A Neuroevolutionary Approach to Draughts Playing Agents

Student Name: T. P. Nguyen

Supervisor Name: S. Dantchev

Submitted as part of the degree of BSc Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

Abstract —

Background

Neural Networks are commonly used to solve a variety of complex problems, but doing so involves a magnitude of manually labelled data. Through neuroevolution (combining Genetic Algorithms and Neural Networks), this issue can be solved by creating a system that can grow in ability over time. It's ability is measured by allowing the system to play against itself in the game of Draughts.

Aims

Neural Networks are commonly trained with back-propagation, but the aim of this work was to explore whether it was possible to also train a neural network by using genetic algorithms to manipulate its weights. The effectiveness of this approach was measured when it was trained to play Draughts.

Method

The method started with designing a feed-forward neural network that took as input, the state of a checker board, and would output an evaluation of the players advantage. When combined with Monte-Carlo Tree Search and a Draughts framework, the system was given the ability to play Draughts. Genetic algorithms were used to adjust the weights and biases of the neural network.

Results

Overall, the neuroevolutionary approach was shown to learn and improve over time. However, the wide scope of the adjustments afforded may have stagnated the learning rate. Mutations consistently improved the quality of the system. Crossovers showed promise, providing a wide range of potential results but on average suggested negligible improvement.

Conclusion

Neuroevolution can be considered as an option to create a draughts-playing AI, but the robustness of the system is quite volatile. To counteract this, extra precautions should be taken in producing a crossover mechanism that is both safe but exploratory as this could otherwise negatively impact the learning rate.

Keywords — Artificial Intelligence, Machine Learning, Neural Networks, Neuroevolution, Genetic Algorithms, Draughts, Checkers, Crossover, Monte Carlo Tree Search

I INTRODUCTION

Contemporary Machine Learning commonly revolves around the use of Neural Networks to deduce abstract relationships, solving different types of problems. This is made possible through the use of a back-propagation algorithm that allows the neural network to learn by comparing errors between it's predicted outcome against the actual outcome. (Rumelhart et al. 1986) However, this involves a multitude of data, generally classified with some manual labour. The motivation behind this project is to explore the effectiveness of genetic algorithms as an alternative to train a

neural network, which could allow the system to learn without using any data. The effectiveness of this approach is tested by its ability to play Draughts.

A Draughts

English Draughts (also known as American Checkers) is a popular 2-player board-game played on an 8x8 chess board (also known as a checker board/draughts board). Players choose a colour and begin with 12 pawns of their respective colours, which are placed on the dark-coloured squares of the board. Beginning with the black pawns, each player takes a turn to move a pawn diagonally by one square. In the event that a pawn reaches the opposite side of the board from where the piece originated, it is promoted to a king. Kings have the ability to traverse backwards in the same diagonal motion as pawns. Assuming that there is space upon landing, a piece (pawn or king) also has the option to capture their opponents piece by moving two consecutive diagonal squares, where the opponents piece is situated immediately opposing the players piece. Multiple pieces can be captured consecutively in a single turn if the moves afford the scenario. A player wins by capturing all of their opponent's pieces. A player loses by having all of their pieces captured. A draw occurs when both players agree to draw after a three-fold repetition (where both players take three moves to simulate the same position), or a player has pieces on the board but cannot move any of them. A draw can also be forced when both players fail to capture a piece after a set amount of moves.

B Neural Networks

Neural Networks (NNs) are non-linear statistical tools which link inputs and outputs adaptively in a learning process similar to how the human brain operates. Networks consist of units, described as neurons, and are joined by a set of rules and weights. The neurons are defined with some characteristics and appear in layers. The first layer is defined as the input layer, and the last layer is the output. Layers between the two aforementioned are described as hidden layers. Data is analysed by their propagation through the layers of neurons, where each neuron manipulates the content of the data by some manner. Learning takes place in the form of manipulation on the weights connecting the neurons. This allows it to model complex relationships between inputs and output, and also allows it to find patterns in the data.

C Genetic Algorithms

A subset of evolutionary algorithms in general, Genetic algorithms (GAs) are a group of search techniques used to find exact or approximate solutions to optimisation and search problems. The methodology is inspired by evolutionism theory; individuals are created based on the improvement of their ancestors' genetic information (genome). Genomes are analogical to a data structure, most commonly a 1D array. Evolutionary algorithms typically consist of a population of agents, a tournament selection process (also known as a fitness function), and the introduction of probabilistic mutation on genomes. This in turn allows evolutionary algorithms to create genomes that increase in quality over time with regards to the tournament. Genetic algorithms distinguish themselves from evolutionary algorithms through their use of algorithmic crossover mechanisms, where two parent genomes are combined together to create offspring. The use of evolutionary algorithms to train neural networks is described as a type of machine learning called Neuroevolution.

D Aims

Whilst the use of evolutionary algorithms and neural networks have been explored independently of one another to create draughts players, the intention is to explore the effectiveness of the combined methods in developing an artificially intelligent draughts playing agent.

D.1 Minimum Objective

The minimum objective was to implement a feed-forward neural network, and a genetic algorithm that interacted with the neural network, allowing it to improve over time. This would need to be connected to a Draughts Interface that allowed the neural network to play Draughts. This was achieved by having each component implemented manually in Python and the NumPy library.

D.2 Intermediate Objective

The intermediate objective involved the implementation of an interface to play against the trained result, and the implementation of a Monte-Carlo Tree-Search (MCTS) algorithm. Also, as the genetic algorithm included a tournament mechanism which would play games simultaneously, this needed to be implemented to run in parallel. The implementation of the tournament mechanism was made possible via the use of a round robin approach and the use of the multiprocessing Python library. MCTS was successfully implemented, which improved the quality of the moves the system made.

D.3 Advanced Objective

The advanced objective consisted of producing a system showing indications that it was learning over time. Measurements were taken to determine whether the genetic algorithm improved the neural network, and which properties of the genetic algorithm made it possible. Measurements were taken at every generation that calculates the distribution of the different components, which is explained in more detail in Results.

The remaining sections of the paper discusses related and relevant work, the proposed solution, results and an evaluation of the project. The paper is then wrapped up with a conclusion, discussing the implications of the project and potential future work.

II RELATED WORK

Draughts was historically used as a testing ground for artificial intelligence and computational capabilities since introduction of computers. This was due to the relatively vast search space, making the game difficult to brute-force, until recently. Research has shown that it is possible in theory to solve the game (Schaeffer & Lake 1996) which was then later *weakly-solved* by the same researchers with the advent of increased computational capabilities. (Schaeffer et al. 2007). Weakly-solvable is defined such that "the game-theoretic value for the initial position has been determined, and that a strategy exists for achieving the game-theoretic value from the opening position." (Allis 1994) The premise of this paper was not necessarily to solve the game itself, but

to rather see whether neuroevolution was a method that, when measured via its ability to play the game, shows that it can evolve in capability over time.

