in a concise input, meaning that the resulting network becomes more complicated. The amount of information given may also result in different behaviours being learned by players due to a lack of/gaining of an understanding about aspects of the game.

Chosen input representations

Initial experimentation, which will be discussed in Section 6.1, and the aforementioned information led to the design of three different input representations to be used in the project, all represented in one-hot encoding. They are;

- **One hot hand** – Each card in the hand of the player (6) is represented by one of 15 possible states, the first 14 represent the values of the card (1-13 + Joker) and the final state represents that the card is unknown (face-down). This results in the game state being represented by 90 inputs.
- **One hot state** – Each card in the game (54) is represented by one of 9 possible locations the card could be in. It is either in the opponents hand (face-up), in the discard pile, in the players hand (one bit for each position), or the cards position is unknown. This results in the state being represented by 486 inputs.
- **One hot state and hand** – This is a combination of the two previously mentioned representations. The input first contains an array of 54 bits representing whether each card in the game is in the opponents hand, followed by a second array of 54 bits representing whether each card in the game is in the discard pile, and a third array of 54 bits representing whether each card in the game is in an unknown position. Finally, the input contains the information about the hand of the player which is identical to the one hot hand representation. Overall, this state is represented by 252 inputs.

Each input has its own method which can convert the current state into its format to allow for players with different inputs to play against each other.

5.5 Description of Algorithms

In this project, players will be trained under two different neuroevolution algorithms, coevolution and NEAT. The following is a description of the final design of the algorithms. The experimentation that led to this design will be further discussed in Section 6.

5.5.1 Coevolution

The implemented coevolution algorithm is influenced by [15], and to some extent [16], focussing on the training of two players simultaneously, referred to as the player and opponent. These players each have fixed-structure function approximators with 27 nodes in the hidden layer and a single node in the output layer. Each epoch sees the two players play 10 games against each other, split into 5 pairs. The player with the lowest total score is considered to be the better player and the worse player is updated.
A player can be updated in three different ways:

1. **Mutation** - A Gaussian noise with mean 0 and standard deviation 0.1 is added to each of the weights of the worse player's neural network.
2. **Move/Crossover** - The weights of the worse player's neural network are moved 5% in the direction of the weights of the better player's network. For example, if the opponent was the better player, the player's weights would become 0.95×player weights + 0.05×opponent weights.
3. **Replacement** - The worse player is replaced by a **mutated** copy of the better player.

The severity of the update made is relative to the difference between the scores of the player. Due to the aforementioned impact of randomness, the algorithm defines a threshold which denotes how many points fewer per game a player needs to score to be considered the better player. This threshold is set at 15 points per game. At the end of each epoch, the player or opponent is updated under the following rules: If the difference between scores is greater than the threshold, then the worse player is updated via crossover. However, if the difference between scores is greater than twice the threshold, then the worse player is updated via replacement. If the difference between scores is less than the threshold, then the opponent is updated via mutation. This process repeats for each epoch in the training process.

5.5.2 NEAT (Neuroevolution of Augmenting Topologies)

The core functionality of this algorithm is provided by NEAT-python [22], an open source python implementation of the NEAT algorithm [17].
NEAT-python provides implementations for the genome encoding, reproduction and mutation of solutions, speciation of a population and converting solutions to neural networks. It also allows for the parameters of the solutions and algorithm, such as the number of inputs and mutation rates, to be modified via a configuration file.
Addition functionality has been added to this in the form of a custom fitness calculation. Two identical yet independent players are created using the solution as their function approximator and they play 10 games against each other. Because the players use the same solution, playing pairs of games would produce identical results and therefore 10 unique games are instead played. The fitness is calculated by averaging the lowest scores achieved by either player across the 10 games and subtracting that from 540, the theoretical maximum possible score. The lower a player scores, the higher their fitness.
Once all solutions have been assigned a fitness, the offspring solutions are created via crossover using solutions selected proportionally to their fitness from the population and mutated. For this implementation, when mutating, solutions have a:

- 50% chance to add a new connection.
- 30% chance to remove a connection.
- 50% chance to add a new node.
- 20% chance to delete a node.
- 80% chance to mutate the weighting of a connection.
- 10% chance to replace the weighting of a connection.
- 70% chance to mutate the bias of a connection.

- 10% chance to replace the bias of a connection.

These new solutions are then added to the population, replacing the older solutions with the exception of the two best solutions from each species. This process then repeats for each generation. In implementation, this algorithm uses a population size of 30 and runs for 50 generations.

6 Experimentation

6.1 Coevolution

6.1.1 Initial implementation

Initially, the coevolution algorithm was implemented as it is described in [15]. The player and opponent play an epoch (training cycle) of 4 games (2 pairs) against each other and the number of wins for both are recorded. If the player wins 2 or fewer games, then they are considered the worse player and their weights are moved 5% in the direction of the opponent's. If the player wins more than 2 games, they are considered to be the better player and their weights are not updated. After every epoch, the opponent is mutated by adding Gaussian noise to each of their weights.

These initial tests used what would be later known as the one hot state input representation and fed this data into a feed forward neural network with 27 nodes in the hidden layer, a similar structure to that used in [15].

Several trials of this algorithm were run with the intention of optimising parameters regarding the saving of log files and player backups and ensuring that the program runs correctly. However, these trials showed no improvements in the scores of both the player and opponent over roughly 1,000 epochs each.

As such, some slight adjustments to the algorithm were made which involved modifying the algorithm parameters and increasing the games played per epoch to 6 games (3 pairs), where the player was updated if they won fewer than 5 games, proposed in [16]. However, there were still no visible signs of learning by either player with scores remaining poor and unchanged.

Whilst it is possible that more epochs might have been needed to begin to show some results, the pattern of results differ from those of [15] and [16]. Their results show a high rate of improvement early on in training with this rate decreasing over time, almost plateauing (see Figure 4).

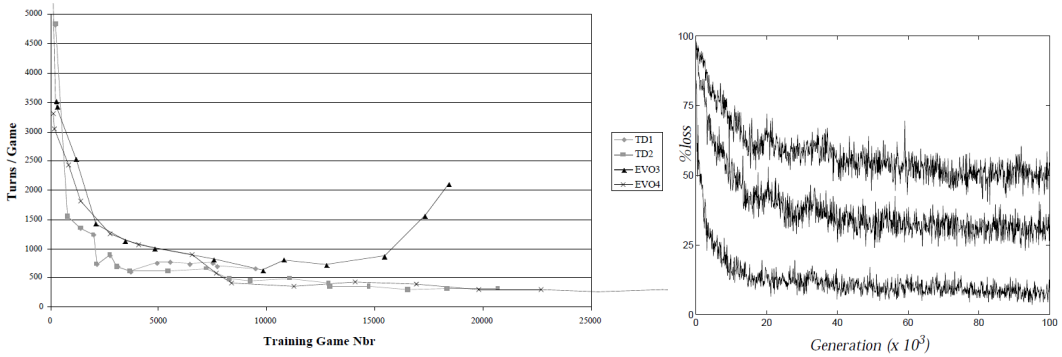

Figure 4: Results from [15] (left) and [16] (right)

6.1.2 Hypotheses for poor results

After analysing the program to ensure it is running correctly, a full analysis of the initial experiment was conducted to highlight any areas of training that might be responsible for the disappointing results. This analysis showed two main points of concern.

The first point is the size of the input and the resulting network it produces. The representation used in initial experiments contained 486 inputs which, when combined with the 27 hidden nodes and their biases, results in a network containing over 13,000 weights and biases, all of which need to be optimised. The

existing literature use far smaller inputs which resulted in much smaller and simpler networks ranging from 1,400 [15] to 4,000 [16] weights. As previously mentioned, a less complex network (fewer weights) will improve the performance of the training algorithm due to the reduced number of parameters to optimise, especially in a hill-climbing algorithm such as this. Therefore, designing and using a more concise or restrictive input representation could benefit the coevolution training algorithm.
The second point is that there are some concerning areas with the design of the initial coevolution algorithm which may be impeding its performance. The algorithm is favoured towards the opponent when determining the better player as the opponent does not need to win as many games as the player does, especially for an increased number of games. Further, there is minimal elitism within the algorithm as good solutions do not tend to remain constant. This is more apparent for the opponent who is always mutated after each epoch regardless of their ability. There are also other statistics that can be used to determine the better player aside from game wins, which is heavily influenced by the ability of the opponent. Also given the visible variance of the scores in the game, an epoch of 6 games may not be a large enough sample of games to reliably determine the better player.

6.1.3 Learning to score hands via supervised learning

Before other input representations are designed, it was first tested whether the desired information could actually be learned from the data it is given. As it is impossible to obtain the value of all states in the game, a neural network was trained under supervised learning with the intent of learning the scoring system of Golf. As the aim of Golf is to reduce the score of your hand, evaluating each game state will involve some estimation of the current score of the hand. Therefore, it is safe to assume that if a network is able to learn from this data, then it is plausible that it can evaluate game states.

The Sci-Kit Learn [18] python library is used for this supervised learning, more specifically its MLPRegressor object. This is trained on a dataset of potential hands and their target scores, using backpropagation via its inbuilt *fit* method, with the aim to reproduce the scores as closely as possible. Once trained, the network is evaluated on a second dataset of hands and scores to assess the accuracy of its predictions.
The network was trained with hands denoted in three different representations. They were:

- **Numerical** - Each card in the hand was represented by the cards value as an integer.
- **One hot value** - The above numerical representation is encoded in one-hot-encoding where each card is represented by 14 bits, one for each value.
- **One hot deck** - Each card in the game (54) is represented by 7 bits, one bit for each position in the hand and a final bit representing that the card is not in the hand.

