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

Presently, competitive Draughts AI players are currently designed to play at a fixed ability. While it has produced very competitive and intelligent players, they require some human intervention to improve their performance. This is mainly due to their dependency on pre-defined move databases. Optimal moves are pre-calculated, and recalled when necessary. Through neuroevolution, this issue could possibly be solved by creating a player that can grow in ability over time, by learning to play itself.

Aims

The objective of this project is to explore how well neuroevolutionary approach can tackle the game of English Draughts. First, we study previous historical successes in the field, and look at the components that helped build their systems. Then, we look at how they can be adapted to suit the objective at hand.

Method

The method starts with designing a feed-forward neural network that takes as input, the state of a checker board, and outputs an evaluation of the players advantage. This is then used in a algorithm that evaluates future moves to predict the best move at a given position. This, alongside a set of weights for the neural network, creates a player that can evaluate potential moves. Finally, the player is then used on an existing Draughts framework that will provide the player with the ability to play Draughts. Genetic algorithms are used to adjust the weights of the neural network, with the intention of making the system learn to evaluate checker boards more precisely.

Results

Overall, the neuroevolutionary approach ahs shown to learn and improve over time. The net learning rate was positive. However, the wide scope of the adjustments afforded may have impacted the learning rate and has consequently shown to be volatile at some situations. The use of crossovers may dramatically influence the quality of the learning.

Conclusion

Under the premise that the fitness function is well defined, neuroevolution can be considered as an option to create a draughts playing AI. However, the robustness of the system is quite volatile. To counteract this, the system is best paired with mutation and crossover methods that are not heavily reliant on entropy.

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

I INTRODUCTION

The intention of this project is to explore the effectiveness of genetic algorithms to improve the neurons of a neural network. Neural networks can be used to evaluate the performance of two players in a zero-sum game of Draughts. We attempt to manipulate the neurons through the use

of genetic algorithms to increase the accuracy of the evaluation. This would potentially allow us to create an effective Draughts playing agent that, when provided with the option to consider a set of moves, would have the ability to play and learn without human input.

A Draughts

English Draughts (also 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, and they 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 in 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. Pieces can be captured consecutively in a single turn if the moves afford the scenario. A player wins by capturing all of their opponents 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 are non-linear statistical data-modelling tools, linking 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 characteristics, and appear in layers. The first layer is defined as the input layer, and the last layer being 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 it can also find patterns in the data.

C Genetic Algorithms

As 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 Charles Darwin's evolutionism theory; individuals are created based on the improvement of their ancestors genetic information (genome). Genomes are analogical to a data structure, most commonly an 1D array. Evolutionary algorithms typically consist of a population of agents, a tournament selection process, and the introduction of probabilistic mutation on genomes. This in turn allows evolutionary algorithms to create genomes that increase in quality over time in relation to the tournament. Genetic algorithms distinct themselves from evolutionary algorithms through its use of algorithmic crossover mechanisms, where two parents 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, my intention is to explore the effectiveness of the combined methods in terms of developing a draughts playing artificial intelligence.

D.1 Minimum Objective

The minimum objective is to implement a feed-forward neural network, alongside a Draughts game interface for the network to evaluate on. This will need to pair with a decision making algorithm that determines the move to choose at a given state of the game. Finally, a simple genetic algorithm would also need to be implemented that interacts with the neural network, allowing the possibility to learn over time.

D.2 Intermediate Objective

The intermediate objective involves the implementation of an interface to play against the trained result, and to create a monte-carlo tree search algorithm. This would allow the system to think in ahead, allowing it to evaluate better quality decisions. Also, as the genetic algorithm includes a tournament mechanism which will play many games simultaneously, this will need to be implemented to run in parallel.

D.3 Advanced Objective

The advanced objective consists of producing a system that indicates that it is learning over time. The later sections of this paper evaluates the quality of the learning rate and other results that determine the extent of its success.

II RELATED WORK

Although Draughts is a perfect information game, it has been historically used as a testing ground for artificial intelligence and computational capabilities since the early introduction of computers. This is 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, but the computational performance was not available to prove the method at the time of discovery (Schaeffer & Lake 1996). The same researchers however reinvestigated the research a decade later, eventually proving the game to be *weakly-solved.* (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) However, the premise of this paper is not necessarily to solve the game itself, but to rather see whether neuroevolution is a method that, when measured via its ability to play the game, shows that it can evolve in performance over time.

Research in artificial intelligence alone has been active since, targeting games with a higher move space, notably Chess and, more recently, Go. This section will summarise specific and notable techniques relevant to the discussion of the task at hand.

A Arthur Samuel

In 1959, An early design of machine learning to play Draughts was devised by Arthur Samuel(Samuel 1959). His algorithm was based on the continual improvement of a linear combination of heuristics, which act as an evaluator function for a given state of the checker board. The heuristics represented different strategies one would have considered when playing Draughts. Some of these heuristics include the number of pieces on side (such that one side is shown to have a particular advantage given that they have more pieces), and the number of pieces positioned along the central diagonal. His evolutionary strategy consisted of agents, each of which represented coefficient values for the heuristics (initially random), and new agents are formed via the manipulation of its successor's coefficients with subtle mutations.