A Arthur Samuel

An early design of machine learning to play Draughts by Samuel (Samuel 1959) revolved around the continuous improvement of a linear combination of heuristics. These heuristics represented different strategies players considered when playing Draughts, and when combined, acted as an evaluator function for a state of the checker board. Some of these heuristics included the number of pieces on a side (where one side had a advantage given that they had more pieces), and the number of pieces positioned along the central diagonal. His evolutionary strategy consisted of agents, each of which represented coefficient multipliers for the heuristics (initially random), and new agents are formed via the manipulation of its successor's coefficients with subtle mutations.

Samuel's game tree-search algorithm primarily revolved around an early concept of the mini-max algorithm. This algorithm took as input the present state of the checker board, expands the move-tree to some ply-depth, and returns the best move to choose. A ply refers to the number of moves in advance. Ply-depth refers to the depth of the tree based on the number of moves in advance. Mini-max can be described using the following instructions:

1. A game tree is expanded from the root node, where the root node represented the current state of the game.

2. Each node represents a state of a checker board. Child nodes represents a future state of the checker board. Child nodes can only be produced from the parent node; i.e. a move from a parent node will produce a new state represented in the child node.

3. Child nodes are expanded until some depth was reached. From there, an evaluation function is used to calculate the value of the leaf nodes at this depth. This evaluation function would produce a value that indicates a player's advantage at that state.

4. The best potential position for a given player is deduced from the nodes at this stage, which is quantified with some value.

5. The best value propagates upwards until it reaches to a child of the root node. From here, the best move can be deduced, corresponding to the state with the highest value.

There are trade-offs to this tree-search algorithm; asymptotic complexity of mini-max is exponential; it being $O(x^y)$ with x being the branching factor, and y being depth. Alpha-Beta pruning, an optimisation to mini-max, consisted of pruning nodes which are already recognised to be disadvantageous. Whilst Alpha-beta pruning did improve the running time of the algorithm (having a best case of $O(\sqrt{x^y})$, the worst case complexity of the algorithm remained $O(x^y)$. This exponential growth is the Achilles' heel to the mini-max approach, which made it less attractive when considering tree-search algorithms for the system.

The performance of Samuel's system is, however, fundamentally handicapped by the quality of the heuristics he devised; i.e. the system can only be as good as whatever Samuel considered as a heuristic to consider when deciding what move to make.

B Blondie24

Samuel's system was the inspiration for Chellapilla and Fogel's Blondie24 AI, where they replaced Samuel's linear combination with a NN. (Chellapilla & Fogel 1999) Blondie24 used an evolutionary algorithm, using a population of fifteen agents $i = 1, \ldots, 15$, each of which consisted of two properties. The first property were weights of the neural network, and the second consisted of a self-adaptive parameter vector, τ_i, which determined how random the mutations are when the agent i evolved.

In each generation, every agent played each other in a round robin style tournament. Agents played an equal number of games, and successors were chosen based on their performance in the tournament. The best agents were chosen to mutate themselves, which produced new agents in the chance that they performed better than their predecessors. Mutation was performed using the formula $\tau_i(j) = \tau_i(j) \cdot e^{\mu}$, where $j = 1, ..., N_w$ represented individual weights of the neural network w, and μ represented a random decimal in the range of $[0 - 1]$. Like Samuel's program, Blondie24 also used Mini-Max with alpha-beta pruning.

Blondie24's neural network structure consisted of a $\{32, 40, 10, 1\}$ set, such that the input layer consisted of 32 nodes, and 1 output node. The output node represented a classifier value in the range of [-1, 1], which represented a particular advantage for a given side of the board. One issue with this particular method was that spatial awareness could not be immediately interpreted as it took an immediate input of the positions on the board. This would make it inherently more difficult for the neural network to generate heuristics based on spatial awareness. Furthermore, crossover mechanisms were not considered (and therefore cannot be classified as using genetic algorithms, hence their classification of evolutionary algorithms).

The motive was not to repeat the work produced by Chellapilla and Fogel, but sections of the best parts of their system were extracted and manipulated to fit the solution described in this paper, such as the neural network structure and the mutation formula.

Blondie24 was influential in the field of machine learning during its time, which led to various studies on the theory behind their success. A notable example was the series of publications by Al-Khateeb and Kendall, where they explored in detail the different features that comprise Blondie24. One of their studies concerning the effects of the ply depth, concluded that having a higher ply unsurprisingly increases the overall performance of the player, and that agents trained at a higher ply performed significantly worse when playing at a lower ply. More importantly, the results in the paper suggested that training with a 4-ply produced the best value function to start with during the learning phase, and then extending the ply for evaluations. (Al-Khateeb & Kendall 2012) This was considered in the final implementation, where the system was tested with ply depths around this value. Their study on the influence of tournaments in Blondie24 concluded that the league structure (commonly seen in standard tournaments) seemed more appropriate to use as opposed to the round robin tournament. (Al-Khateeb & Kendall 2009) Round Robin tournaments refers to having each participant play every other participant. This was however more difficult to set up in parallel, which was a large contributing factor in the runtime of the system.

C Return of "Genetic" Algorithms

Since Blondie24, there has been an increasing number of published papers describing the promising utility of GAs and NNs in general. GAs have been shown to evolve weights for a

convolutional-NN in a classification task better than back-propagation, in nearly all situations, other than when the number of generations generated for NNs are small. (Perez n.d.) Another research showed GAs ability to compute optimal structures for deep NNs, using an image recognition dataset as the benchmark for competitiveness against the networks. (Xie & Yuille 2017)

However, one recent paper by Clune et al. returned to the use of GAs as the premise to train especially deep convolutional networks with success. (Such et al. 2017) Their neuroevolutionary system was used to play popular Atari games on the OpenAI framework, and was found to perform as well as other contemporary evolution strategies and reinforcement learning algorithms.

The success of the program was based on their system's ability to perform safe mutations, a technique that revolved around measuring the sensitivity of the network to changes on some of the weights. (Lehman et al. 2017) This afforded the system to evolve deep NNs (In this particular case, containing over 100 layers and over four million weights and biases), in a manner that allowed random but safe and exploratory searches. Their safe mutation exploited the knowledge of the NN structure, by retaining sampled experiences of the network (inputs that were fed into the network and its outputs). This allowed the weights to be manipulated by comparing their significance in the neural network with the output, by using the sampled experiences.

While they do not consider the use of crossover techniques in their paper, they briefly mention that it would very well be possible to produce a safe crossover method that exploits structures of a NN, inspired by their research produced on safe mutations.

Unfortunately, it was unrealistic to apply the same techniques described in their paper into the proposed system. The machine used as the base (consisting of hundreds to thousands of GPU cores) was of a magnitude greater than what was available for the task at hand, but a more simplistic and adapted model can be produced using similar ideas.