Two metrics will be used to assess the accuracy of each trained network. They are:

- **Mean absolute error** – This shows, on average, how much the network's predictions are incorrect by. An ideal value is 0, with higher values indicating a more inaccurate network. Mean absolute error is used over mean squared error as it is easier to relate back to the scoring of the game.
- r^2 **value** – Known as the coefficient of determination, it provides a measure of how close the data is to the regression line. The value can range from 0 to 1, where a higher value represents a better fit of the data. It should be noted that a high r^2 value does not always denote an accurate network, this could be an instance of overfitting where it is only accurate to the dataset it has been given. However, given that the scoring of Golf is inherently a predictable occurrence, this is not of great concern especially when combined with a second measurement.

Each representation was used to train 10 networks on a training set of 100,000 hands. The networks were evaluated on a further dataset of 100,000 hands, the results shown in Table 4.

Representation	Mean Training Time /s	Mean Absolute Error	Mean r^2 Value
Numerical	209.90	1.773	0.889
One hot value	673.97	0.839	0.979
One hot deck	1273.00	0.552	0.990

Table 4: Averaged metrics from supervised learning of Golf scoring

These results firstly show that the scoring system can be learned by a neural network and therefore it is safe to assume that a network is able to learn to evaluate game states. In addition, these results also show that when the data is encoded with one-hot encoding, the accuracy of the network improves as seen by the lower mean absolute error and higher r^2 value. Finally, networks trained with one hot deck, a representation that closely resembles the input for the initial experiments, were the most accurate. This implies that the further the data is abstracted, the more accurate the network is able to be. However, it should be noted that as the input size increased, the time each network took to train also increased.

6.1.4 Redesigning input representations

In an attempt to drastically reduce the input size, a representation was designed to contain the minimal amount of information about the game state that a player would need. This resulting input, known as one hot hand, only contained data regarding the players hand. This is very similar to the one hot value representation, the difference being an added bit to allow incomplete hands to be represented.

Two thirds of the initial input was devoted to representing solely the hand of the player. The newly designed one hot hand representation allows for a much more concise representation of the hand, thus it can be incorporated into the design of the initial input data to greatly reduce the input size. This input still contained information about the game state and was thus named one hot state and hand.

The designs of these input representations are described in Section 5.4.

Players using the two newly designed inputs were trained under the same conditions as the initial tests and unfortunately produced the same pattern of results, with no sign of learning. Therefore, it is safe to assume that the input representation is not the main cause for the poor results. Given this and the performance of the one hot deck representation under supervised learning, the initial input representation, now known as one hot state, was added back to the collection of inputs despite its size.

6.1.5 Redesigning the coevolution algorithm

As mentioned above, there are some concerning aspects of the design of the initial coevolution algorithm that might make it unsuitable for this specific environment. As such, these areas were modified or redesigned to address these concerns.

6.1.5.1 Increasing the number of games

From the data of the initial experiments, it is evident that there is a significant variability in the scores that players achieve within the same epoch. This raises questions about the reliability of determining the better player as the randomness of the game can result in players of worse ability winning. To improve the reliability of conclusions, they should be drawn from a larger sample size. Therefore, increasing the number of games played per epoch to 10 games (5 pairs) should improve the reliability of the algorithm when determining the better player.

6.1.5.2 Changing the updating

The algorithm always required the player to win more games against their opponent to be considered "better". In addition, good solutions rarely remained as they were. The opponent was always updated after each epoch, meaning that, not only did the player lack consistency in the quality of their opponent, good opponents were always changed. In addition, the player was always more likely to be updated due to the unfair conditions, thus it was more likely to move away from a promising solution. This was more problematic with an increased number of games where a player could win the majority of the games yet not meet these required conditions.

The algorithm was modified by allowing for both the player and opponent to be updated in the same way. Doing so removes any biases that the algorithm has towards the opponent but also allows for the better player, be it the player or opponent, to remain unchanged between epochs. In addition, the severity of the updates correlated to the difference in quality of the players.
With reference to the types of update discussed in Section 5.5 and incorporating the above change, the redesigned updating algorithm was as follows:

- If a player won 7-8 games in an epoch, they are considered the better player and the weights of the other player were moved 5% in the direction of that player.
- If a player won 9-10 games in an epoch, they are considered a significantly better player and the other player was replaced with a mutated copy of that player.
- If neither player won more than 6 games, they are considered of similar ability and neither networks is updated.

However, players trained under this updated coevolution algorithm again failed to demonstrate any signs of learning, with scores remaining consistently poor.

6.1.5.3 Changing how the better player is determined

As mentioned, the problem with using game wins as a metric is that this depends greatly on the quality of the opponent. For example, a player that scores 280 is a better player than one that scores 300 and a player that scores 190 is a worse player than a player that scores 180. Whilst these scenarios are acceptable in isolation, they are not if they occur within the same epoch. In the second scenario, the player is being "penalised" for losing whilst scoring less than the player that is "rewarded" for winning. This can result in some inconsistencies when determining the better player this way.

Given the aim of Golf is to score as few points as possible, the better player can also be determined based on the total scores of the players across an epoch. However, due to the randomness within the game, players of similar quality may not always score similarly. Therefore, some threshold needs to be established to denote how similarly players need to score to be considered of similar quality.
To establish this threshold, two random players (see Section 4.3) were played against each other in epochs to analyse how two players of similar ability score. On average, the random player's total scores differed by 5.91 points per game in an epoch, with the differences ranging between 0.4 to 11.5 points per game per epoch (see Appendix A). From these results, it is safe to assume that if a player scores 15 or more points per game less than its opponent, then it is highly likely that the player is of better ability.

This information then led to the design of the final coevolution algorithm, described in Section 5.5 by modifying the previously designed algorithm to incorporate player scores. One further addition made was to mutate only the opponent if the two players are deemed to be similar in ability. This prevented the players from becoming too similar to a point where neither would be updated because their scores were consistently within the threshold. This addition took inspiration from the initial coevolution algorithm, where the opponent was always mutated. It is then safer to assume that if the player is no longer improving then it has reached its optimum solution for this trial.
Players were then trained under this final coevolution algorithm, each with a different input representation and all players showed signs of learning. The scores of the players reduced during training in a pattern similar to that of the results of [15] and [16]. In addition, the player also played 10 games (5 pairs) against a random player at the start of each epoch. This allowed for some comparison of the ability of the player whilst it is learning against a consistent, albeit poor, benchmark. Analysis of these results will be explored further in Section 7.1.

6.2 NEAT

One of the main aims of the project is to compare the performance of the coevolution and NEAT algorithms. To ensure a fair comparison, the algorithms should be similar where possible, mainly when determining the quality of the player.

By developing the coevolution algorithm first, not only are these methods and representations known, they are known to provide desirable results, thus minimal experimentation is needed. This means that game scores will be used for assessing player/solution quality over a period of 10 games (5 pairs). In addition, players/solutions will be trained using the three different input representations.
Further, whilst it is not feasible to run the NEAT algorithm for the same number of generation/epochs as the coevolution algorithm due to it being a more intensive algorithm, it should still play a similar number of games in total. The algorithm therefore runs with a population size of 30 for 50 generations, resulting in a total of 15,000 games being played.

6.2.1 NEAT-python

NEAT is a difficult algorithm to implement due to the complexity of its genetic encoding, which allows for the reproduction of solutions amongst other things. This project therefore uses NEAT-python [22], an open source python implementation of the NEAT algorithm, to implement this core functionality, as described in Section 2.
A major benefit of using NEAT-python, aside from providing the core functionality, is that it has been thoroughly tested, not just as part of its development but also from feedback provided by other developers that have used this library. As such, it is a more efficient and reliable implementation, with regards to correctness, than one that has been custom developed. However, adding custom functionality to the algorithm is bounded by the limitations of NEAT-python. This means that certain features are not accessible without making changes to the source code, doing so may affect the reliability and correctness of the code provided.

6.2.2 Calculating fitness

The NEAT algorithm differs from the coevolution algorithm as it is a population-based evolutionary algorithm. This means that each solution needs some measure of its quality, referred to as fitness, to allow the algorithm to determine the better solutions within the population. As this project is exploring learning through self-play, the fitness of a solution needs to be calculated from the results of said solution playing games.

6.2.2.1 Opponent

In the coevolution algorithm, a player trains against an opponent, whom is exclusive to the player and whose quality improves over time. In NEAT, solutions are part of a population and do not have exclusive opponents. Each solution therefore needs an opponent to play its games against.
This opponent could be a random solution from the population, the quality (fitness) of which will improve over time due to the properties of a population-based algorithm. This closely resembles the behaviour of the opponent in the coevolution algorithm. However, this will lead to inconsistencies with the fitness calculations between generations due to the varying quality of opponents.
The opponent could also be the best solution from the previous population, whose ability will again increase over time. However, this opponent will most likely be better than most, if not all of the solutions in the current population, thus negatively affecting the performance of the solutions. This causes further problems at the initial generation as there is no previous generation to refer back to.
Further, all three of the above proposed opponents cause conflicts with NEAT's speciation of solutions. NEAT divides solution into species based on how structurally similar solutions are, preventing them from competing against solutions from a different species when reproducing. This gives newer solutions time to develop and optimise themselves before competing against the dominating solutions of the population [17]. Allowing for solutions from potentially different species to affect their fitness counteracts this.
Solutions could play games against a random player (see Section 4.3) which will provide a very consistent opponent that is not a solution generated by the NEAT algorithm, not causing the aforementioned conflicts. However, it will always be a poor performing player and may not provide a challenging enough opponent to play against.
Finally, the opponent for a solution could be itself. This method will restrict the variety of opponents that the solution will face, however the fitness evaluation will be consistent for each solution and will

calculate fitness without any external solutions. Therefore, in the implementation the solution will play against itself due to this method providing consistency amongst fitness calculations, not conflicting with NEAT's ideology and party due to the limitations and restrictions of NEAT-python.