Samuel's decision making algorithm primarily revolved around an early concept of the mini-max algorithm. This algorithm takes as input the present state of the checker board, and returns the best move to choose. Mini-max algorithm is described using the following instructions:

1. Each node represents a state of a checker board. Child nodes represent a future state of the checker board, given a move. Child nodes can only be produced from the parent node; i.e. a move from a parent node will produce a child node.

2. Child nodes are expanded until some depth is reached. From here, an evaluation function is used to calculate the value of the leaf nodes at this depth. This evaluation function produces a value that indicates a players advantage in a given state.

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

4. The best value propagates upwards until it reaches to a child of the root node. This represents an agent's current state of the game.

The use of alpha-beta pruning helped to ignore evaluating disadvantageous states, reducing the overall number of neural network evaluations. Once the possible moves have been considered, the best child node is chosen, representing the best move at a given position.

There are inherent trade-offs to this decision algorithm, especially with utilising higher ply counts; asymptotic complexity of mini-max is exponential; it being $O(x^y)$ for x being the branching factor, and y being depth. The branching factor consists of the moves from each agent in a given game. This exponential growth is the achilles heel to the mini-max approach.

A later investigation by a checkers magazine evaluated Samuel's player to perform at below Class B (Schaeffer 1997, Fogel 2000); (Class B being categorised with an ELO range of [1600-1799]). This implied that it played at a performance that can be seen at club level players. However, the performance of the system is handicapped by the quality of the heuristics Samuel 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

The idea of evolving neural networks to evaluate board-states was based on the success Chellapilla and Fogel had in evolving their own Draughts playing neural networks. Their method

used Samuel's system as a foundation and modified the evaluation method, using feed-forward neural networks instead of a distinct linear combination. Their resulting player, Blondie24, was then taught to play without priori knowledge in a similar vein to Samuel's system. (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 (which are initially random), with the second property being a self-adaptive parameter vector, τ_i, which determined how random the mutations are when the agent i evolved.

$$\tau_i(j) = \tau_i(j) \cdot e^{\mu} \tag{1}$$

Figure 1: The mutation formula in Blondie24, where $j = 1, ..., N_w$ represents individual weight of the neural network w, and μ representing a random decimal in the range of $[0 - 1]$.

In each generation, every agent plays each other in a round robin style tournament. Agents plays an equal number of games, and successors are chosen based on their performance in the tournament. The best agents are chosen to mutate themselves, producing new agents in the chance that the new agents perform better than prior. Mutation is performed using the formula in Figure 1. Like Samuel's program, Blondie24 also used Mini-Max with alpha-beta pruning.

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

We do not wish to repeat this work, but it is important to understand that we wish to abstract and modularise parts of the best features of this system to modify and extend them to the solution described in the paper.

C Post Blondie24

Blondie24 was influential in the field of neuroevolution during its time, leading to various studies on the theory behind its success. A notable example is the series of publications by Al-Khateeb and Kendall, where they explore in detail the different features that comprise Blondie24. One of their studies concerning the effects of the the ply depth, concluded that having a higher ply-depth unsurprisingly increases the overall performance of the player, and that agents trained at a higher ply performs significantly worse when playing at a lower ply. More importantly, the results in the paper suggest that a 4ply depth produces the best value function to start with during the learning phase, and increasing the ply depth for evaluations. (Al-Khateeb & Kendall 2012) Their study in the influence of tournaments in Blondie24 concluded that the league structure seemed more appropriate to use as opposed to the round robin tournament. (Al-Khateeb & Kendall 2009) This is however more difficult to set up in parallel, which may be a largely contributing factor during the implementation of the system.

D Return of Genetic Algorithms

Since then, there has been a non-trivial number of published papers describing the promising utility of genetic algorithms and neural networks in general. Genetic algorithms have been shown to evolve weights for a convolutional neural network in a classification task better than back-propagation, in nearly all situations, other than when the number of generations generated for neural networks are small. (Perez n.d.) Another research showed its ability to compute optimal structures for deep neural networks, using an image recognition dataset as the benchmark for competitiveness against the networks. (Xie & Yuille 2017)

However, one recent paper by Clune et al. on behalf of Uber Labs returned to the use of genetic algorithms as a basis 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 revolves around measuring the sensitivity of the network to changes with some of the weights. (Lehman et al. 2017) This afforded the system to evolve deep neural networks (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 exploits the knowledge of the neural network structure, by retaining sampled experiences of the network (inputs that feed into the network and its outputs). This allows the weights to be manipulated by comparing its significance with the output, by comparing it with the sampled experiences.

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

Unfortunately, it is 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 cores) is of a magnitude greater than what is available for the task at hand, but a more simplistic and adapted model can be produced using similar ideas.