A popular trend within the contemporary research in this section was the shared absence of the use of crossover algorithms. One paper describes the use of different crossover algorithms on NNs having no clear advantage over each other, in comparison to a uniform crossover (uniform crossovers are also commonly defined as the textbook crossover) which disrupted the overall performance of the NNs. This could be due to building blocks being disrupted as chunks of the sequential weights are swapped out of each other (Emmanouilidis & Hunter 2000). It was important to consider this when devising a crossover method, such that the effect it has on the NN should be subtle enough that it should pose similar results to the network prior to crossovers.

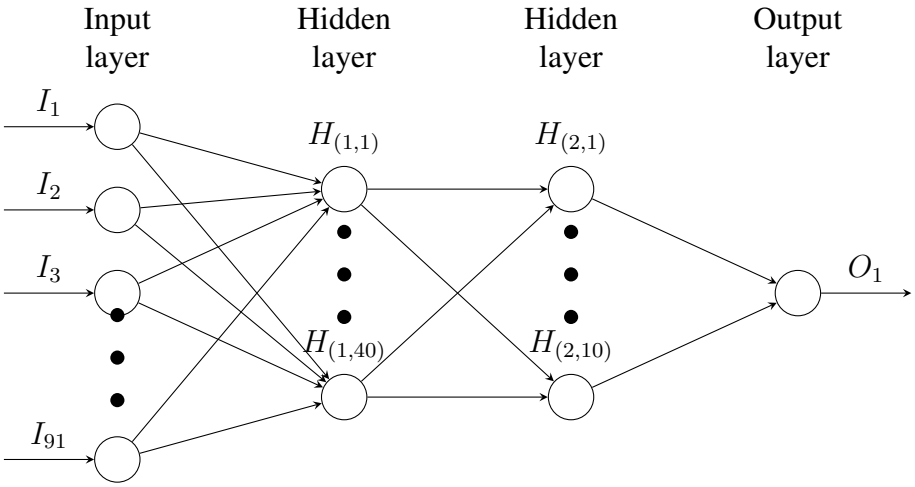
III SOLUTION

This section describes the implementation process of the system. Implementing the checker board interface is not included in the discussion as it is not one of the objectives. However, the sections below are considered under the premise that the interface was built prior to their development.

A Evaluation Function

A feed-forward multilayer neural network was used to evaluate the board. The network contained 4 layers; the input layer consists of 91 nodes, with the output node having 1. Hidden layers had 40 and 10 nodes respectively, and the dimensions are based on the success of Blondie24's model (Chellapilla & Fogel 1999). Figure 1 shows a visual representation of the network structure.

Figure 1: The chosen neural network model. The preprocessed values of the checker board are used as input. An output was produced after propagation that ranged from [-1,1].



The inputs of the network were a pre-processed 1D array of the checker board. Pre-processing the input began with retrieving the positions of the pieces on the checker board in the form of an 1D array and calculating all possible individual subsquares of the board. A subsquare is a square subsection of the board. The subsquares kernel size range from a 3x3 to 8x8. Figure 2 visualises the retrieval of the subsquares. With the kernels, the number of pieces in each subsquare are summed up. A new array was formed, comprising of the sums of the different kernels that make the board. The resulting array (representing 91 entries) is the preprocessed input, which was fed to the neural network.

To create values for the subsquares, the black pawns were weighed with a value of $v = 1$, and white pawns as $-v$. It is commonly understood that a king is worth more than a pawn piece, but its precise value advantage is disputed. This could have been decided by the system such that it evolved to understand that a king was eventually worth more than the pawn. however, for the sake of simplicity, a king's piece value was weighted at $1.5v$. This value was chosen based on the results produced by Al-Khateeb's research on piece values, where evolution on the piece differentials plateaued towards the value 1.5. (Al-Khateeb & Kendall 2010).

Weights w of a neuron are multiplied by their input value I, summed with their bias B, and are passed through an activation function $O = f((I * w) + B)$, to become an input for another neuron. Activation functions, when visualised, usually have a sigmoid curve, but may also take the form of other shapes. Common characteristics of activation functions include values which are monotonically increasing, continuous, differentiable and bounded.

The choice for the activation function is tanh (a type of sigmoidal function), shown in figure 3. This facilitated the ability to simulate a zero-sum formula (for a range of [-1 to 1]) due to its symmetric properties around the origin.

Results are cached, such that evaluations are saved and then can be recalled immediately, in the event that the same input was requested to be evaluated again. This showed varying levels of time improvement based on the ply. As the ply increased, the size

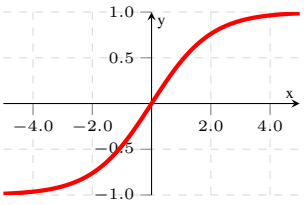


Figure 3: Graph of tanh function $f(x) = \frac{2}{1+e^{-x}} - 1$.

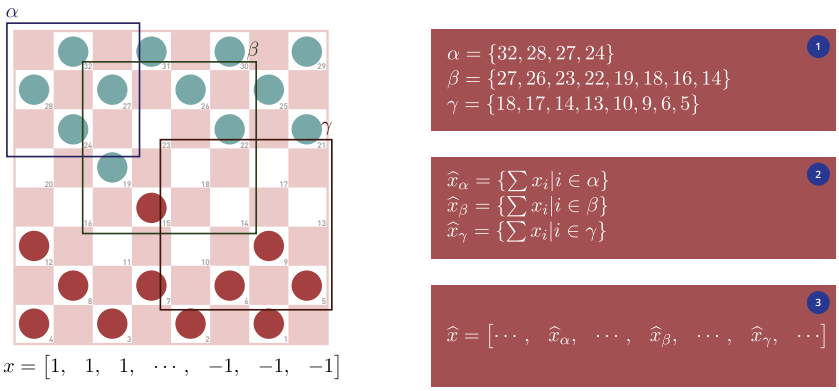


Figure 2: A diagram visualising the processing of subsquares on board (represented as the 1D array x) to create the array $\widehat{x}$. Each index in x represents a position on the board, such that each item in x represents a value of the piece on the board. α, β and γ are examples of 3x3 and 4x4 subsections.

of the state space increased, and therefore the chances of recalling the same set of inputs decreased. The cache was also used for our safe mutations algorithm, described later on in Section C.3.

The neural network was implemented manually via NumPy to allow direct access to the weights, allowing the genetic algorithm access to manipulate them.

B Tree-Search Algorithm

Tree-search algorithms allows the system to consider the future positions by allowing the neural network to evaluate states of the game further ahead from its current position. Initially, the tree-search algorithm implemented was the same mini-max approach as in Samuel's program (Samuel 1959), but was then progressed to a modified Monte-Carlo Tree Search (MCTS). A basic MCTS differs from mini-max where future moves are probabilistically played to the end of the game. It acts as a sampling of possible outcomes, and does not depend on an evaluation function at all. A survey found that MCTS can dramatically outperform mini-max based search engines in games where evaluation functions are otherwise difficult to obtain (Browne et al. 2012). MCTS was played in rounds which consist of four operations:

- Selection: A selection strategy was played recursively until some position was reached.
- Play-Out: A simulated game was played. Again, a basic form of MCTS would play until an end-game was reached.
- Expansion: A node was added to the tree.
- Back-propagation: The result of the game was back-propagated in the tree of moves.