6.2.2.2 Environment

To ensure further consistencies, each solution will play the same set of games when calculating fitness. Due to there being more than two solutions per game, an array containing shuffled decks and random seeds for each round is generated before any fitness is calculated. When each round is played, the corresponding seed is set and the corresponding deck is copied from the array and used for that round, allowing for a more efficient way to ensure the rounds are played with the same initialisation.
However, when two identical solutions play against each other in this implementation, they score the same when the positions are flipped, as also mentioned in [16]. Therefore, the need to play pairs of games in this method is redundant and instead 10 unique games are played. The solution will then be assessed based on the lowest score obtained from either position for all games.

This led to the design of the fitness calculation as described in section 5.5. Players were then trained using the three input representations and all showed signs of learning. Again, the results bared similar patterns to that of [15] and [16], but also the results of the coevolution algorithm. These results will be discussed further in Section 7.2.

7 Results and Analysis

To analyse the performance of the two neuroevolution algorithms, the performances of the final players after training are evaluated in addition to how those performances have changed over time (trends of the data). Analysis of the training process is accomplished visually by generating graphs that map specific metrics over time. The post-training analysis consists of analysing the game data gathered from 5,000 games (2,500 pairs) against an opponent. The game data/metrics used and their reasons are described in Section 5.2. Graphically, these will be represented by line graphs and stacked bar graphs.

When generating the data for this analysis, it should be done against the same opponent, or one of a similar quality, in order for comparisons between the two algorithms to be made. Game data was therefore generated by playing games against the random player, a consistent, albeit poor player. The results of two random players playing against each other are displayed in Table 5.

	Mean Score	**Percentage of games won**	**Mean matches per game**	**Mean turns per round**	**Percentage of rounds ended**
Random Player 1	278.80	50.52%	1.49	20.24	49.85%
Random Player 2	279.29	48.60%	1.50	20.24	50.15%

Table 5: Averaged results of the random player playing against the random player

Due to the differences between the two algorithms, additional data will be used to analyse each algorithm individually. NEAT allows for some additional population-based analysis whilst coevolution will be able to draw further data from games against the players training opponent.
The graphs in this section display the averaged data of each type of player, meaning the same training algorithm and representation, and the tables contain an averaged set of results. The full tables of results are available in Appendices B and C. In addition, players trained with an input representation will be referred to as *Representation* player (i.e. one hot hand player).

7.1 Coevolution

For each input representation, 5 players were trained under the designed coevolution algorithm.

7.1.1 Post-training

Table 6 contains the averaged statistical data of the trained players from games played against the random player.

	Mean Score	**Percentage of games won**	**Mean matches per game**	**Mean turns per round**	**Percentage of rounds ended**
One hot hand	140.72	99.99%	4.99	22.68	44.43%
One hot state and hand	189.85	98.74%	1.74	18.53	67.83%
One hot state	224.05	91.82%	1.71	15.20	83.37%

Table 6: Averaged results of the coevolution trained players against the random player

All players trained under each input representation are able to score consistently fewer points than the random player, therefore displaying an understanding of the game. The one hot hand players obtained the lowest scores followed by the one hot state and hand players and the one hot state players respectively. In addition, all players win the majority of games against the random player with the one hot hand players wining the most (nearly every game).

The one hot hand players also make a far greater number of matches per game compared to the other players, an indication as to why they are able to score the lowest. However, not every player using one hot hand was able to make matches as consistently as this. The average number of matches per game against the random player ranged from 2.17 to 7.27 (see Appendix B). The number of matches made by players using the other two representations are similar and show a minimal increase compared to that of the random player, although some trained players are able to match slightly more.

The one hot hand players play the longest rounds and end the smallest percentage of those rounds when playing the random player, followed by one hot state and hand and one hot state respectively for both metrics. This might imply that they have developed a greedier play-style by waiting for more desirable cards to appear.

7.1.2 Training

Mean Scores

As mentioned prior, the results of existing papers [15] and [16] show fast improvement during early training which slows down as training continues (see Figure 4). Figures 5 and 6 show graphs that plot the mean scores obtained by the player and its opponent throughout their training session.

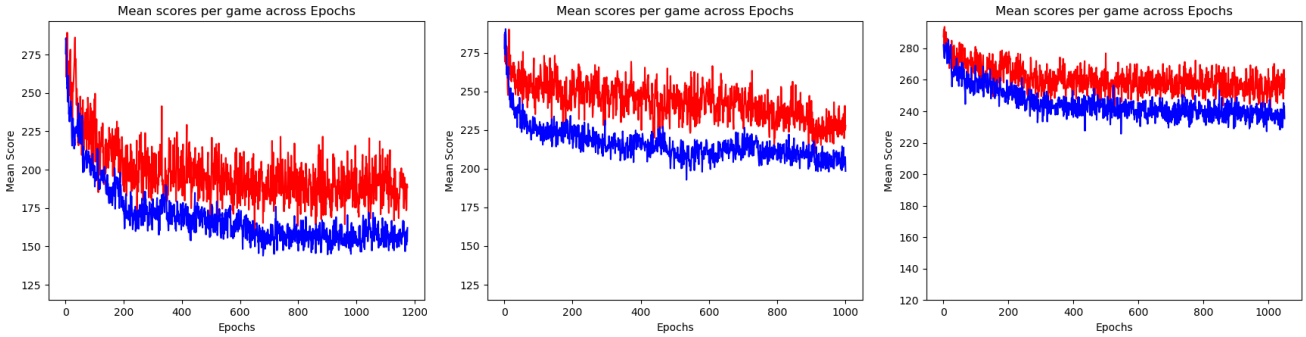

Figure 5: Mean scores of the player (blue) against its training opponent (red) during training.
One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

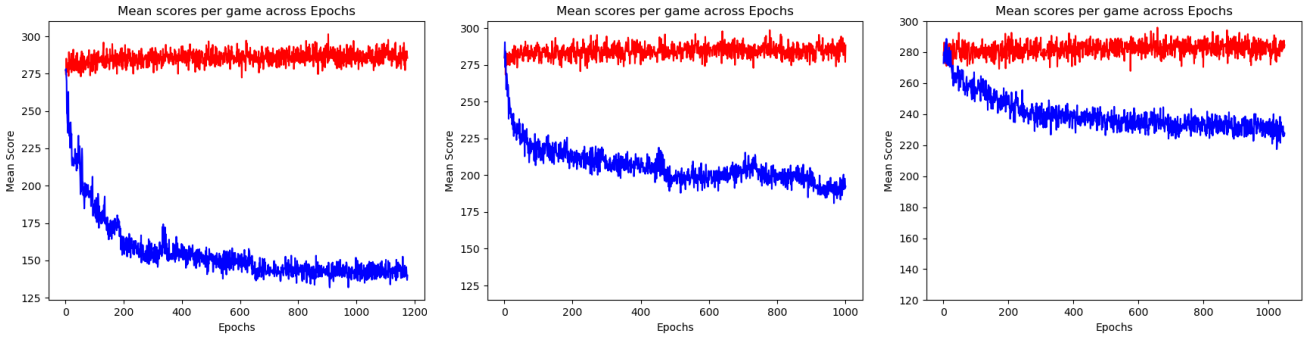

Figure 6: Mean scores of the player (blue) against the random player (red) during training.
One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

These graphs show that each player started from a poor position (high score) and has improved itself over time. The patterns of improvement shown are very similar to those of the existing literature, with the rate of improvement starting fast and slowing down during training.
The one hot hand players are able to make the most improvement in score, followed by the one hot state and hand players and the one hot state players respectively. In addition, it is evident from the graphs that the players perform better against the random player than they do against their training opponent. Therefore, against the random player the patterns and trends of learning are much more apparent.
The scores of the players tend to plateau at some point during training, meaning that they are either making minimal improvements or making no improvements at all, shown graphically where the scores appear to level out. This is due to the player reaching the optimal solution for its configuration, known as an optima. Due to the complexity of the search spaces of the function approximators, there are many local optima that the player can reach, some better than others. This results in the differing levels of performance of the fully trained players as the more complex search space will have far more optima.

Number of Matches

As discussed, the one hot hand players are the only players showing an increase in the number of matches made in games. The graphs, plotting the number of matches made over time, show that this increase in the number of matches seems to correlate with the decrease in the mean scores of the player. This could be due to the increased focus on the hand in the input that the one hot hand players use, allowing them a greater opportunity to identify and learn to the matching mechanic.
By contrast, the graphs of the two other representations show minimal improvement, if any, in the number of matches made, remaining level through the majority of training.