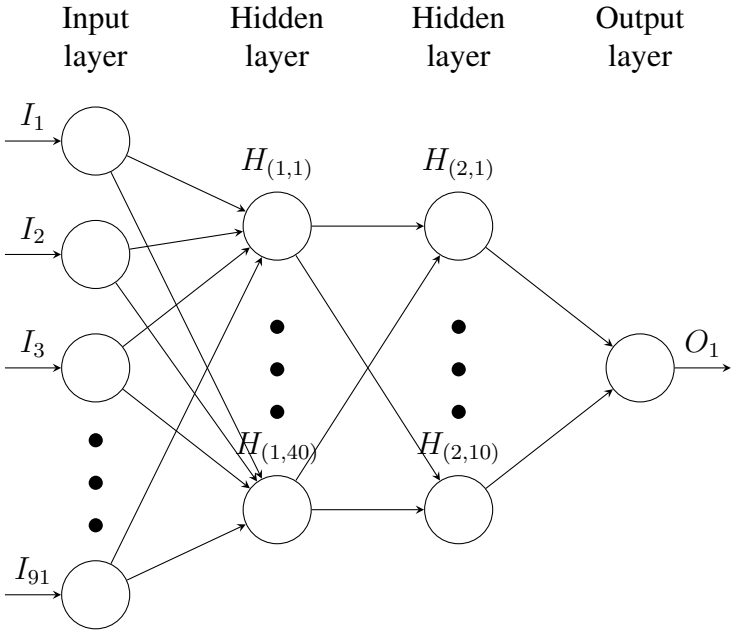
A popular trend between the contemporary research is the shared absence of the use of crossover algorithms. One paper describes the use of different crossover algorithms on neural networks having no clear advantage over each other, in comparison to a uniform crossover (uniform crossovers are also commonly defined as the textbook crossover) which disrupted the overall performance of the neural networks. 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 is important to consider this when devising a crossover method, such that the effect it has on the neural network 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 our objectives. However, the sections below are considered under the premise that the interface is built prior.

A Evaluation Function

Figure 2: The chosen neural network model. The preprocessed values of the checker board are used as input. An output is produced after propagation that ranges from [-1,1].



A feed-forward multilayer perceptron 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 2 shows a visual representation of the network structure.

The inputs of the network is a preprocessed 1D array of the checker board. Preprocessing the input starts 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. The subsquares kernel size range from a 3x3 to 8x8. Figure 3 visualises the retrieval of the subsquares. With the kernels, the number of pieces in each subsquare are summed up. A new array is formed, comprising of the sums of the different kernels that comprised the board. The resulting array (comprised of 91 entries) represents the preprocessed input, which is 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 it is disputed about its precise value advantage. This could be decided by the system such that could can evolve to understand that a King is eventually worth more than the pawn. For the sake of simplicity however, 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 plateau 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

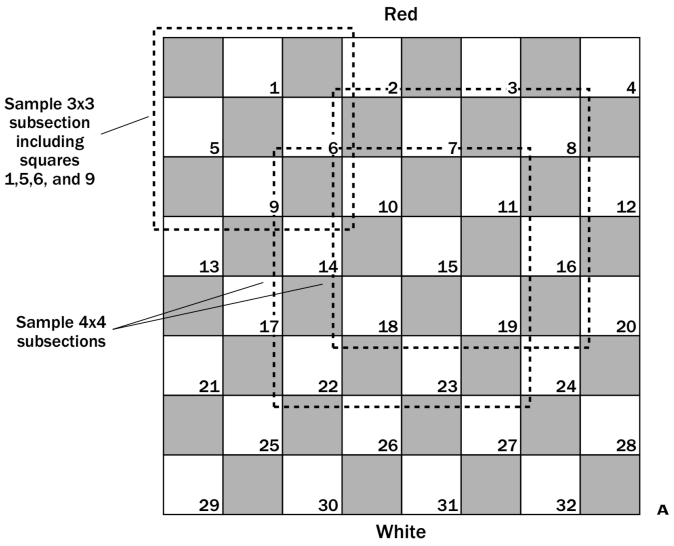


Figure 3: A diagram visualising the processing of subsquares.

the form of other shapes. Common characteristics of activation functions include values to be monotonically increasing, continuous, differentiable and bounded.

Our choice for the activation function is tanh (a type of sigmoidal function), shown in figure 4. The use of tanh facilitates the ability to simulate a zero sum formula (for a range of [-1 to 1]) due to its symmetric properties around the origin.

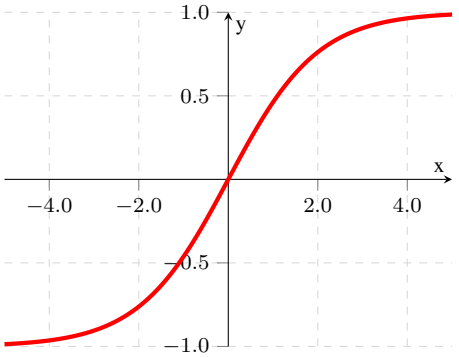


Figure 4: Graph of tanh function $f(x) = \frac{2}{1+e^{-x}} - 1$ (An example of an activation function with a sigmoid curve).