These rounds are repeated until a certain period of time was met or a certain number of rounds are computed. Once finished, a node was chosen based on its total probability of reaching a winning outcome.

MCTS-EPT (or MCTS with early playout termination) modifies the MCTS random play-out approach. Instead of allowing the random moves play to the end of the game, the number

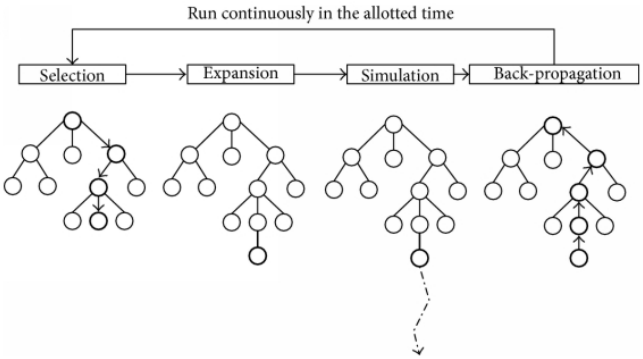


Figure 4: A diagram describing the MCTS process.

of moves traversed are capped and an evaluation function (which was the neural network) was applied at that future state instead. (Lorentz 2016) The model was then adapted such that when the number of possible moves was less than 4, the ply would extend by the number of moves. This reduced the need to depend on random playouts of moves to the end state, and ensured that the quality of the moves was contingent on the evaluation function. For the results, we use the 1,3 and 6 ply to determine whether the system played consistently based on their depth. As opposed to using a time threshold, the number of MCTS rounds r was based on the ply p, such that $r = 300p$. This allowed the system to process moves in a manner that the performance of the system was hardware-independent, as faster machines could have processed more rounds within the same fixed set of time.

C Genetic Algorithms

The genetic algorithm (GA) was the premise of the system's learning strategy. GAs are used to train the neural network. The various algorithms that form the collection of GA strategies are discussed below. For the system, a population size of 15 was chosen, due to the restraints on available computational performance.

C.1 Population Generation

In genetic algorithms, the population serves as a base that allows agents in the pool to play each other. Every generation has its own population of agents, and the population size was consistent across the generations. The initial population consisted of randomly generated weights and biases of the neural network, with values from [-1,1].

For a population size of 15, the next generation was created using the best five agents from the current generation (discussed in *C.2 Tournament Selection*.) They continued to play in the next generation; this strategy is described as elitism. The next eight players were generated through the use of crossover strategies (see *C.4 Crossover Strategy*). The weights of the 1^{st} and 2^{nd} place agents are used as inputs to the crossover strategy and generated 4 off-springs. Two were reciprocal crossover representations of each other, and the other two were directly mutated from the parents themselves. Another 4 children are created using the same strategy, with the 2nd and 3rd agent's weights. The remaining two were direct mutations of the 4th and 5th place agents.

$$w_n = w_p + \frac{m}{\sqrt{2 * \sqrt{K}}} \tag{1}$$

Figure 5: The mutation formula, where w_p is the current weight, K represents the number of weights and biases in the neural network, and m representing a random floating-point in the range of [-1,1].

C.2 Tournament Selection

To sort the quality of the players in a population, a tournament selection process was deduced. This allowed the best players who would continue to play in the next generation to be chosen.

Each agent in the population plays a minimum of 5 games as Black, against randomly selected opponents. Each game lasts a maximum of 100 moves from both players. If a winner was not deduced at this stage, a draw was called. Draws were also called when there was a three-fold move repetition from both players. A win was worth 2 points, a draw being none and a loss being -1 point. Both the agent and its opponent received a score from the game. Scores were then tallied up at the end of the tournament. Players were sorted by the number of points they scored. The best players had the highest number of points.

All the games were run in an 'embarrassingly parallel' fashion through the use of Python's multiprocessing library. Whilst the league structure was found as the better approach (Al-Khateeb & Kendall 2009) it would increase the overall running time as the initialisation of some games are dependent on the outcomes of other games. This would lead to a longer overall running time even though fewer games are played, whereas the embarrassingly parallel method allowed every game to play simultaneously.

C.3 Coefficient Mutation

To create variation between agents and their offspring, statistical anomalies were made through the use of mutations. This was used as one of the learning mechanisms that helped change the deciding factors of the neural network. Weights and biases of an agent's neural network would increment by a random value that was created using the equation in Figure 5.

Similarly to the activation function, the weights would have a soft maximum of [-1, 1]. This consequently meant that the mutation was not controlled, but rather dependent on the number of weights in the system. The more weights in the network implies a less significant mutation. A novel safe-mutation, inspired by Uber Lab's approach (Lehman et al. 2017) was also used. A subset of historical neural network calculations was used as a premise to guide the mutation. The method is as follows:

1. With the current set of weights w, choose a subset of pre-calculated input and output tuples $\phi_w = (I, \lambda_{I,w})$ where I represents the input values of the network, and $\lambda_{I,w}$ as the output.
2. Generate a perturbation y of the weights calculated using the formula in Figure 5. Use y as a basis of a neural network.
3. Using the inputs I in ϕ_w, calculate a new set of input and output tuples $\phi_y = (I, \lambda_{I,y})$.
4. Calculate the divergence $\delta = \lambda_{I,y} / \lambda_{I,w}$.

5. Calculate the quality of y based on the number of inequalities $\delta \geq 1$.

6. Repeat parts 2-5 some n times, keeping note of the best y.

The intuition behind this was that if ϕ_w was a subset of moves that led to a winning game, then the manipulation of the weights was left open as long as it can replicate the performance of winning games.

C.4 Crossover Strategy

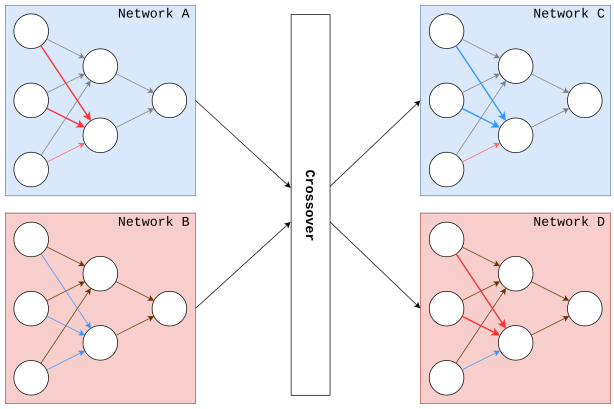


Figure 6: A diagram visualising the crossover process.

The distinct learning mechanism unique to genetic algorithms was the use of crossovers. This combines the traits that build two parent agents to create children. This scenario the weights and biases of the parent's neural networks was used.