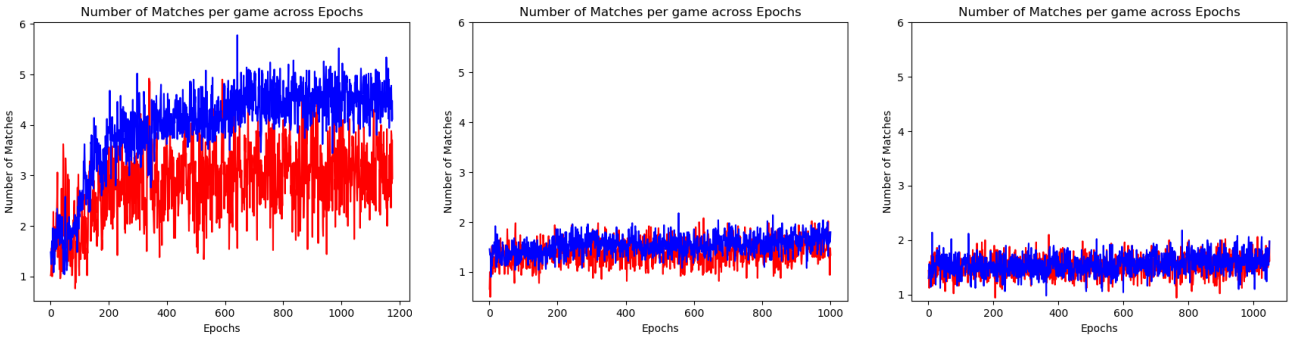

Figure 7: Mean matches made by the player (blue) against its training opponent (red) during training.
One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

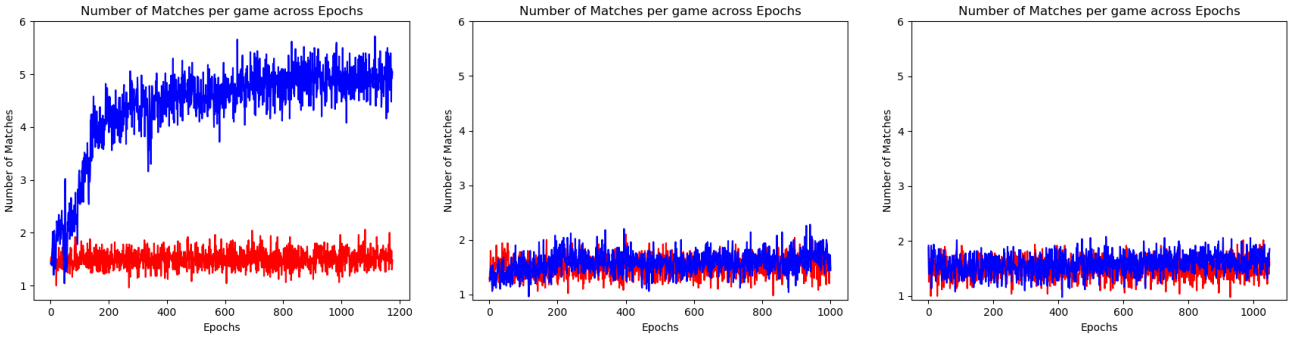

Figure 8: Mean matches made by the player (blue) against the random player (red) during training.
One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

Card frequency

The graphs in Figure 9 are a visualisation of the frequency of cards in the final hands of players across training when playing against the random player.

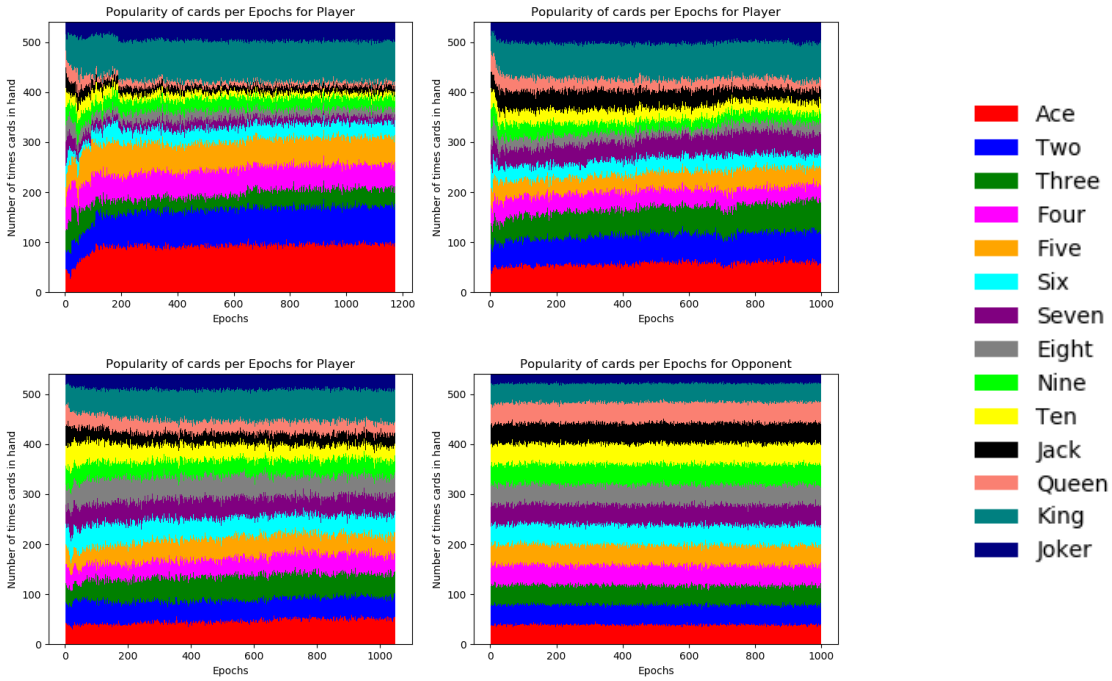

Figure 9: Frequency of cards chosen by the player against the random player during training.
One hot hand player (top left), One hot state and hand player (top right), One hot state player (bottom left), Random Player (bottom right)

From these graphs, it is apparent that all three types of players learn to favour some cards more than others. This is shown by the larger bars on the graph in comparison to the random player, whose bars remain equal and proportional (there are only two Jokers).

Of the cards that are favoured, most of the trained players recognise and subsequently favour the lower scoring cards, those being the Aces (Red), Twos (Blue), Kings (Teal) and Jokers (Navy). The one hot hand players favour cards more drastically, as evident by the far wider bars, compared to the other players, which corroborates the earlier conclusions that they have a greedier play-style.

Whilst the other representations do not favour cards as drastically, it is evident from their graphs that they do avoid the higher scoring cards, those being Queens (Pink), Jacks (Black) and Tens (Yellow).

Percentage of rounds ended

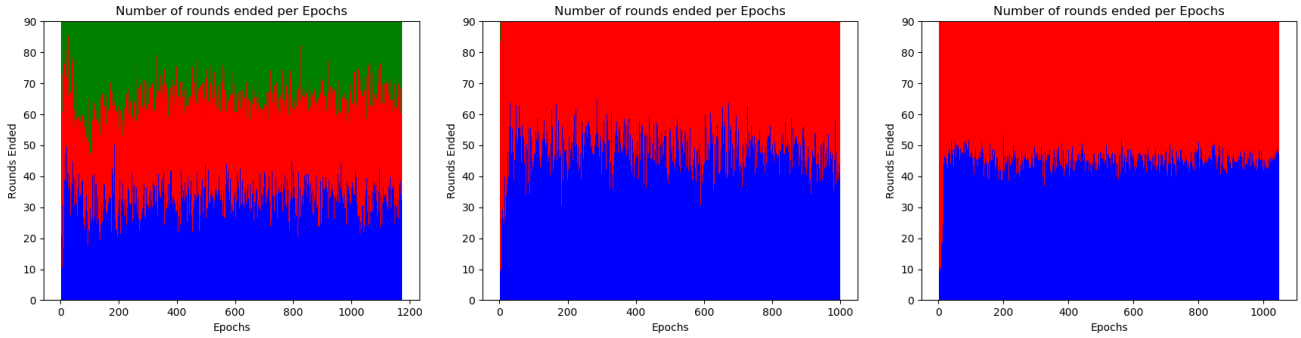

Figure 10: Number of rounds ended the player (blue) against its training opponent (red) during training. One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

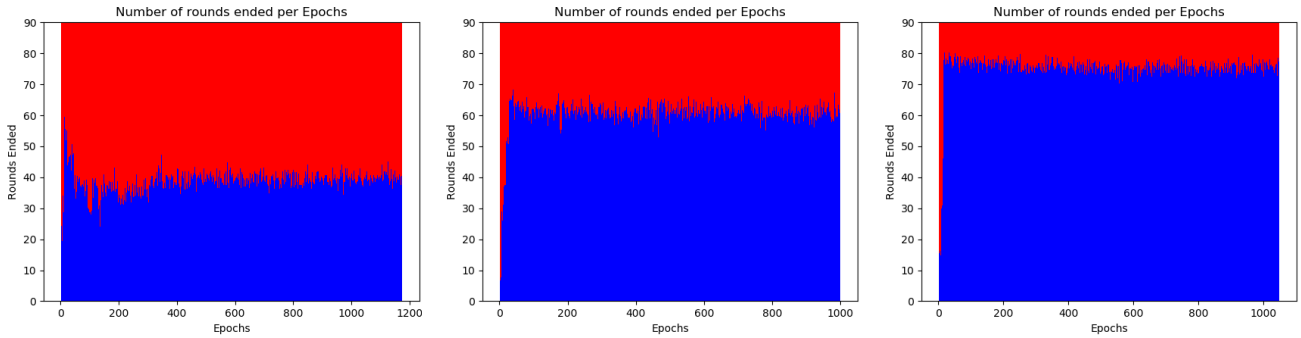

Figure 11: Number of rounds ended by the player (blue) against the random player (red) during training. One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

As discussed earlier, the rounds of the one hot hand players are often ended by the opposing random player. From Figure 10, the main reason for this is that many rounds are terminated during training due to stalemates between the player and its training opponent, denoted by green bars. Here, both players draw a card from the deck and immediately discard it, thus not progressing the game further. Where more rounds are terminated, the greedier the play-style is and thus the random player ends more rounds (see Figure 11).
By comparison, the two other representations, which have shown no obvious greedy behaviour, very rarely have rounds terminated and thus end more rounds against the random player.

