Playing Draughts using Neural Networks and Genetic Algorithms

Student Name: Thien P. Nguyen

Supervisor Name: Stefan Dantchev

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

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

January 23, 2018

***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 due to their dependency on pre-defined move databases. Optimal moves are pre-calculated, and recalled when necessary. By combining Neural Networks and Genetic Algorithms, this issue could possibly be solved by creating a player that can grow in ability over time, without the dependency on move-banks.

Aims

The purpose of this project is to explore approaches to tackle the game of English Draughts via the use of machine learning techniques. First, we study previous historical successes in the field, and look at the components that helped build their systems. Then, we look at contemporary methods of computer science that could be used to evolve the historical systems. The project will establish whether this approach provides an effective performance on the game.

Method

The initial population consists of randomly generated AI players, which will play each other to determine the best player out of the population. The performance of championing AI players at every generation of the genetic algorithm are measured against previous champions. Appropriate algorithms are implemented to determine the overall development of the system's ability to play Draughts.

Proposed Solution.

The proposed solution starts with designing a feed-forward neural network that evaluates the probability of a particular side winning, given a state of a checkerboard. 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 checkerboards more precisely.

***Keywords* —** Artificial Intelligence, Neural Networks, Genetic Algorithms, Draughts, Machine Learning, 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.

Draughts

English Draughts (or Checkers) is a popular 2-player board-game played on an 8x8 chess board. Players choose a colour and begin with 12 pawns each 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 be forced when both players fail to capture a piece after a set amount of moves.

Genetic Algorithms

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 by the crossover of the genetic information (genome) of their parents. Genomes are metaphors of genetic information; it typically refers to a data structure, most commonly an 1D array. Genetic Algorithms generally consist of a population of agents, a tournament selection process, algorithmic crossover mechanisms against the genomes of the agents, and the introduction of probabilistic mutation on genomes.

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 units 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 through their propagation through the layers of neurons. Learning takes place in the form of the manipulation of the weights connecting the units in the layers. This allows it to model complex relationships between inputs and output, and it can also find patterns in the data.

Motivation

Whilst the use of evolutionary algorithms and neural networks have been explored to create draughts players, my intention is to explore the effectiveness of genetic algorithms and neural

networks. The intention is to determine the possibility of developing a performing draughts playing agent by the use of GANNs (Genetic Algorithms and Neural Networks).

Deliverables

Minimum

- Implement a feed-forward neural network
- Implement a checkers game interface
- Implement a decision making process that chooses a move from a given position of a checkers board.
- Implement a genetic algorithm that uses a population of agents that have their own independently functioning neural network.

Intermediate

- An interface to play against an agent.
- A monte-carlo search of the move space.
- Multi-processing of the tournament selection process for the genetic algorithm.

Advanced

- Evidence shown to indicate that the system is learning over time.
- An agent produced by this process can play to an ELO of at least 1200.
- Agents can be played against human input through a simple user interface.

Related Work

Arthur Samuel pioneered the concept of an self learning program that can play Checkers, through the use of using evolutionary algorithms (Samuel 2000). However, the work described did not consider the use of neural networks (as it was not conceived at the time). It was also dependent on a set of heuristics Samuel devised. This could handicap the agent's ability to play, as they are dependent on the effectiveness of Samuel's heuristics.

The idea of evolving neural networks to play Draughts is based on the success Chellapilla and Fogel had in evolving their own Checkers neural networks (Chellapilla & Fogel 1999). Their work, Blondie24, used a neural network as the evaluator function, and used an evolutionary algorithm (most notably mutation on parents) to evolve the agents. However, crossover mechanisms were not considered (and thus not using genetic algorithms). Blondie24's neural network structure consists of a $\{32, 40, 10, 1\}$ set. Spatial awareness is not considered 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. Relative distance between pieces are not encapsulated in a method that can be immediately interpreted by the neural network. Their

decision making algorithm primarily revolved around a mini-max algorithm with a ply cap. Advancements in the field led to more successful algorithms, which will be explained later on.

Lai's Giraffe (Lai 2015) program used a deep reinforcement learning technique to play Chess to an advanced level in a relatively quick time-period. Giraffe's decision function involved using Temporal Difference Learning (TD-Leaf), an improvement on MiniMax, by Baxter et al. (Baxter et al. 1999) However, Giraffe has been trained on previously played games from grandmaster tournaments as a reference, and was not entirely self-taught.

II DESIGN

Algorithms and Data Structures

.1 Decision Making Algorithm

To choose the best move given a current position, an initial decision making process for an agent could revolve around the use of minimax algorithm. This was the case for Chellapilla and Fogel's agent. (Chellapilla & Fogel 1999). This algorithm would expand a tree of potential moves. Each node represents a state of a checkerboard. Child nodes represent a state of the checkerboard for a given move. Child nodes can only be produced from the parent node; i.e a move from a parent node will produce a child node.