Two off-springs were created from a pair of parents, with each offspring being the reciprocal crossover of each other. A random node from the dominant parent was chosen. A dominant parent was the agent who gained more points in the tournament round than the other (who would be the submissive parent). If a dominant parent could not be deduced, it was randomly chosen.

Once a node was chosen, a subset of weights and biases that were fed into the chosen node were swapped with the submissive parent's values at the same positions. As values in the weights can range from -1 to 1, dominant input weights are values that are greater than 0.75 or less than -0.75. Figure 6 visualises the process. In the diagram, Network A and B are parents, creating reciprocal offspring networks C and D. A was the dominant network, and A's node with the red inputs was chosen. The weights and biases of the red inputs were swapped with the values at the same position in network B. Network C was formed with Network B's dominant weights, with the rest being composed from A. Network D was the reciprocal, having Network A's dominant weights and the rest from B.

The crossover should create a subtle modification to a neural network, by swapping a small subset of weights and biases at a time. Having a more dramatic crossover could potentially change the structure of the network flow, reducing its effectiveness.

D Testing

Each component of the system, such as the neural network, MCTS, and genetic algorithms, were built in their own classes. This helped to achieve two things; unit testing, and modularisation. The use of unit testing (performed per class) helped to verify that individual components worked as expected, especially with a project of this scale. Components that are shown to work are then added to the system. Training the system required time (discussed further in Figure 12), and realising errors was a notably laborious task. Modularisation of the code allowed for quick deployment of new features, such as migrating tree-search algorithms.

E Evaluation Method

The success of the research question was measured using various statistics. At the end of a given generation, the growth of performance was measured by using the generation's champion. When a new champion was generated, it was played against the previous 5 champions from earlier generations. 6 games were played for each previous champion, with 3 being as Black, and 3 being White. A mean score was calculated from those 6 games. The overall performance of the current champion was the mean of the means of the 5 sets. This value would represent the learning value. A cumulative chart of the learning values represent the learning rate.

A positive learning value can be deduced when the mean of means were greater than 0. Point scores for the champion games were measured by 1,0,-1 where a Win counts as 1 point, 0 for a draw, and -1 for a loss. The weights were scaled differently to the regular tournament in order to portray an even distribution of the learning rate.

The genomic properties that were involved in creating the champion were also accounted for. This was combined with the score the champion produced from the learning value to determine the likelihood of creating a champion based on the genomic properties, and the score distribution it was likely to produce.

The end player, representing the champion at the last generation, was used to create comparison games against its ancestors. This player was to be first tested against an agent who was choosing random moves to verify that the system was not playing in some random fashion. Measurements from these games, alongside its champion scores were used as evidence to determine whether the agent was learning over time, thus leading to an answer to the research question.

Other measurements include the number of moves played in a game, and the CPU time utilised for system training. These statistics measured the success of the tree search implementation, and its relationship with the CPU timing. The two statistics were then correlated against each other to determine any spurious relationships between the two.

F Tools

The system was built using Python 3.6 and the `NumPy` library. Initial runs operated on a 1 and 2-ply load to determine the stability of the system. As the memory usage of the system was negligible, it was not considered when choosing hardware. This was run on a Linux (Ubuntu) machine containing a 4-core Intel i5 6600u processor. Development and debugging was also handled on this machine. Once testing has shown to be stable, the system was then configured to run on Durham's MIRA distributed system (Debian). MIRA contains 4 Intel Xeon E7-8860 CPUs (Each CPU contains 16 physical cores running in 2.2Ghz, with hyper-threading). This

comes to a total of 64 Cores, and 128 threads. The `multiprocessing` library was used to run the tournaments in parallel. To interpret the results, charts were generated at the end of every generation using `pyplotlib` . A consistent SSH connection was used to ensure the simulations were kept alive for the duration of the simulation on MIRA. Once training was complete, performance measurements was also conducted on MIRA, alongside other statistical evaluations to reduce computational time. The end champion was then transferred to the initial testing machine to be played against human input.

G Issues

Verifying that the moves were properly chosen was difficult and slow, as games needed to be played out first, with the quality assurance handled afterwards. The main issue with implementing the system were two-fold. The main issue was the sensitivity of the program, which was compounded by the time spent in terms of realising issues with the implementation. There were no obvious clues to analysing why the system may not perform as expected due to the scale of the system, and the issues were not realised until some time was spent during simulations.

Although the system was built where the games in the tournaments were "embarrassingly parallel", the machine used for training did not scale as efficiently as expected. Most of the computation was spent on calculating floating-point matrix operations (Which represent the neural network evaluations.) Later research has shown that the use of hyper-threading does not necessarily show to scale the performance, even though more threads exist. (Leng et al. 2002) This was due to the shared floating-point arithmetic registers in the CPU, where two threads utilise the registers of one physical core. The use of GPU acceleration was not considered due to the lack of availability, and the added complexity involved for its implementation.

IV RESULTS

For the experiment, three different neuroevolutionary systems were created, each running at different ply-depths. All systems run for 200 generations. For Figures containing three charts, each chart represents the results of the agents playing at 1-ply, 3-ply, and 6-ply, respectively. Each simulation utilised a population of 15 agents.

A Learning Rate

The learning rate was derived by comparing the ith generation's champions with its predecessor champions from the previous five generations $[i-1, \cdots, i-5]$. For each previous champion, six games were played, with the outcome of the current champion's performance scored and tallied up. An average of the outcomes was deduced, representing the learning rate. The learning rates of the first 2 champions were not measured due to insufficient comparisons. This measurement was performed for every generation until the end of training. These measurements do not influence the actual learning process of the system, but were used as one of the indicators of whether the system was learning in ability over time, i.e. whether the new champion was better than its predecessors. These results were plotted cumulatively to show the overall learning rate. A trend line was produced using a least squares linear polynomial fit.

A cumulative growth across all ply-depths can be seen such that returns are in the net positive as more generations are added. Furthermore, all ply-depths show a positive trend line. This

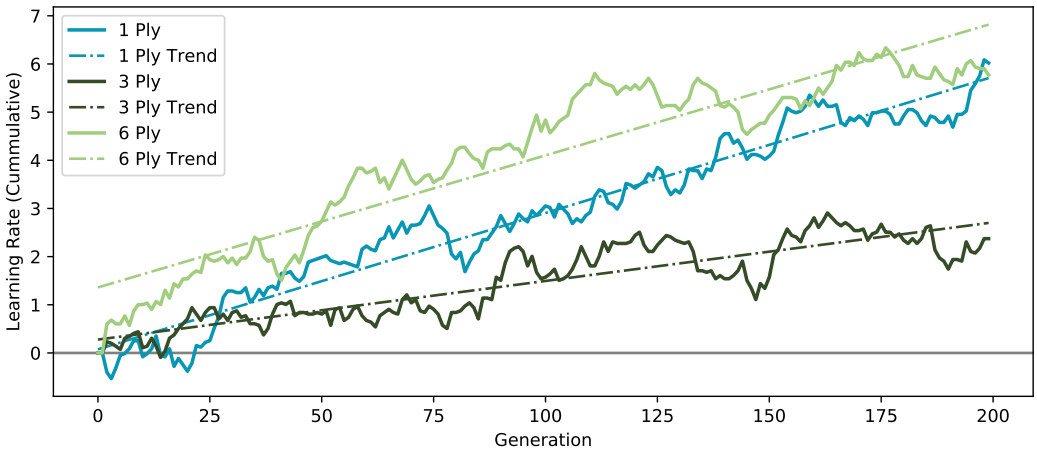


Figure 7: A chart showing the cumulative learning rate across the generations for the different ply-depths.