7.2 NEAT

Due to the added complexity regarding the overhead of the algorithm, only 3 players were trained for each input representation under the NEAT algorithm. In addition, the differences in the algorithms means that the analysis must also differ. For example, data from games against the training opponent will be replaced by data from the population of solutions during training.

7.2.1 Post-training

The following tables contain the averaged statistical data of the trained players from games played against the random player.

	Mean Score	Percentage of games won	Mean matches per game	Mean turns per round	Percentage of rounds ended
One hot hand	130.51	99.99%	5.18	26.47	15.75%
One hot state and hand	164.68	99.88%	2.27	28.04	6.47%
One hot state	217.79	93.93%	1.82	19.17	58.48%

Table 7: Results against the random player

All players trained under each input representation are again able to score fewer points on average than the random player, thus displaying an understanding of the game. The one hot hand players consistently score the lowest, followed by the one hot state and hand players and the one hot state players respectively. All players also win the majority of games against the random player. In addition, the one hot hand players are able to make significantly more matches per game compared to the other players, although there is a larger difference between the one hot state and hand players and the one hot state players.
The one hot hand players play long rounds and end a low percentage of them, again showing evidence of a greedy play-style. However, the one hot state and hand players also show these greedier tendencies, which differs from the coevolution players. Their rounds last longer, on average, and they end a very low percentage of those rounds, even when compared with one hot hand players. The one hot state players still score the highest, make the fewest matches, end the highest percentage of rounds and have the shortest rounds, which is consistent with the results from coevolution.

7.2.2 Training

To analyse how the player/solutions have changed during training, information about the population is mapped. The metrics of the best and worst solutions (determined by fitness) are plotted along with the population average and, where relevant, the highest and lowest recorded metric at each generation. In addition, the best solution at each generation plays an epoch of 10 games (5 pairs) against the random player to provide additional methods of comparison with coevolution.

Mean Scores

The following graphs show the mean scores of the population during training, which were used to assign fitness, and of the best solutions against the random player.

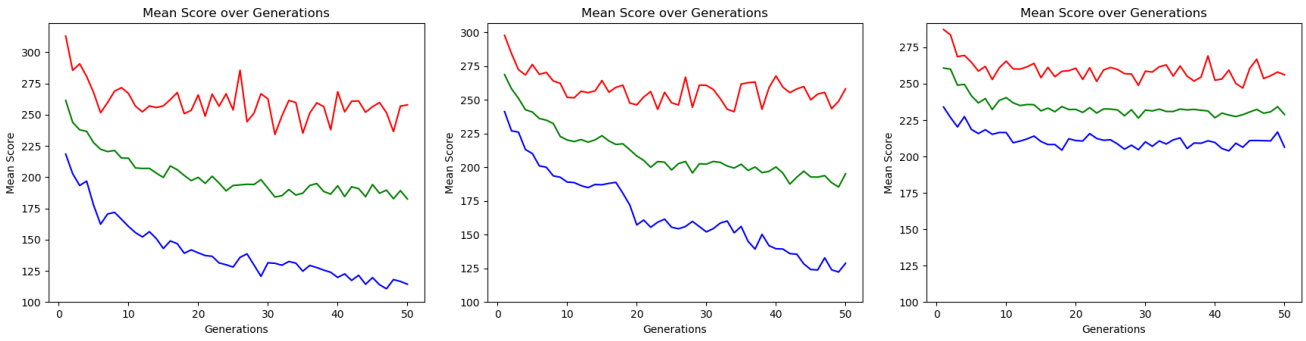

Figure 12: Highest (red) and lowest (blue) scoring solutions in the population and population average (green) at each generation.
One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

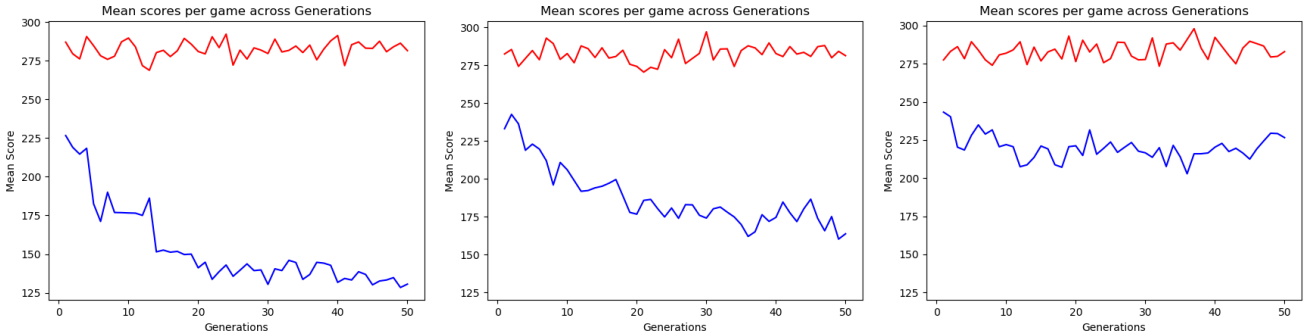

Figure 13: Mean scores of the best solution at each generation (blue) against the random player (red).
One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

As is the expected behaviour of an evolutionary algorithm, the performance of all solutions in the population improve from the initial generation. This is evident by the average scores of the population decreasing during training. The patterns of improvement are also similar to the results of [15] and [16], with greater improvement earlier in training compared to the latter stages of training. This can also be seen in the games against the random player.

The rate of learning is greater for one hot hand players and one hot state and hand players, but one hot state players do not display as much of a significant change in performance, if any at all. This is clearer in the results against the random player. Despite the minimal amount of learning, the one hot state players stay consistently better than the random players, implying that the way in which the networks are constructed under NEAT is advantageous to performance.

Due to populations containing multiple solutions, there is a greater possibility of solutions starting in more favourable configurations. This is shown by the best solution of the first scoring less than the random player, before any reproduction and mutation has taken place.

The scores of the solutions still plateau or slow down, however as there are more solutions in a training session, there is a greater probability that any solution will escape a local optima and improve itself.

Number of Matches

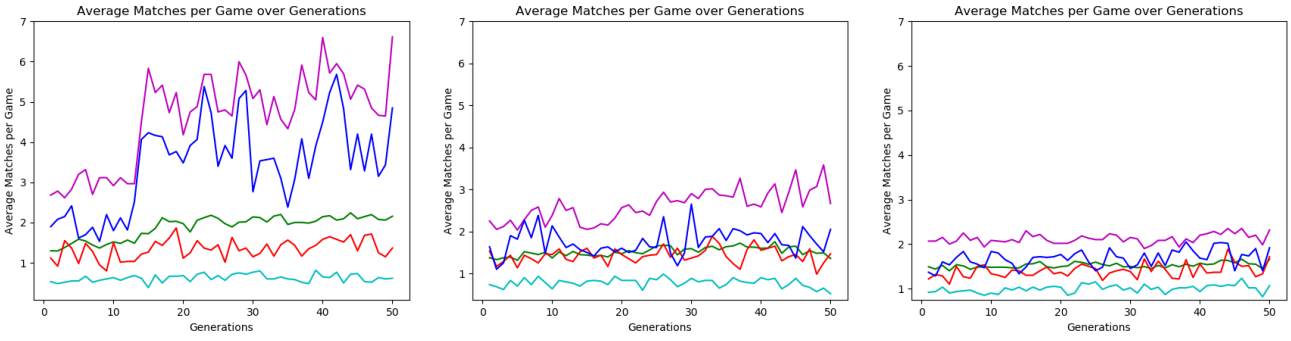

Figure 14: Mean number of matches made by the best (blue) and worst (red) solution at each generation as well as the population average (green), highest (magenta) and lowest (cyan) at each generation.
One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

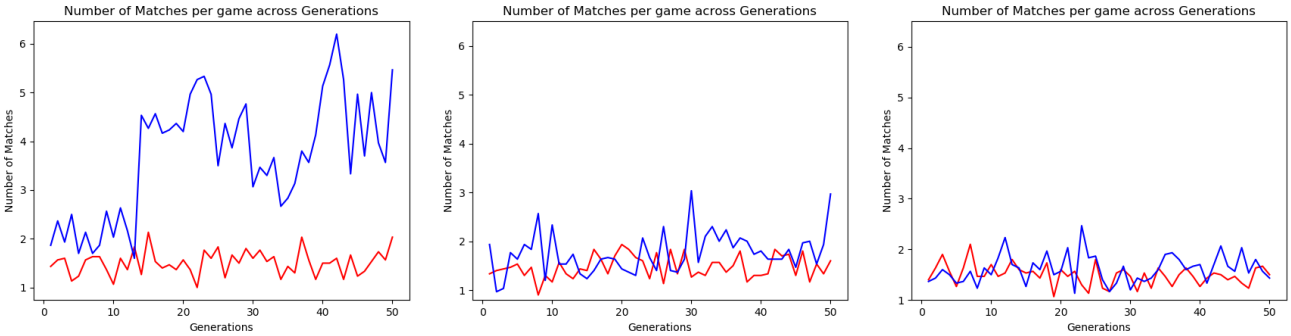

Figure 15: Mean number of matches made by the best solution at each generation (blue) against the random player (red).
One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

The one hot hand players were the only players to show any significant improvement in the number of cards matched. From the graphs that moment at which solutions learn to match can be seen by the large spike.
These graphs also show that there are solutions in each population that are able to make more matches than average. The best solutions for one hot hand players tend to be those that make more matches and the worst solutions make fewer. For the other representations, there is minimal correlation between the number of matches and the quality of the solutions as the better and worse solutions make a similar number of matches as the population average.