Results are cached such that evaluations are saved such that they can be recalled immediately, in the event that the same input is evaluated. This has shown varying levels of time improvement dependent on the ply depth. As the ply increases, the size of the state space increases, and therefore the chances of recalling the same set of inputs decrease. The cache is also used for our safe mutations algorithm, described later on in Section C.3.

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

B Decision Making Algorithm

Initially, the decision making 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 randomly 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 is played in rounds which consist of four operations:

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

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

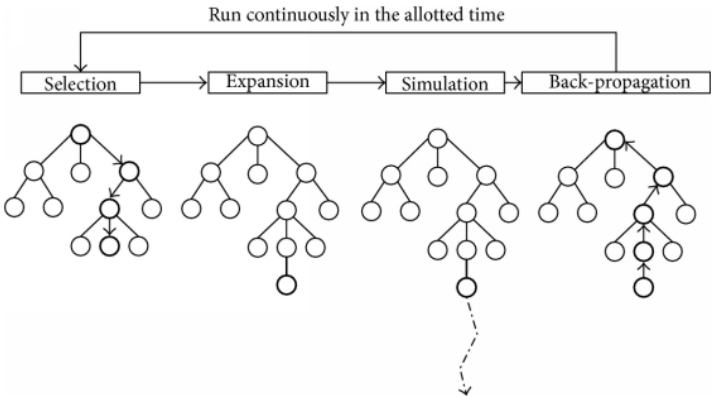


Figure 5: A diagram describing the MCTS process.

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

C Genetic Algorithms

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

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

Figure 6: 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.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 is consistent of 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 would continue 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 being directly mutated from the parents themselves. Another four children were 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.

C.2 Tournament Selection

To sort the quality of the players in a population, a tournament selection process was deduced. This allows us to choose the best players who will continue to play in the next generation.

Each agent in the population played 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 is not deduced at this stage, a draw is called. Draws are also called when there is a three-fold move repetition from both players. A win is worth 2 points, a draw being none and a loss being -1 point. Both the agent and its opponent receives 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 of 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 lead to a longer overall running time even though less 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 are made through the use of mutations. This is used as one of the learning mechanisms that help change the decision

factors of the neural network. Weight and biases of an agent's neural network would increment by a random value that is created using the equation in Figure 6.

Like 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. Soft mutation (Lehman et al. 2017) is also used. We use a subset of historical neural network calculations as a premise to guide our 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 6. 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 the best y.

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

C.4 Crossover Strategy

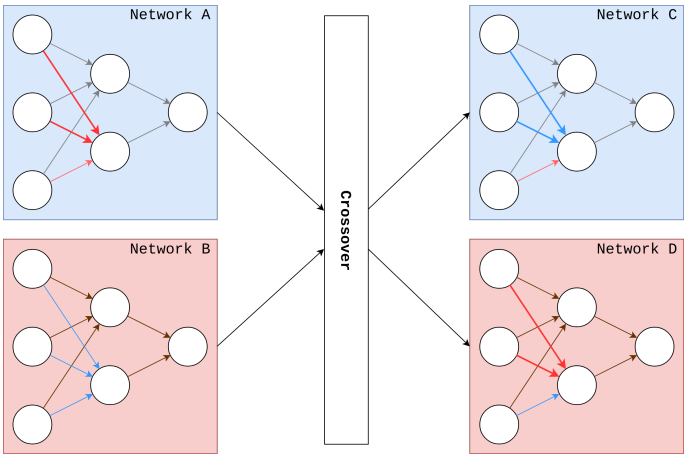


Figure 7: A diagram visualising the crossover process.

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

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 is chosen. A dominant parent

is the agent who gained more points in the tournament round than the other (who would be the submissive parent). If a dominant parent cannot be deduced it would be randomly chosen.

Once a random node was chosen, a subset of weights and biases that are fed into the chosen node are swapped with the submissive parents 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 7 visualises the process. In the diagram, Network A and B are parents, creating reciprocal offspring networks C and D. A is the dominant network, and A's node with the red inputs is chosen. The weights and biases of the red inputs are swapped with the values at the same position in network B. Network C is formed with Network B's dominant weights, with the rest being composed from A. Network D is 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

At the end of a given generation, we measure growth of performance by using the generation's champion. When a new champion is generated, it is played against the previous 5 champions from earlier generations. 6 games are played for each previous champion, with 3 being as Black, and 3 being White. A mean score is calculated from those 6 games. The overall performance of the current champion is the mean of the 5 sets of games. A positive improvement is when the mean of means are greater than 0. Point Score for the champion games are measured by 1,0,-1 where a Win counts as 1 point, 0 for a draw, and -1 for a loss. The weights are scaled differently to the regular tournament in order to portray an even distribution of the learning rate.