suggests that the system was more likely to show an increase in ability at every generation as opposed to a reduction. Interestingly, the 3-ply shows a slower learning rate than the other plys. However, even with safe mutations, it is shown that the performance over time was volatile for all ply-depths. This could potentially be prevented by creating a new set of agents and restarting the generation's tournament when a loss was seen. Further measurements were taken to understand the reasoning of the potential troughs in performance.

B Performance

To measure the overall system performance, the champion of the final generation was compared against a subset of their much earlier predecessors to measure the retention of move heuristics over time. For each predecessor, the trained system played 128 games (half as black and the other for white), and the number of wins, losses and draws were counted. For debugging purposes, all agents played against a randomly playing program to determine whether the system correctly evaluated moves. As there was a limited selection of ancestral agents tested against the systems, a trend line was produced (with a least squares cubic polynomial fit) for the non-random games.

For all systems, it appears that the winning rate increased over time, with the draw rate and loss decreased over time. For plys 3 and 6, it performed significantly better than its generation 0 counterpart (when considering the percentage of losses). This was not the case for the 1-ply - where it lost more frequently than its wins and draws combined - against its generation 0 counterpart. This would be caused by the lack of oversight heuristics retained as the system does not look ahead further enough to decide what heuristics would be kept by the neural network model.

C Champion Distribution

A novel approach was devised to determine the influence of genetic algorithms. Measurements were taken to show the genome type distribution of the champions across the generations,

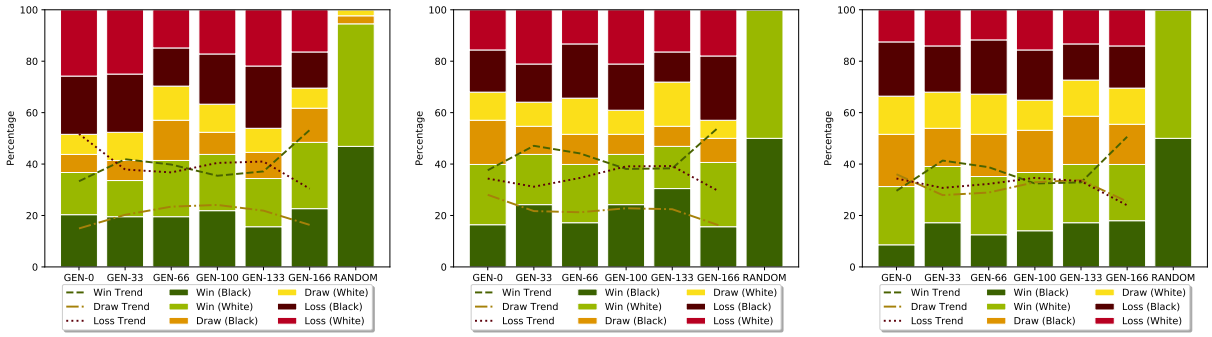


Figure 8: Charts showing the performance of the trained systems against their ancestral counterparts.

to determine whether the champions were more likely to be chosen based on the influence of mutations and crossovers. Each of the champions were looked at, and their genomic properties were collated to show a distribution of the chances of a particular agent being the champion. Champions that were also champions from prior generations were classed as persistent. Well performing agents from previous generations (but not champions) were classed as elite. Typically, elite agents would encompass the persistent agents but for the sake of measurements they were taken separately. This was also the case for mutations; although every offspring was mutated, some offspring were created using genomic crossovers (which were also classed separately), described in section C.4.

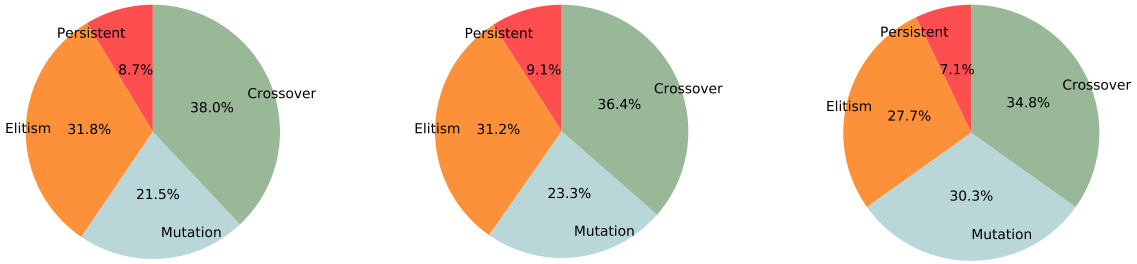


Figure 9: Distributions of the different genomic identities across the tournament champions.

Interestingly, the influence of crossovers reduced when the ply increased, and the influence of mutations increased across the generations. Overall however, crossover holds the highest probability across the plys which would suggest that the use of crossover methods (the distinctive feature of genetic algorithms opposed to other evolutionary methods) combined with mutation, would increase the chances of producing champions. The influence of the learning components of genetic algorithms (crossover and mutation) increased in relation to the increasing ply. (representing 59.5%, 59.7% and 65.1% of the distributions, respectively.)

D Quality of Inheritance Methods

Building on from the chart provided by Figure 9, the scores retrieved by the different genomic identities were compared against their earlier counterparts. This determined whether the use

of genetic algorithms can overall improve the quality of neural networks. To measure this, a generation's champion was correlated with their score (the same measurement used to evaluate the learning rate in Figure 7) produced when measured against their predecessors. The results are shown in Figure 10. Red circles indicate individual values for their classes.

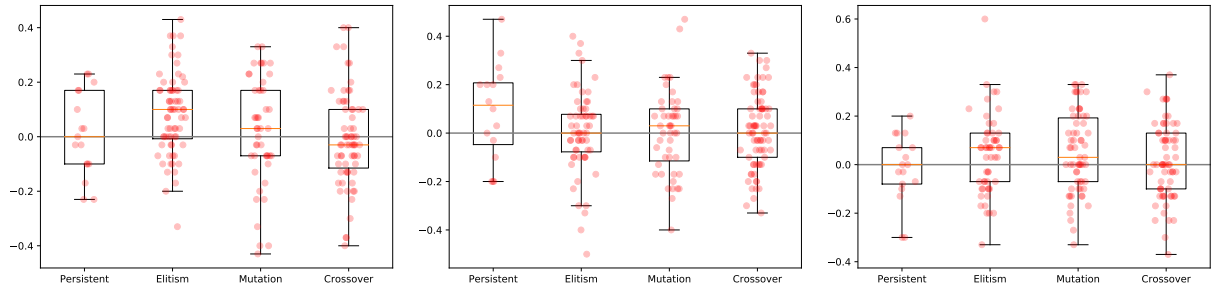


Figure 10: Box-Plots of the different genomic identities and their distribution of scores. The yellow line in the box represents the mean score.