Card frequency

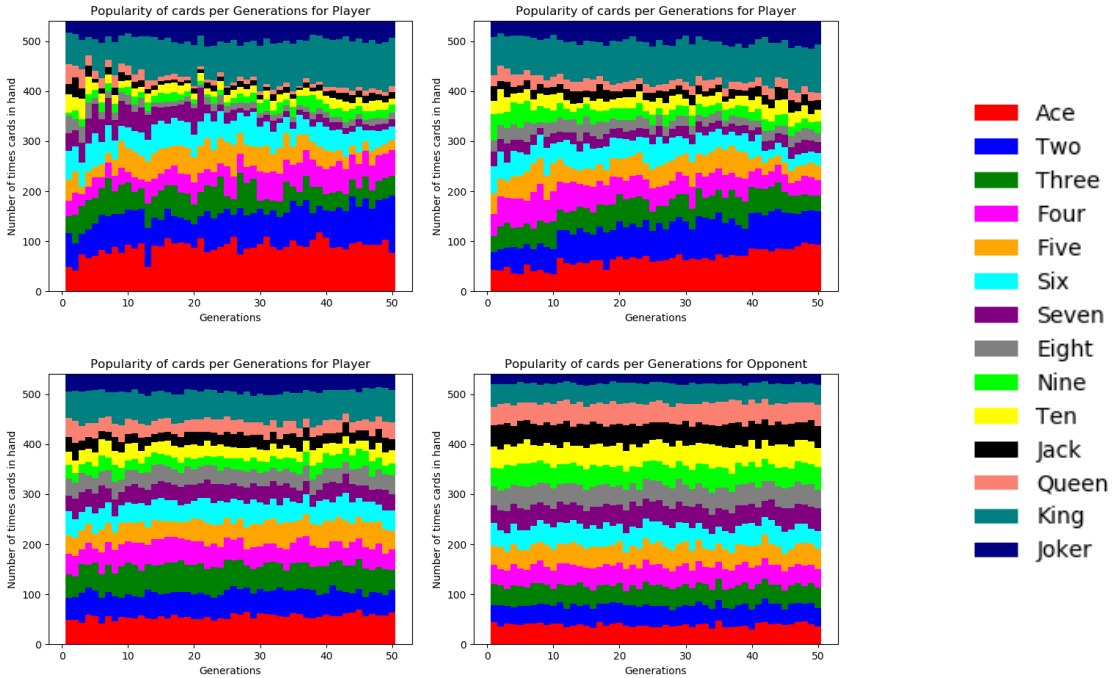

Figure 16: Frequency of cards chosen by the player against the random player during training.
One hot hand player (top left), One hot state and hand player (top right), One hot state player (bottom left), Random Player (bottom right)

Again, all three types of player visibly favour certain cards, which is evidence of some amount of understanding. One hot hand players and one hot state and hand players display this more clearly, favouring

lower scoring cards such as Jokers (Navy), Kings (Teal) and Aces (Red). In addition, the higher scoring cards such as Jacks (Black) and Queens (Pink) appear less frequently. The graphs also shows the change in the frequency of these cards over time, thus further visualising their learning. The one hot state players also show this behaviour, but not to the same extent.

Percentage of Rounds ended

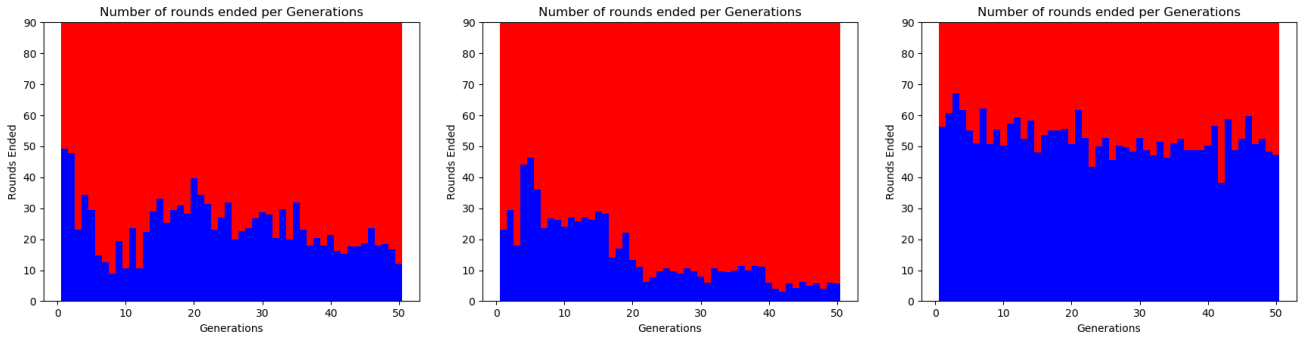

Figure 17: Number of rounds ended by the best player at each generation (blue) against the random player (red). One hot hand player (left), One hot state and hand player (centre), One hot state player (right)

As discussed earlier, both the one hot hand players and one hot state and hand players end a very low percentage of the rounds played, signifying a greedy behaviour. These graphs show that this greedy behaviour is likely learned as the earlier generations end a higher percentage of rounds, with this decreasing over time. Even when considering the one hot state players, there is a visible decrease in the number of rounds ended through the generations, implying that this greedy behaviour might be as a result of the NEAT algorithm.

7.3 Comparison between algorithms

When comparing the statistics of the two methods, there are some clear similarities in the patterns of results. For both methods, players trained with the one hot hand input representation were able to score the lowest followed by players trained with the one hot state and hand and one hot state representations respectively. This implies that the better players are those that have a smaller input and are thus able to learn better **regardless of the training method**. This is likely due to the smaller and less complex networks that are generated and used as a result.
In addition, the players using the one hot hand representation were able to learn to make more matches per game in comparison with the other representations. This is again likely due to the reduced network size, but also an increased focus on the player's hand, meaning that it is more likely to discover the matching aspect of the game. However, these matches could also be a result of its greedier approach to taking cards. If the player takes the lower scoring cards more often, then the probability of matching them by chance increases due to the hand containing more cards of the same value. This may also be the reason why one hot state and hand players occasionally make more matches than average.

The players trained with the one hot state input representation are consistently the worst players, scoring the highest and matching the fewest. This is also likely due to its input size which results in a larger and more complicated network. However, this player consistency ends the most rounds and plays the shortest rounds. This could imply that the player's decisions are impacted by the additional state information it is provided with.

Whilst the performances (scores and matches) of the one hot state and hand players are consistent between algorithms, the behaviours differ. Under coevolution, the player's behaviour is a midpoint of the behaviours of the one hot hand and one hot state players. Under NEAT, the players display significantly

greedier behaviour, ending very few rounds and playing those rounds for longer. Against the random player, rounds contested by the NEAT players take on average take 9.51 turns longer to complete compared with the coevolution players.
This could be due to the differences in the structures/topologies of the networks from the training algorithms. The coevolution players have a fixed network size whilst the topology of the NEAT players changes during training. The initial network structures in this NEAT implementation consist of no hidden nodes and the NEAT algorithm aims to produce networks with simplistic topologies. This means that the NEAT players will have less complex networks which affects their training. This could also explain why the players trained under NEAT were able to consistently score fewer points and make more matches than those trained under coevolution.

7.4 Round Robin Tournaments

Until now, players have only been assessed against the random player, which is a player of admittedly poor quality. To further assess the quality of the trained players and also determine the best player overall, several round robin tournaments are played. These allow for players of similar or differing inputs and training algorithms to compete against each other, enabling further analysis of their performances.
In a match within a round robin tournament, two players will compete against each other across 100 games (50 pairs). Here, the ability of the player is determined by the number of games won, and in the event of a tie the average score is used to break it. Player wins are used here as a metric as the quality of the opponent is now a factor of the analysis.
Firstly, several round robin tournaments are played internally amongst like players, meaning the same input and training algorithm. This allows for the single best player of each type to be determined, allowing for the final round robin tournament to be vastly reduced in size. The results for these tournaments are available in Appendix D. A final round robin tournament was then played consisting of the 6 best players from the previous tournaments and the random player. The following tables contain the results of the final round robin tournament.

7.4.1 Results

	COEVO-Hand	**COEVO-State & Hand**	**COEVO-State**	**NEAT-Hand**	**NEAT-State & Hand**	**NEAT-State**	**Random**	**Average**
COEVO-Hand		92%	93%	63%	93%	99%	100%	90.00%
COEVO-State & Hand	8%		76%	21%	38%	89%	100%	55.33%
COEVO-State	7%	24%		22%	27%	67%	97%	40.67%
NEAT-Hand	36%	77%	78%		73%	95%	100%	76.50%
NEAT-State & Hand	7%	61%	73%	37%		91%	100%	59.83%
NEAT-State	0%	10%	31%	5%	8%		94%	24.67%
Random	0%	0%	3%	0%	0%	5%		1.33%

Table 8: Percentage of games won by players in the round robin tournament

	COEVO-Hand	COEVO-State & Hand	COEVO-State	NEAT-Hand	NEAT-State & Hand	NEAT-State	Random	Average
COEVO-Hand		140.53	174.50	111.70	112.34	146.18	130.19	135.91
COEVO-State & Hand	187.43		195.91	168.84	179.31	185.93	166.94	180.73
COEVO-State	221.33	220.59		219.06	221.11	216.28	207.08	217.58
NEAT-Hand	125.00	139.31	186.42		67.57	146.36	125.33	131.67
NEAT-State & Hand	162.51	167.46	201.82	85.98		163.33	153.07	155.70
NEAT-State	230.25	226.14	233.54	215.02	235.36		218.82	226.52
Random	283.54	287.45	283.14	287.56	287.91	280.97		285.10

Table 9: Mean scores of players in the round robin tournament

Player	Mean Win Percentage	Mean Score
COEVO-Hand	90.00%	135.91
NEAT-Hand	76.50%	131.67
NEAT-State & Hand	59.83%	155.70
COEVO-State & Hand	55.33%	180.73
COEVO-State	40.67%	217.58
NEAT-State	24.67%	226.52
Random	1.33%	285.10

Table 10: Results of the round robin tournament, ordered by mean win percentage

7.4.2 Analysis of results

From the statistical data against the random player and the results of the round robin tournament, it is clear that players trained with the one hot hand input are the better players. These players win the majority of their games against all other trained players, scoring on average 20 points fewer per game than the next best players. Players trained with the one hot state and hand input are then the next best players, followed by the players using the one hot state input, which struggle to win against the other trained players. However, all players are able to comfortably and consistently beat the random player, showing that they have all gained some understanding of the game.