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

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 would help to ignore evaluating unnecessary moves, reducing the number of calculations. Once the possible moves have been considered, the best child node is chosen, representing the best move at a given position.

The mini-max method will have a search depth of 4ply (where the agent will search four moves ahead.) This should allow the agents to form a basic strategy where they can plan their moves in advance. There are inherent trade-offs with having a higher ply count; asymptotic complexity of mini-max is exponential; it being $O(x^y)$ for x being the branching factor, and y being depth. Branching factor will consist of the moves from each agent in a given game. This exponential growth is the achilles heel to the mini-max approach.

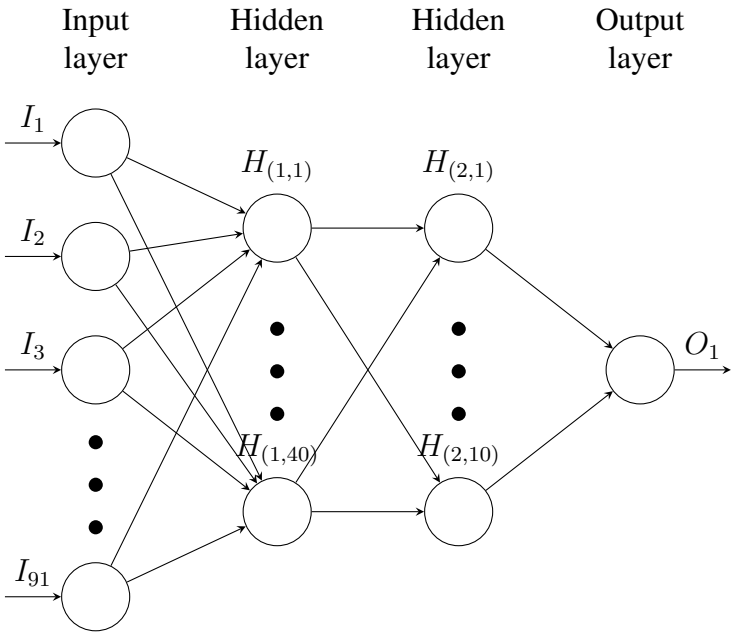
Once initial development on the mini-max algorithm is complete, The project will migrate to a hybrid technique that combines mini-max and a more contemporary algorithmic paradigm, Monte-Carlo Tree Search (MCTS). MCTS differs from mini-max where future moves are randomly played to the end of the game. It acts as a sampling of possible outcomes, and does not depend on an evaluation function at all. The random simulation of games are skewed such that more reasonable moves are chosen. A survey by Browne et al. 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-EPT (or MCTS with early playout termination) is introduced by Lorentz (Lorentz 2016). MCTS-EPT 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 can be used from that caped position instead. The termination ply would be

capped at 6ply. This could potentially improve the amount of foresight for a given set of moves, without the need to depend on random generations of moves to the end, and the need to evaluate more moves (as typically needed even with alpha-beta pruning on a mini-max algorithm).

We retain the minimax algorithm in order to evaluate the performance differences between our two decision making algorithms.

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



.2 Neural Network

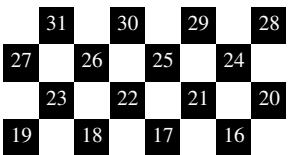
To evaluate the board, we use a feed-forward multilayer perceptron style neural network. The network would contain 4 layers; the input layer consists of 91 nodes, with the output node having 1. Hidden layers will have 40 and 10 nodes respectively. Figure 1 shows the network in the form of a diagram.

Common knowledge infers that a king is worth more than a pawn, but it is disputed about its precise value advantage. For the sake of simplicity, a king's piece value is to be weighted at $1.5x$ a pawn value; a value of a pawn will be 1.

The input array takes in the form of the grid array of the board. The intention is to weigh the Black pawns with a value of 1, and white pawns as -1. To create the input layer, we treat the checkerboard into a 1D array, with the indexes displayed in figure 2. The array is used to calculate all possible sub-squares of the checkerboard, ranging from a 3x3 kernel to a 8x8. Each sub-square is summed up to create an input node. There are consequently 91 combinations of the sub-squares, thus forming the input layer.

Weights of a neuron are summed with their bias, and are passed through an activation function (or transfer function) $O = f((Input * weight) + Bias)$ to become an

Figure 2: The indexes of the 32 pieces of the input layer are the immediate values of the positions on the board.



input for another neuron. Activation functions usually have a sigmoid curve, but may also take the form of other shapes. Common characteristics of activation functions include values to be monotonically increasing, continuous, differentiable and bounded.