Unsurprisingly, crossovers had a larger range of possible values compared to its mutation counterparts. This was most likely due to the weights being placed at different positions on the neural network, disrupting any heuristics it would have otherwise devised. However, the mean crossover score suggested that it negatively impacted the learning rate. Furthermore, the range and mean scores for the elite genomes (which in this specific description includes persistent agents) suggests that they were more likely to increase the learning rate.

E Other Observations

E.1 Number of Moves

The number of moves was measured to verify that the system was able to produce quality moves across the training phase. Figure 11 shows the distribution of moves for the games played over the generations.

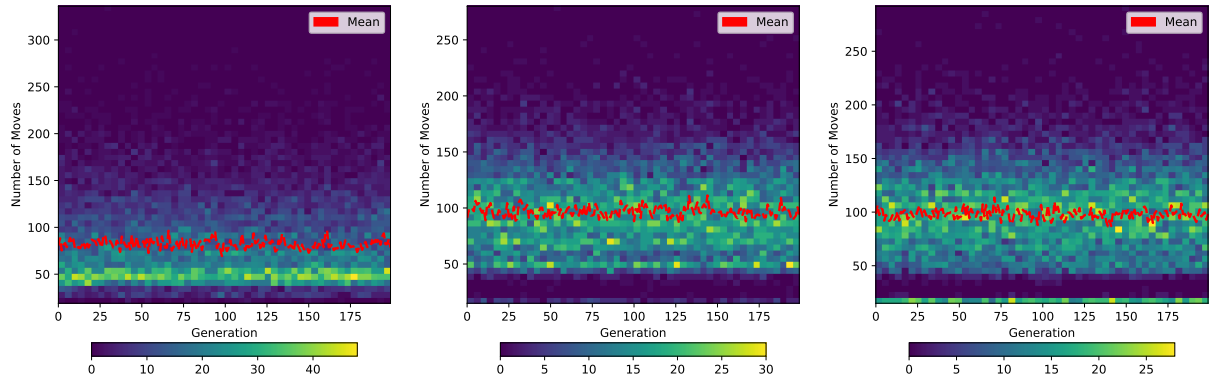


Figure 11: 2D Histograms representing the number of moves played in the games played in the generations.

From the charts, the median number of moves in a game for 1-ply was around 50. For the higher plys, the median was not as tightly grouped together. When combined with the mean

number of moves, suggested that when both players chose stronger quality moves, their games lasted longer. The mean number of games for both 3 and 6 plys were similar. However, the similarity of the two charts was most likely caused by the hard limit imposed by the game rules (see *C.2 Tournament Selection*), where draws were forced when a piece was not taken after 100 moves.

The number of moves for the higher ply depths was compared to other known level of abilities. Data scraped from the US National 1952 ACF Tournament (Loy 1952) showed that the average number of moves in the tournament games was 54.74. 10000 randomly played Draughts simulations showed an average of 69.69 moves. With players in plys 3 and 6 playing to somewhat equal ability (relative to their population), the increased number of moves suggested that it was more difficult to see a disparity between the agents in their respective populations. This consequently confirmed that the tree-search algorithm was implemented correctly, reducing the unknowns considered during evaluation.

E.2 CPU Times

	1-Ply	3-Ply	6-Ply
Generation Mean	0:04:24	0:17:32	0:38:45
Net (All Generations)	7:21:25	1 day, 5:14:41	5 days, 9:12:25

Figure 12: Mean and Net running times of system training for the various depths.

To measure the time complexity of the program, time measurements were taken for every game played, and the time length for a generation to finish the tournament. As expected, the system was shown to scale in a relatively linear fashion according to Figure 12. This was most likely caused by growing the search limit for MCTS in a linear fashion based on the ply.

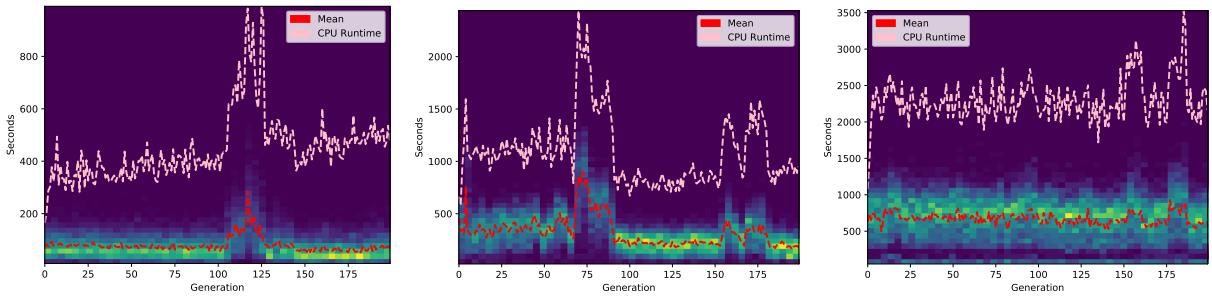


Figure 13: 2D Histograms of the average times spent on the games during the simulations.

Figure 13 suggests that the CPU time was unstable during training, but further analysis was shown that it did not correlate with the performance of the system. This was most likely related to the use of a time-shared machine (Durham's MIRA) to run the simulations, which may have throttled the resources for the system for enable other users to run their own programs.

V EVALUATION

The original research question is now reflected upon: "is it possible to create a performing draughts playing agent by the use of Genetic Algorithms and Neural Networks?". The question is answered through the examination of the strengths and limitations of the system.

A Strengths

Figures 7, 8 and 9 suggest that the system has improved in ability over time. This is pivotal to the research question, as no human intervention (i.e data suggested by humans) was used to help make the system learn. It also reinforces the idea that derivative based learning algorithms are not necessarily the only ways to train a neural network. This is possible for the case of playing Checkers, as an infinite number of games can be simulated. The use of crossovers suggested that it could potentially improve the quality of the system.

This suggests that neuroevolution is a viable method for problems with vast search spaces, or problems where data is hard to access. Although this is the case for Deep learning and evolutionary computation in general, the benefit of using genetic algorithms is that the system relies on a relatively simple learning heuristic that is open to domain specific adaptation. Gradient based learning requires the function translatable to some form of a differential sum of partial derivatives, which may not be feasible in some circumstances.