There are minimal differences when comparing the performances of players with the same input trained under different algorithms. The coevolution one hot hand and one hot state players win more and score less when playing against their NEAT counterparts. Conversely, the NEAT one hot state and hand player wins more and scores less against the coevolution one hot state and hand player. This leads to the conclusion that the algorithms, as they are implemented here, produce similar results with regard to performance.

However, there are some noticeable differences in the behaviour that certain players exhibit. From the internal NEAT round robin tournaments, the one hot hand and one hot state and hand players scored significantly fewer points against like players than against the differing players in the final tournament (Appendix D Tables 29 and 30). This can also be seen when the NEAT one hot hand and one hot state and hand players play each other in the final round robin where they score far fewer points that against

any other opponents. This is likely as a result of the fitness calculation involving solutions playing games against themselves, thus they are able to perform better against players of similar behaviour.

Further, the coevolution trained one hot state player, whilst losing a majority of its games, causes its opponents, with exception of the random player, to score higher than they normally would. On average, when a player plays the coevolution trained one hot state player, they score 17.4% more than they do against other players. This is likely due to the additional game information it is given and thus it would be making moves to negatively affect the performance of the opposing player. This could also explain why the score of the random player doesn't increase as it does not have a plan/strategy for playing the game.

7.5 Greedier Random Player

Whilst the random player shows that the other players have learned to play Golf, it is difficult to judge how well that learning is as the random player is of poor quality. Most systems trained to play games are usually compared against that of a human opponent. However, this is difficult to achieve and draw reliable conclusions from as humans have a wide variety of abilities, which is shown here to affect the performance of the trained players. Other games such as Chess [23] or Go [9] have an established rating system to allow for the quality of the players to be assessed. The game Golf does not have such a system. The greedier random player (see Section 4.3) was therefore introduced to the final round robin tournament to compare the performances of the trained players against a better performing random player with some prior understanding of the game. The results are as follows:

	COEVO-Hand	**COEVO-State & Hand**	**COEVO-State**	**NEAT-Hand**	**NEAT-State & Hand**	**NEAT-State**	**Random**	**Average**
Win Percentage	38%	83%	91%	43%	86%	96%	100%	76.71%
Mean Score	155.85	153.45	172.19	144.99	149.54	159.02	146.60	154.52

Table 11: Win percentages and mean scores of the greedier random player

	COEVO-Hand	**COEVO-State & Hand**	**COEVO-State**	**NEAT-Hand**	**NEAT-State & Hand**	**NEAT-State**	**Random**
Win Percentage	58%	17%	9%	56%	13%	4%	0%
Mean Score	150.56	185.59	216.86	138.90	178.42	223.37	282.14

Table 12: Win percentages and mean scores of opponents when playing the greedier random player

Whilst this player was able to comfortably beat both the one hot state and hand and one hot state players, the one hot hand players of both algorithms were able to win a majority of the games against it. This shows that the one hot hand players are of a high quality as they are able to beat a player whose ability would be closer to that of human players due to hard coded decisions implemented into its design. It should also be noted that the behaviour of the coevolution one hot state player is still apparent in these results.

8 Evaluation

The success of this project centres around whether players are able to learn to play Golf. Therefore, as all players were able to improve upon their initial scores, this project can be viewed as being successful. In addition, the pattern of the training results for the coevolution algorithm corroborate those from existing literature, [15] and [16], which adds reliability to the conclusions drawn. Whilst the methodology implemented from existing literature required some adjustments, such as how the better player is determined and how players are updated, the general concept remained unchanged. Finally, the experimentation process resulted in the different inputs being designed which allowed for further analysis to be made, expanding the overall scope of this project.

There are, however, some areas of this project that could be improved upon. Despite the consistency of results, a minimal number of trials were run for each combination of input and algorithm (5 for each coevolution player, 3 for each NEAT player). This was due to the time and resources that each trial required, taking anywhere from 10 to 30+ hours to complete. Due to the size and complexity of the search spaces, the algorithm's performance and results can be affected by the initialisation of its solutions. Therefore, increasing the number of trials would increase the reliability of any conclusions drawn.

It is unlikely that the full potential of the NEAT algorithm has been fully explored due to the use of a small population size (30) and a low number of generations (50). This was again partially due to the time and resources that each trial needed to run. The scores of some of the trained players were still decreasing by the final generation, meaning it hadn't yet reached its optima. Further, only once during any trial did any population contain more than a single species of solution. This shows that these small parameters might have negatively impacted the performance of the NEAT algorithm, despite the positive results that it showed.
In addition, the NEAT players perform significantly better against similar players than they do against differing players. This is likely to be a result of their fitness calculation in which they play games against a copy of themselves. This shows that the player might be learning the behaviour of the opponent rather than the game which is not desirable.

This project did not collect or use any data which involved human players. Instead, a heuristic player was used to provide a consistent human-like performance, as human players vary in ability. Such human data could have been used to provide an indication of whether the trained players can play Golf at a human-level. However, gathering this data would require at minimum some interface to allow for human interaction with the game and a considerable amount of time to collect a sufficient amount of data.

9 Conclusion

To assist this conclusion, the initial research questions are revisited along with a further question posed from the experimentation process.

Is it possible for a player to learn to play the card game Golf by evaluating game states?
The results of the training clearly show that it is possible to train a player to play the game of Golf by learning to evaluate game states. All of the trained players, regardless of algorithm, were able to consistently score fewer points per game against a variety of opponents than the random player. In addition, all players were able to win the overwhelming majority of games played against said random player.

How well have the players learned? Are neuroevolution methods useful in this scenario?
When playing against the greedier random player, that has some understanding of the game, not all players were able to successfully compete against it. Similar to the conclusions drawn from [15], whilst

there is evidence of learning, the ability of the trained players is not human-level. However, some players were able to win the majority of the games against the greedier random player and perform better against the same opponents. This shows that it is possible to train some players using neuroevolution to a point where they are better than a heuristic player that has some understanding of the game.
However, due to a lack of points of comparison, both for human data and results from other machine learning techniques, the full extent of the learning made by the players is unclear.

How do the different neuroevolution techniques affect the trained players?
Both networks were able to produce players of similar performance for each input. Whilst NEAT players were able to score less against the random opponent, the coevolution players were competing with and beating some of them head-to-head.
Behaviour-wise, there were differences between the players. NEAT players had a tendency to play longer and end fewer rounds against the random player. This is evident of either a greedier behaviour or a lack of understanding of an end game. Coevolution players, on the other hand, were much more likely to end rounds and took fewer turns on average. This is likely due to the differing topologies of the network. The coevolution players' function approximator had a consistent hidden layer which may allow for a greater understanding of the game, compared to the NEAT players' which has very few hidden nodes. However, there is no clear evidence to show that these behavioural changes are significantly impacting the player's performance.

Does the input have a significant impact on performance?
What was discovered through experimentation and training was that the amount of information that a player has and the way in which it is represented has the biggest impact on player performance and behaviour. Players with smaller inputs were able to score lower, regardless of their neuroevolution algorithm, likely due to the smaller network that is generated as a result. The players with an increased input size conversely saw an increase of the scores they obtained. The restricting of state information also appears to result in players developing a greedier play-style with regards to what cards they take and how long they play for.
Conversely, those of larger input sizes with unrestrictive data tend to develop a contrasting play-style, resulting in much shorter rounds. The additional information provided can also affect how the player makes decision, as displayed by the coevolution trained one hot state player which negatively affects the scores of its opponents. However, given the superior results the restrictive inputs produce, it leads to the conclusion that the information about the player's hand is the most important information in the game of Golf.

Further research on this project could be done by improving upon the problematic areas address in Section 8. For example, an environment could be developed to allow for humans and the trained players to play games against each other to gather a better understanding of the extent of the players' learning how they perform against human players. The NEAT fitness algorithm could be improved upon by increasing the variety of opponents that players face to prevent it from specialising against similar players. In addition, the algorithm could be run with a larger population size for more generations to explore whether the speciation improves its performance. Finally, as the concept of coevolution has been shown to be effective for two differing card games (Golf and Gin Rummy [15]), players could be trained to play a wider variety of card games to better understand the general performance of the coevolution, and potentially NEAT, algorithms for game playing. This could include further games implementing the draw and discard mechanic or those that are entirely different, such as Hearts [24].