Our initial choice for the activation function is tanh, shown in figure 3. There exists inherent issues related to the properties of their derivatives, discussed by Nair and Hinton (Nair & Hinton 2010), described as the missing gradient problem. However, since this issue is related to the properties of a gradient based learning method and not through a stochastic learning method (of which genetic algorithms are), this is not a concern for the project. Sigmoidal function also facilitates the ability to simulate a zero sum formula (for a range of -1 to 1) due to its central symmetric properties.

.3 Genetic Algorithms

The genetic algorithm is the premise of the system's learning strategy. GA's 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 is chosen.

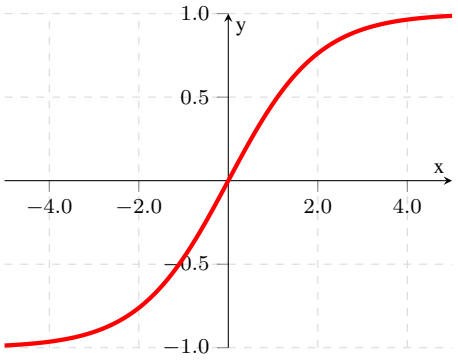


Figure 3: Graph of tanh function $f(x) = \frac{2}{1+e^{-x}} - 1$ (An example of an activation function with a sigmoid curve).

.4 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, with all having the same number of agents. The initial population will consist of randomly generated weights and biases of the neural network, with values from [-1,1].

For a population size of 15, the next generation is created using the best five agents from the current generation. This is discussed in *.5 Tournament Selection*. They will continue to play in the next generation; this strategy is described as elitism. These agents are also used as a base to create 10 new agents from.

The next eight players are generated through the use of crossover strategies, explained in *.7 Crossover Strategy*. The weights of the 1^{st} and 2^{nd} place agents are used as input to the crossover strategy and will generate 4 offsprings. Two are reciprocal crossover representations from each other, and the other two being directly mutated from the parents themselves. Another four children will be created using the same strategy, with the 2nd and 3rd agent's weights. The remaining two will be direct mutations of the 4th and 5th place agents.

.5 Tournament Selection

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

Currently, it revolves around having each agent in the population playing 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 points. Both the agent and its opponent receives a score from the game. Scores are tallied up at the end of the tournament. Players are sorted by the number of points they scored. The best players would have the highest number of points.

.6 Coefficient Mutation

To create variation between agents in preparation for the next generation, we create statistical anomalies 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 will increment by a random value that is created using the following formula, where $WeightP$ 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]. Equation 1 describes the mutation formula.

$$WeightN = WeightP + \frac{m}{\sqrt{2 * \sqrt{K}}} \tag{1}$$

The weights, like the activation function (in 3), will have a soft maximum of [-1, 1]. This would consequently mean that the mutation is not controlled, and dependent on the number of weights in the system; The more weights in the network implies a less significant mutation.

.7 Crossover Strategy

Another learning mechanism provided from the use of 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 offsprings would be created from a pair of parents, with each offspring being the reciprocal crossover of each other. The weights of both parents (now each treated as a 1D array of coefficients), are divided contingent on the number of weights and biases for a given layer. Each layer should be treated separately to reduce the potential dependency on a purely randomly generated neural network. For each set of weights in a given layer, Algorithm 1 describes the crossover process in pseudocode.

Choice of Programming Language

Several contenders exist with each having their inherent benefits. C++ is considered due to it's support for popular machine learning packages (most notably Google's TensorFlow.) In terns of performance, C++ trumps most languages due to its lower level characteristics. C++ has notable parallelised packages (A popular library is OpenMP), which can assist in the overall throughput of the system. However, it would be difficult to write, due to my unfamiliarity with the language. Also, programs written in this language are less portable. It is not suitable for running on university machines without the use of a sandbox.

Algorithm 1 Crossover Strategy

procedure CROSSOVER($parent_1$, $parent_2$)
loop for weights w in layers:
    $n \leftarrow$ length of w
    $i_1 \leftarrow$ *random integer*$(0, n-1)$
    $i_2 \leftarrow$ *random integer*$(0, n-1)$
    if $i_1 > i_2$ **then**
        swap i_1 and i_2
    $weights_{w,child1} = parent_1[0 \dots i_1] + parent_2[i_1 + 1 \dots i_2] + parent_1[i_2 + 1 \dots n]$
    $weights_{w,child2} = parent_2[0 \dots i_1] + parent_1[i_1 + 1 \dots i_2] + parent_2[i_2 + 1 \dots n]$
Return *child1,child2*

Javascript is a contender; Node.JS is a very powerful and utilises a popular package manager, npm. It is difficult to write multi-processed programs as Node.JS runs on a single thread by nature. It also lacks the support of popular machine learning libraries and performs relatively slower in some programming operations.