B Limitations

Figure 12 and 13 indicated a far slower learning rate than traditional supervised NN of equivalent quality. For instance, research has shown to create a notably strong Chess AI within the span of 72 hours by training a neural network against pre-classified tactics problems and predetermined grandmaster tournament games. (Lai 2015). A comparison between the two is fundamentally difficult, however, as the system was not tested against an AI that is universally recognised to be strong.

The crossover mechanism is a hit and miss, and is shown in figure 10, on average, to show a negligible improvement in the learning rate. This was also seen in another research into the potentially negative influence of the textbook crossover mechanism (Emmanouilidis & Hunter 2000), where they also found it to negatively impact learning overall. The results have shown however, crossovers show a wide range of scores, including largely positive indicators (Also shown in Figure 10). If provided a well devised crossover mechanism, can dramatically improve the quality of the learning over time.

The quality of the neuroevolutionary algorithm boils down to the quality of the tournament method. Round-Robin is not necessarily the most efficient (from a number of games played perspective, ignoring parallelism) and does not eliminate cases where there is an ambiguous decision between the champions (which could be due to several players having an equal number of points). This could also relate to the fact that the retention of heuristics necessary to win against much older predecessors were lost. This may be a fault of the tournament mechanism, where the population were made to optimise against each other, as opposed to having a common premise such as optimising the previous champion. Creating a high-quality tournament is difficult and domain specific, or rather, there is no universal fitness function for all problems.

C Approach

The mechanism for devising the learning rate was more efficient than comparing the present champion against a holistic comparison of the rest of the system. This measurement of learning rate was somewhat arbitrary, as the learning rate is deduced against the population and not necessarily the system as a whole, and may best be compared against systems that are classified to play at a particular level. This would have helped in determining how strong the system was against popular AI checkers programs.

As most of the experiments are constrained by computational power and time, it may be best to re-evaluate this concept until either computational resources are more available, such as the use of GPU acceleration. The system should also be left to run for longer generations as presently the results do not show a notably strong correlation. The lack of computational resources also influenced the number of experimental testing that can be measured, such as the population size, the number of generations, the dimensions of the neural network, and the learning rate.

Another consideration for the project is to experiment with different crossover algorithms that utilise the temporal difference between inputs and outputs. This could assist in the creation of safe crossovers which could also potentially accelerate the learning rate, by retaining the best aspects of two agents.

There is an incredibly vast scope of future work that would extend the project. One interesting avenue is to consider using a safer method to produce crossover mechanism, exercising other contemporary methods in machine learning. It may also be the case that more time could be spent exploring problem solving mechanisms outside of computer science. In this particular example, the algorithms used are inspired by various biological and neuroscientific understandings. More research into this field may assist in exploring creative methods of solving issues concerning the genetic crossover mechanisms and mutation rates.

VI CONCLUSION

The project was successful in terms of fulfilling its objectives and Aims. A system was implemented using evolutionary methods, and it successfully plays checkers in a manner in which it learnt from over time. To the extent of its growth remains suspect however. The main findings of this project is as follows:

1. It is evidently possible to create a checkers playing agent that learns to play itself, using genetic algorithms and neural networks.

2. Neuroevolution is inefficient compared to their gradient based counterparts. This however was understandable due to the heavy dependency of entropy disguised as learning.

3. The tournament can be anything, but it is important to derive a high quality tournament mechanism.

4. How the crossover was implemented can be a deciding factor in how the agents learn to play over time. The crossover method used has shown a negligible impact on average, but has produced a wide range of possible results, some of which could greatly increase the quality of the learning rate.

The major implication is that it is a possible contender for tasks that require unsupervised learning, and that data needed to produce a strong classifier is hard to access. Provided a high quality tournament method can be designed, neuroevolution could be adapted to handle any simulation based problem, where the objective can be adapted to fit into some optimisable classifier.

References

Al-Khateeb, B. & Kendall, G. (2009), 'Introducing a round robin tournament into blondie24', *CIG2009 - 2009 IEEE Symposium on Computational Intelligence and Games* **44**(0), 112–116.

Al-Khateeb, B. & Kendall, G. (2010), The importance of a piece difference feature to Blondie24, *in* '2010 UK Workshop on Computational Intelligence (UKCI)', pp. 1–6.

Al-Khateeb, B. & Kendall, G. (2012), 'Effect of look-ahead depth in evolutionary checkers', *Journal of Computer Science and Technology* **27**(5), 996–1006.

Allis, V. (1994), Searching for solutions in games and artificial intelligence, PhD thesis, University of Limburg, S.l. OCLC: 905509528.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. & Colton, S. (2012), 'A Survey of Monte Carlo Tree Search Methods', *IEEE Transactions on Computational Intelligence and AI in Games* **4**(1), 1–43.

Chellapilla, K. & Fogel, D. (1999), 'Evolving Neural Networks to Play Checkers without Expert Knowledge', *IEEE Transactions on Neural Networks* **10**(6), 1382–1391.

Emmanouilidis, C. & Hunter, A. (2000), A Comparison of Crossover Operators in Neural Network Feature Selection with Multiobjective Evolutionary Algorithms, *in* 'GECCO-2000 Workshop on Evolutionary Computation in the Development of Artificial Neural Networks, Las Vegas'.

Lai, M. (2015), 'Giraffe: Using Deep Reinforcement Learning to Play Chess', *arXiv:1509.01549 [cs]* . arXiv: 1509.01549.

Lehman, J., Chen, J., Clune, J. & Stanley, K. (2017), 'Safe Mutations for Deep and Recurrent Neural Networks through Output Gradients', *arXiv:1712.06563 [cs]* . arXiv: 1712.06563.

Leng, T., Ali, R., Hsieh, J., Mashayekhi, V. & Rooholamini, R. (2002), 'An empirical study of hyper-threading in high performance computing clusters', *Linux HPC Revolution* **45**.

Lorentz, R. (2016), 'Using evaluation functions in Monte-Carlo Tree Search', *Theoretical Computer Science* **644**, 106–113.

Loy, J. (1952), 'US National 1952 Games', *The American Checker Federation* .
URL: *http://www.usacheckers.com/show_game_start.php?event=US%20National%201952*

Perez, S. (n.d.), 'Apply genetic algorithm to the learning phase of a neural network'.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), 'Learning representations by back-propagating errors', *Nature* **323**(6088), 533–536.

Samuel, A. (1959), 'Some studies in machine learning using the game of checkers', *IBM Journal of Research and Development* **3**, 210–229.

Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P. & Sutphen, S. (2007), 'Checkers Is Solved', *Science* **317**(5844), 1518–1522.

Schaeffer, J. & Lake, R. (1996), 'Solving the Game of Checkers', *Games of No Chance* **29**, 119–133.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. & Clune, J. (2017), 'Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning', *arXiv:1712.06567 [cs]* . arXiv: 1712.06567.

Xie, L. & Yuille, A. (2017), 'Genetic CNN', *arXiv:1703.01513 [cs]* . arXiv: 1703.01513.