A Experimentation Tables of Results

Trial	Mean Absolute Error	r^2 Value	Training Time /s
1	205.22	1.854	0.875
2	233.06	1.741	0.872
3	221.20	1.081	0.970
4	252.76	1.926	0.885
5	216.51	1.793	0.872
6	189.89	1.889	0.892
7	219.64	1.835	0.893
8	150.73	1.874	0.866
9	169.59	1.909	0.884
10	240.07	1.824	0.876
Average	**209.90**	**1.773**	**0.889**

Table 13: Supervised training results with a numerical representation

Trial	Mean Absolute Error	r^2 Value	Training Time /s
1	504.46	0.287	0.997
2	1118.35	0.985	0.973
3	751.84	0.996	0.972
4	955.39	0.835	0.978
5	691.68	1.206	0.963
6	461.38	1.164	0.966
7	438.53	1.037	0.974
8	632.82	0.435	0.994
9	648.23	0.803	0.983
10	511.01	0.642	0.987
Average	**673.97**	**0.839**	**0.979**

Table 14: Supervised training results with a one hot value representation

Trial	Mean Absolute Error	r^2 Value	Training Time /s
1	2334.44	0.751	0.983
2	1046.56	0.732	0.983
3	1514.16	0.331	0.997
4	1014.97	0.436	0.995
5	1023.44	0.718	0.984
6	925.25	0.403	0.996
7	827.73	0.483	0.993
8	1486.69	0.422	0.993
9	1543.41	0.435	0.994
10	1013.37	0.813	0.983
Average	**1273.00**	**0.552**	**0.990**

Table 15: Supervised training results with a one hot deck representation

Epoch	Mean absolute difference in score
1	11.0
2	5.3
3	5.7
4	1.2
5	8.1
6	2.8
7	11.5
8	0.4
9	8.0
10	5.1
Average	**5.91**

Table 16: Mean absolute difference in the scores obtained by two random players in an epoch

B Coevolution Tables of Results

Player number	Mean Score	Percentage of games won	Mean matches per game	Mean turns per round	Percentage of rounds ended
1	123.33	57.30%	6.86	44.80	11.75%
2	151.98	94.38%	6.16	31.94	50.34%
3	155.08	54.08%	5.94	26.16	54.08%
4	185.01	65.98%	2.49	18.80	44.38%
5	147.03	55.22%	2.38	23.77	38.67%
Average	**152.49**	**65.39**%	**4.77**	**29.09**	**39.84**%

Table 17: Results of the one hot hand players against their training player

Player number	Mean Score	Percentage of games won	Mean matches per game	Mean turns per round	Percentage of rounds ended
1	118.18	100.00%	7.27	27.85	11.65%
2	155.46	100.00%	6.09	22.59	46.60%
3	140.78	100.00%	6.57	21.65	51.49%
4	161.06	99.96%	2.86	19.75	59.87%
5	128.10	99.98%	2.17	21.56	52.54%
Average	**140.72**	**99.99**%	**4.99**	**22.68**	**44.43**%

Table 18: Results of the one hot hand players against the random player

Player number	**Mean Score**	**Percentage of games won**	**Mean matches per game**	**Mean turns per round**	**Percentage of rounds ended**
1	215.23	56.78%	1.83	16.31	71.90%
2	192.75	58.98%	1.57	20.79	52.08%
3	208.67	78.78%	1.35	14.92	35.99%
4	181.81	83.08%	2.69	19.38	57.85%
5	217.34	72.12%	1.23	16.44	38.55%
Average	**203.16**	**69.95**%	**1.73**	**17.57**	**51.27**%

Table 19: Results of the one hot state and hand players against their training player

Player number	**Mean Score**	**Percentage of games won**	**Mean matches per game**	**Mean turns per round**	**Percentage of rounds ended**
1	209.68	96.40%	1.81	16.21	78.60%
2	180.39	99.68%	1.53	20.37	57.94%
3	188.02	99.28%	1.47	17.76	72.01%
4	172.27	99.82%	2.71	19.38	63.82%
5	198.89	98.50%	1.16	18.91	66.78%
Average	**140.72**	**98.74**%	**1.74**	**18.53**	**67.83**%

Table 20: Results of the one hot state and hand players against the random player

Player number	**Mean Score**	**Percentage of games won**	**Mean matches per game**	**Mean turns per round**	**Percentage of rounds ended**
1	218.06	59.60%	1.80	13.16	56.85%
2	231.02	59.28%	1.59	13.23	50.92%
3	239.89	59.88%	1.45	13.72	51.71%
4	229.82	68.60%	1.83	14.24	49.08%
5	246.23	56.40%	1.55	14.35	48.10%
Average	**233.00**	**60.75**%	**1.64**	**13.74**	**51.33**%

Table 21: Results of the one hot state players against their training player

Player number	**Mean Score**	**Percentage of games won**	**Mean matches per game**	**Mean turns per round**	**Percentage of rounds ended**
1	205.41	97.74%	1.84	14.27	87.15%
2	220.56	94.38%	1.70	14.64	85.66%
3	231.10	89.74%	1.53	15.15	83.44%
4	223.02	93.00%	1.89	15.85	80.88%
5	240.14	84.26%	1.61	16.10	79.71%
Average	**140.72**	**91.82**%	**1.71**	**15.20**	**83.37**%

Table 22: Results of the one hot state players against the random player

C NEAT Tables of Results

Player number	Mean Score	Percentage of games won	Mean matches per game	Mean turns per round	Percentage of rounds ended
1	126.25	100.00%	3.96	26.94	17.35%
2	144.04	99.96%	3.42	26.36	20.31%
3	121.25	100.00%	8.15	26.10	9.59%
Average	**130.51**	**99.99**%	**5.18**	**26.47**	**15.75**%

Table 23: Results of the one hot hand players against the random player

Player number	Mean Score	Percentage of games won	Mean matches per game	Mean turns per round	Percentage of rounds ended
1	173.29	99.82%	2.83	29.02	4.74%
2	167.07	99.90%	2.33	29.23	2.58%
3	153.67	99.92%	1.64	25.88	12.09%
Average	**164.68**	**99.88**%	**1.74**	**28.04**	**6.47**%

Table 24: Results of the one hot state and hand players against the random player

Player number	Mean Score	Percentage of games won	Mean matches per game	Mean turns per round	Percentage of rounds ended
1	211.15	95.68%	2.04	19.72	58.94%
2	230.04	90.30%	1.49	18.90	62.62%
3	212.17	95.82%	1.94	18.89	53.87%
Average	**217.79**	**93.93**%	**1.82**	**19.17**	**58.48**%

Table 25: Results of the one hot state players against the random player

D Internal Round Robin Tournament Results

Player	Mean Win Percentage	Mean Score
5	73.75%	134.72
3	57.75%	145.80
1	55.25%	140.33
4	34.25%	172.30
2	25.25%	168.36

Table 26: Results of the coevolution one hot hand round robin tournament, ordered by mean win percentage

Player	Mean Win Percentage	Mean Score
4	62.50%	192.92
2	58.25%	200.13
3	56.00%	203.35
5	36.75%	214.91
1	34.00%	226.48

Table 27: Results of the coevolution one hot state and hand round robin tournament, ordered by mean win percentage

Player	Mean Win Percentage	Mean Score
1	77.50%	213.53
2	55.75%	233.23
4	45.25%	237.68
3	43.75%	240.30
5	25.25%	250.47

Table 28: Results of the coevolution one hot state round robin tournament, ordered by mean win percentage

Player	Mean Win Percentage	Mean Score
3	75.50%	100.92
1	53.00%	116.02
2	19.00%	138.95

Table 29: Results of the NEAT one hot hand round robin tournament, ordered by mean win percentage

Player	Mean Win Percentage	Mean Score
3	98.50%	84.77
2	43.50%	133.10
1	8.00%	151.37

Table 30: Results of the NEAT one hot state and hand round robin tournament, ordered by mean win percentage

Player	Mean Win Percentage	Mean Score
1	58.50%	224.58
3	57.50%	223.77
2	32.00%	241.93

Table 31: Results of the NEAT one hot state round robin tournament, ordered by mean win percentage

References

[1] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 3, no. 3, pp. 210–229, 1959.

[2] P. Langley, "The changing science of machine learning," *Machine Learning*, vol. 82, no. 3, pp. 275–279, 2011.

[3] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1701–1708, 2014.

[4] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*, pp. 173–182, 2016.

[5] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, pp. 160–167, ACM, 2008.

[6] C. Voyant, G. Notton, S. Kalogirou, M.-L. Nivet, C. Paoli, F. Motte, and A. Fouilloy, "Machine learning methods for solar radiation forecasting: A review," *Renewable Energy*, vol. 105, pp. 569–582, 2017.

[7] C. Perlich, B. Dalessandro, T. Raeder, O. Stitelman, and F. Provost, "Machine learning for targeted display advertising: Transfer learning in action," *Machine learning*, vol. 95, no. 1, pp. 103–127, 2014.

[8] M. Bowling, J. Fürnkranz, T. Graepel, and R. Musick, "Machine learning and games," *Machine learning*, vol. 63, no. 3, pp. 211–215, 2006.

[9] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[11] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[12] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[13] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[14] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 1, pp. 25–41, 2017.

[15] C. Kotnik and J. K. Kalita, "The significance of temporal-difference learning in self-play training td-rummy versus evo-rummy," in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 369–375, 2003.

[16] J. B. Pollack, A. D. Blair, and M. Land, "Coevolution of a backgammon player," in *Artificial Life V: Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems*, pp. 92–98, Cambridge, MA: The MIT Press, 1997.

[17] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[20] J. Obenaus, "Implementing a doppelkopf card game playing ai using neural networks," September 2017.

[21] T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[22] A. McIntyre, M. Kallada, C. G. Miguel, and C. F. da Silva, "neat-python." `https://github.com/CodeReclaimers/neat-python`. Accessed: 2019-01-29.

[23] M. E. Glickman and A. C. Jones, "Rating the chess rating system," *CHANCE-BERLIN THEN NEW YORK-*, vol. 12, pp. 21–28, 1999.

[24] J. McLeod, "Card games: Hearts." `https://www.pagat.com/reverse/hearts.html`, 2018. Accessed: 2018-11-15.