Evaluation Method

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

E Tools

The system was implemented using Python 3.6 and NumPy. Initial runs operated on a 1 and 2-ply load to determine the stability of the system. This was run on a linux (Ubuntu) machine containing a 4-core Intel i5 6600u processor with 12GB of memory. Development and debugging was also handled on this machine. Once testing has proven to be stable, the system ran on Durham's MIRA distributed system (Debian) with a 1,3 and 6-ply move depth. This machine utilises 4 Intel Xeon E7-8860 processors (Each CPU contains 16 physical cores running in 2.2Ghz, with hyperthreading). This comes to a total of 64 Cores, and 128 threads. MIRA also comes with 256GB of DDR3 memory.

To keep simulations running on MIRA, a consistent SSH connection was used to ensure the simulations were kept alive for the duration of the simulation. Once training is done, testing was

also conducted on MIRA. The end champion is then transferred to the initial testing machine in order to be played against by human input. Statistical evaluations and calculations are also calculated on MIRA to reduce computational time.

F Verification and Validation

- Written with test cases - Individual testing of class procedures

G Setup and Issues

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

Interestingly, the machine used for training did not perform necessarily as efficiently as expected. Although the system is built such that the games in the tournaments are "embarrassingly parallel", the machine does not scale accordingly. Most of the computation is spent on calculating floating point matrix operations (Which represents the neural network evaluations.) Later research has shown that the use of hyperthreading does not necessarily show to scale the performance, even though more threads exist. (Leng et al. 2002) This is 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 is not considered due to the lack of access.

IV RESULTS

For the experiment, we create three different neuroevolutionary agents with separate ply-depth: 1,3 and 6. Ply Depths 1 and 3 are run for 100 generations, and 6 is run for 200 generations to measure the overall potential of the system. For sections containing three charts, each chart represents the results of the agents playing at 1-ply, 3-ply, and 6-ply, respectively, in that order. Each simulation utilises a population of 15 agents.

A Learning Rate

The learning rate is derived by comparing a generation's champions with its predecessor champions from the previous five generations. For each older champion, six games are played, with the outcome of the current champion's performance scored and tallied up. An average of the six outcomes are calculated, representing the learning rate. We do not take the learning rate of the first 2 champions as this would not produce a sufficient comparison. This measurement is performed for every generation until the end of training. It is also the case that this does not influence the actual learning process of the system, but is used as one of the indicators of whether the system is learning in ability over time.

A cumulative growth across all ply depths can be seen such that returns are in the net positive as more generations are added. However, even with safe mutations, we can also see that the learning is very volatile. As the champion is chosen from a tournament, it may be related to the

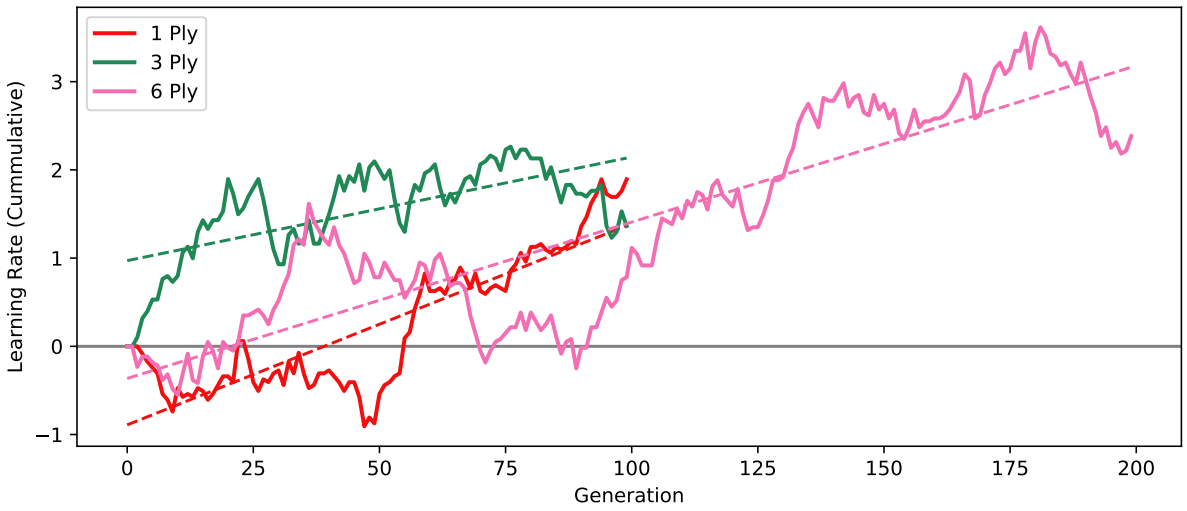


Figure 8: A chart showing the cumulative learning rate across the generations for the different ply depths.

quality of the following generations having a lower ability to evaluate the board than it's parent generation. This could potentially be prevented by reusing the previous champion and restarting a particular generation if a loss can be seen. This could however funnel the system towards a local maximum in terms of learning. Further measurements are taken to understand the reasoning of the potential troughs in learning.