I have chosen Python 3.6 due to my familiarity with the language, and the support of popular scientific packages, most notably NumPy. Python is also portable with a very wide compatibility; for instance it is pre-installed on all popular UNIX machines and also has support from the university machines. Python development will be on Visual Studio Code, which again is a familiar tool and is also suited to the project.

When writing the system, there will be a heavy dependency on NumPy due to its C++ bindings. This increases the overall speed of the program relative to the performance of standalone python, especially in the case of numerical operations. The neural network would be written using this language, as opposed to the use of machine learning libraries to understand the inner workings of neural networks. Tournaments can be easily parallelised using python's multiprocessing library.

Object Oriented approaches are taken for the majority of the components that constitute the system, ranging from the neural network to the tournament system. Data structures are implemented using their own classes and methods where applicable. The modularity of object oriented programming provides the affordance of easier debugging and testing.

Players weights are stored in two forms, one of which is to be stored on an MongoDB NoSQL instance, and another local copy in JSON. This allows the individual agents to be played against humans.

Tools

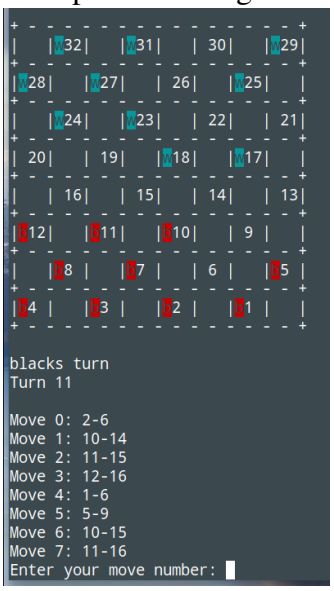
Initial runs will operate on a 1-ply load to determine the stability of the system on a machine containing a 4-core Intel i5 6600u processor with 12GB's of memory. Development and debugging will also occcur on this machine. Once testing has proven to be stable, the system would run on Durham's MIRA (128-core Intel) distributed system with a 6-ply move depth. In order to keep simulations running on MIRA, MOSH is used to maintain a consistent connection to MIRA. The end champion is then transferred to the initial machine in order to be played against by human input. Statistical evaluations and calculations will be calculated on MIRA to reduce computational time.

Interface

As our primary intention is to find an agent that learns to play, creating a relatively friendly user interface is not necessarily important, i.e. a simple text-based interface would suffice. Due to the inherent computational strain the project requires, an interface based around a command-line interface (CLI) is to be used. This allows the system to initiate faster than GUIs (Graphical User Interfaces). CLIs reduces the need for package dependencies, configurations and set-ups opposed to GUIs. The simulation would show statistics such as estimated finishing times, current generation count, scores of players in a given generation and the cumulative score of progress of the system.

When it comes to humans playing against the agent, the checkerboard can also be rendered using ASCII plaintext, users can make inputs through text (in the console or terminal); where the game will show human users the possible moves that a person can take. This system will be used to play against the agent, where the human will take inputs the moves on behalf of the agent's online opponent. A possible rendition is shown in figure **??**.

Figure 4: An example CLI interface demonstrating a game of draughts between user input and an Agent.



Testing and Evaluation

At the end of a given generation, we measure growth of performance by using the generation's champion. Presently we will use the mean of means approach. 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 and -1 for a loss. The weights are scaled differently to the regular tournament in order to accurately portray the difference between previous champions.

Evaluation Method

At the end of the generation run, the end player will be used to compete against human players on various online multi-player checkers websites in order to determine an accurate ELO rating of the system.

The end player is also used to create comparison games. This player is tested against an agent who is choosing random moves, an agent who is using pure monte-carlo tree searches, and an agent from half the generations generations prior to its current state. These statistics will be used as evidence in order to determine whether the agent is learning, and can also be seriously considered as a viable alternative to other machine learning training approaches.

References

Baxter, J., Tridgell, A. & Weaver, L. (1999), 'TDLeaf(lambda) : Combining Temporal Difference Learning with Game-Tree Search', *arXiv:cs/9901001* . arXiv: cs/9901001.
URL: *http://arxiv.org/abs/cs/9901001*

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*

Lai, M. (2015), 'Giraffe: Using Deep Reinforcement Learning to Play Chess', (September).
URL: *http://arxiv.org/abs/1509.01549*

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*

Nair, V. & Hinton, G. E. (2010), Rectified linear units improve restricted boltzmann machines, *in* 'Proceedings of the 27th international conference on machine learning (ICML-10)', pp. 807–814.

Samuel, A. L. (2000), 'Some studies in machine learning using the game of checkers', *IBM Journal of Research and Development* **44**(1.2), 206–226.