B Performance

To measure the system performance at the end of training, the champion of the final generation is compared against a subset of their predecessors. For each predecessor, the trained system plays 128 games (half as black and the other for white), and the number of wins, losses and draws are counted to produce the following statistics. For debugging purposes, all agents are also played against a random program to determine whether the system is evaluating moves properly. Fortunately, all agents successfully won every game.

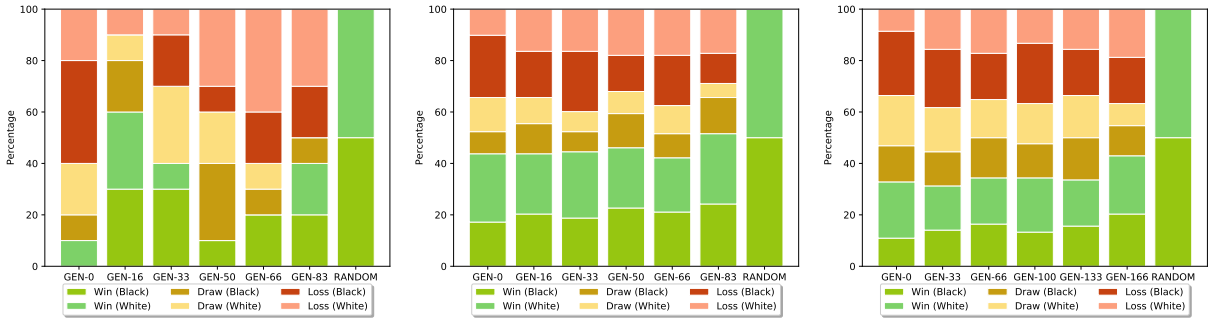


Figure 9: Charts showing the performance of the trained systems against their ancestral counterparts.

For the ply depth of 1, we can see an immediate net trend, with the end system beating earlier

generations except for the first generation. This may make sense, as heuristic retention would be lost as the search depth is so low. This does however suggest that the system is worse off than playing with an initialised system (without training). For the higher ply depths (3,6) however, the agents manage to consistently maintain a higher chance of winning/drawing against their earlier counterparts. It is difficult to see a trend line. For the 6-ply, it is evident that there is an increase of draws caused against the earlier agents, which could be seen as an advantage.

C Champion Distribution

The following charts shows the following distribution charts the champions of the generations. The intention behind this measurement is to determine whether champions are inherently chosen based on the influence of mutations, crossovers of simply elitism. Measurements are taken such that the champion of each generation is looked at, and their genomic properties are collated to show a distribution of the chances of a particular agent being the champion. Note that while every offspring is mutated, some of the offspring are created using genomic crossovers, described in section C.4.

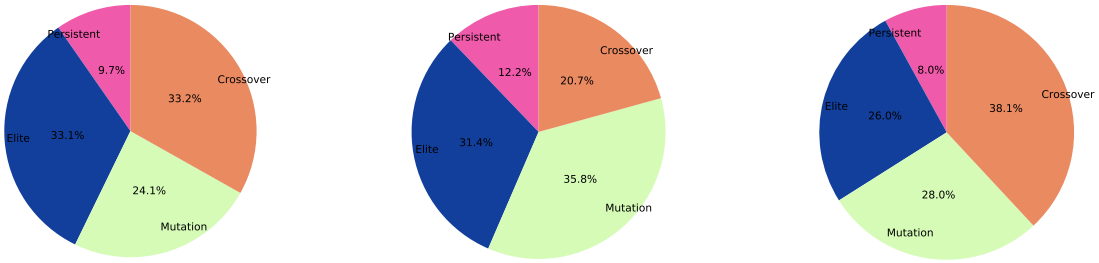


Figure 10: Distributions of the different genomic identities across the tournament champions.

The influence of offspring become more apparent for larger plys, with mutation and crossover appearing more frequently, coming at 66.1% of the distribution. Interestingly, for ply depths 1 and 6, crossovers tend to be the most frequent occurrence when it comes to producing champions at every generation. This suggests that crossover methods tend to produce better agents than their counterparts.

D Influence of Inheritance Methods

Building on from the chart provided by Figure 10, we look into the scores retrieved by the different genomic identities when they are compared against their earlier counterparts. This is to determine whether the use of genetic algorithms can overall improve the quality of neural networks. To measure this, we correlate the generation's champion with their score (the same measurement used to evaluate the learning rate in Figure 8) produced when measured against their predecessors. The results are shown in Figure 11.

Interestingly, for the 3-ply, the crossover method is most likely to produce a growing system, producing the highest mean. However, across all of the ply depth agents, we can see that the use of mutation produces the best possible agents. For the higher 6-ply depth we can see that it did not significantly influence the learning rate of the system.

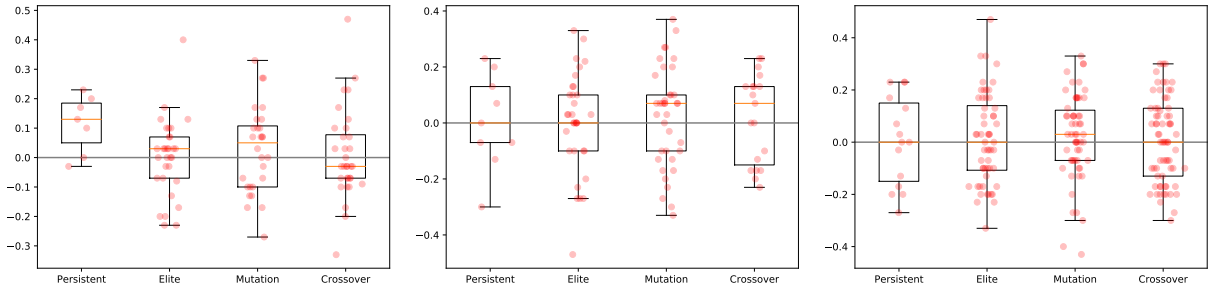


Figure 11: Box-Plots of the different genomic identities and their distribution of scores. The yellow line in the box represents the mean score.

E Other Observations

E.1 Number of Moves

The number of moves is measured to verify that the system is producing realistic and quality moves over the training phase. The following charts show the distribution of moves for the games played over the generations, alongside the mean. The results are shown in Figure 12:

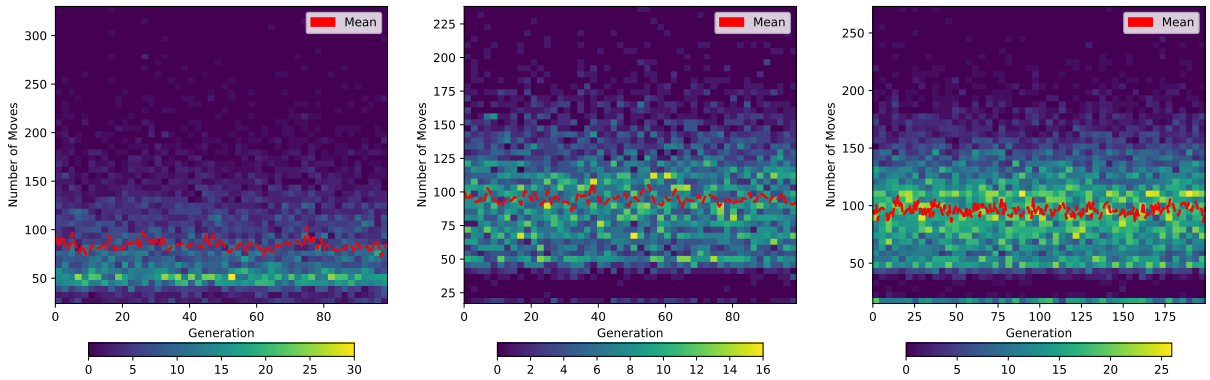


Figure 12: 2D Histograms representing the number of moves played in the games played in the generations.

For the 1-ply simulations, the average mean of moves is lower, but is counteracted by the much larger range of games that have a large number of moves. Understandably, the lower quartile for the 1-ply is lower than the others. This may be due to the level of precision, where it does not think further ahead enough to make better moves. This explanation applies inversely for the larger ply-depths, notably the 6-ply chart, where the level of precision is higher, leading to a tighter distribution of games having similar numbers of moves played.

As the system is shown to play fairly consistently across the generations, this verifies that the system is choosing consistent decisions across the board. This confirms that the decision making algorithm is implemented precisely, reducing the number of unknowns for evaluation.

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

Figure 13: Mean and Net running times of system training for the various depths.

E.2 CPU Times

To measure the efficiency of the program, time measurements are taken for every game played, and the time length for a generation to finish the tournament. As expected, the system is shown to grow in running time linearly according to Figure 13. This is caused by of growing the search limit for MCTS in a linear fashion based on the ply-depth.

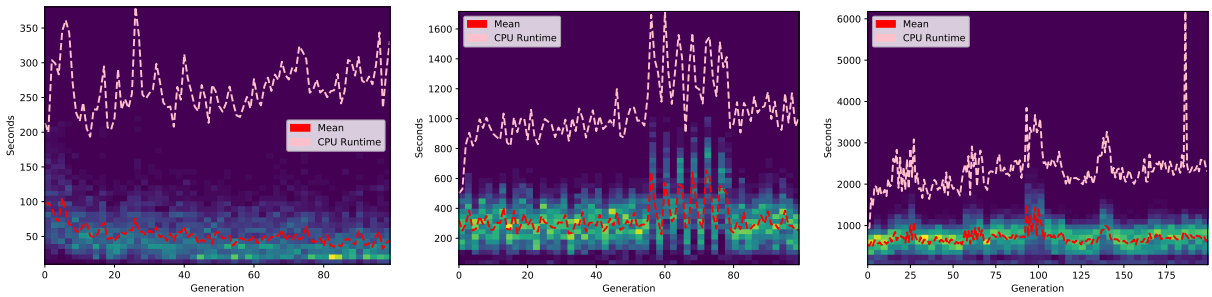


Figure 14: 2D Histograms of the average times spent on the games during the simulations.

Figure 14 suggests that the CPU time is very unstable, but it does not correlate with the performance of the system. This is most likely related to the use of a time-shared machine (Durham's MIRA).

V EVALUATION

We now reflect back to the original research question: "is it possible to create a performing draughts playing agent by the use of Genetic Algorithms and Neural Networks?". We answer this through the examinations of the strengths and limitations of the system.

A Strengths

Because the agents are essentially playing itself to learn over time, no training data is necessary for the program to evolve itself. This relates to the fact that an infinite number of games can be simulated. The use of crossovers, which differentiates genetic algorithms and its other evolutionary algorithms has shown to improve the quality of the players, especially the case in the 3-ply.

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

The results show a slow learning rate than typical ventures as there is an immediate comparison to make against, and requires many more samples (often exponentially more) to get a weight of equivalent quality. The quality of the neuroevolutionary algorithm boils down to the quality of the fitness function, which in our case is the tournament method. Round-Robin is not necessarily the most efficient (from a number of games played perspective, and not necessarily 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). One of the issues of the approaches taken in the paper is that when compared to agents from much earlier predecessors, some of the characteristics of the neural network used to win against them are lost. This was especially the case for the 1-ply program, where it performs worse than an randomly initialised agent. Creating a high quality fitness function is difficult and its domain specific, or rather, there is no universal fitness function for all problems. This poses as a benefit and a weakness for the neuroevolutionary approach.

The crossover mechanism is a hit and miss, and is shown to impede the learning rate for some of the longer ply-depths. This may be due to its disruptive nature, as heuristics are destroyed. As feared by findings by (Emmanouilidis & Hunter 2000), we can see the potential fears of the negative influence of the textbook crossover mechanism, where it has shown in the results to discourage learning.

C Approach

The modularity of the system implementation was pivotal to debugging the system. The program itself is quite sizeable, and it helps that individual tests can be conducted in most cases. It also helps that the order of the implementation helps.

Measuring the learning rate is quite difficult and arbitrary, and may best be compared to against systems that are classified to play at a particular level.

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 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 mutation, exercising other contemporary methods in machine learning. It may also be the case that more time could be spent in examining references outside of computer science. In this particular example, the algorithms used are inspired by various biological and neuro-scientific understandings. More research into this field may assist in finding more creative methods of solving some of the issues concerning genetic crossover mechanisms and mutation rates.

VI CONCLUSIONS

In terms of fulfilling its objectives, the project was an overall success. A system was implemented using evolutionary methods and it successfully plays checkers in a manner in which it can learn from over time. To the extent of its growth remains suspect however. Efforts to measure the learning rate is one of the non-trivial challenges that makes neuroevolution a particular field of interest, as it is inherent that it can tackle problems in a somewhat novel manner.

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, with little human intervention.

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

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

4. The quality of crossovers can be a deciding factor in how the agents learn to play over time.

The major implication is that it is a possible contender for tasks that require unsupervised learning, and that the fitness function cannot be trivially decomposed into set of derivatives that allow regular derivative based learning to shine.

Provided a high quality fitness function can be designed, the applications of neuroevolution expand beyond its applications in zero-sum games. Neuroevolution in general is useful based on its ability to solve problems without the use of training data, and generating evolving content over time. When applied in a successful manner, neuroevolution can be applied towards the understanding of biological neural networks and the evolution of intelligence itself.

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.
URL: *http://ieeexplore.ieee.org/abstract/document/809083*

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'.
URL: *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.152.7867*

Fogel, D. (2000), 'Evolving a Checkers Player Without Relying on Human Experience', *Intelligence* **11**(2), 20–27.
URL: *http://doi.acm.org/10.1145/337897.337996*

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.
URL: *http://arxiv.org/abs/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.
URL: *http://linkinghub.elsevier.com/retrieve/pii/S0304397516302717*

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

Samuel, A. (1959), 'Some studies in machine learning using the game of checkers', *IBM Journal of Research and Development* **3**, 210–229.
URL: *https://ieeexplore.ieee.org/document/5392560/*

Schaeffer, J. (1997), *One Jump Ahead*, Springer-Verlag.

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.
URL: *http://science.sciencemag.org.ezphost.dur.ac.uk/content/317/5844/1518*

Schaeffer, J. & Lake, R. (1996), 'Solving the Game of Checkers', *Games of No Chance* **29**, 119–133.
URL: *http://library.msri.org/books/Book29/files/schaeffer.pdf*

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.
URL: *http://arxiv.org/abs/1712.06567*

Xie, L. & Yuille, A. (2017), 'Genetic CNN', *arXiv:1703.01513 [cs]* . arXiv: 1703.01513.
URL: *http://arxiv.org/abs/1703.01513*