Playing Othello Using AlphaZero

Student Name: Ben Adam Rainbow

Supervisor Name: Barnaby Martin

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

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

Abstract —

Context/Background - AlphaZero has recently set the benchmark for all computer game playing algorithms by achieving super human ability at Go. Can these new techniques set out in AlphaZero be applied to games where computers have previously beaten human professionals and improve their playing ability?

Aims - The aim of this project is to implement an AlphaZero program that contains a neural network that can play a series of games with the same neural network architecture and achieve an improvement in play performance over existing techniques on these games, focusing on the game of Othello in particular.

Method - Develop a series of games that allow for a series of players to play against each other. Construct a neural network that can be trained on data generated from within the game and allow predictions to be queried via a web connection to allow any game to query the neural network.

Results - AlphaZero is shown to be a competent and competitive player at Othello against a Monte Carlo tree search and can beat a Monte Carlo tree search with 3x less "$thinking\ time$". AlphaZero also defeats an average experienced player in myself with only 2s of "$thinking\ time$". The AlphaZero player is also competitive across multiple other games including Connect-4 and checkers.

Conclusions - The results have shown that AlphaZero is a competitive player against a Monte Carlo tree search and humans. This performance could be improved through increased training and inference speed. Using TensorRT runtime environment inference speeds can be achieved using quantisation while the use of multiple GPUs or replacing the GPUs with TPUs can cause a massive performance increase.

Keywords — AlphaZero, Deep Learning, Monte Carlo Tree Search, Othello, Connect-4, Checkers.

I INTRODUCTION

Othello was originally invented with the name Reversi in England around 1880 by Lewis Waterman (R.C.Bell 1979). The Game was re-branded and registered under the new name Othello by Tsukuda Original, a Japanese game company. Following the publication of a book entitled "How to Win at Othello" by Goro Hasegawa the game grew in popularity. Since the re-branding Reversi has grown in popularity across the world, especially in Japan. There has been a Reversi World Championships held every year since 1977, with Japan being one of the most decorated countries. (R.C.Bell 1979)

The standard Othello board consists of an 8x8 grid of squares. The starting position is a 2x2 square in the centre of the board with two white pieces in opposite corners and two black pieces in the other corners, as shown in Figure 1. The players then take turns placing a piece of

their colour such that there exist a horizontal, vertical, or diagonal line that connects to a square occupied by their piece where all pieces between the new piece and the old piece are all of the opponents colour, as seen in Figure 2. The opponents' pieces on this line are then flipped to the player's colour. If no legal move persists the player passes their turn and if no player can play, such as when the board is full, the game ends. The player with the most pieces of their colour at the end of the game is crowned the winner. Black is always the starting player.

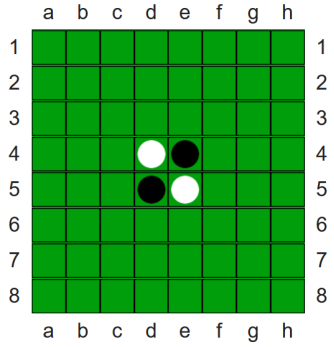

Figure 1: Starting Position for Othello

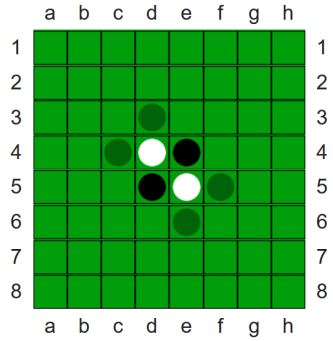

Figure 2: Valid Moves for Black

A Background

The motivation to apply AlphaZero to the problem of Othello is due to its claim of being a general-purpose reinforcement learning algorithm that can learn tabula rasa, without domain-specific human knowledge or data (Silver et al. 2017b). By only playing games against itself AlphaZero was able to convincingly beat world champion programs at the game of chess, shogi and Go. Therefore I plan to implement these techniques on the game of Othello.

A.1 AlphaGo

The first real achievement for the DeepMind team, who authored the paper on AlphaZero, was back in 2015 with AlphaGo. The game of Go has long since been considered as one of the most challenging games for artificial intelligence owing to its enormous search space (Silver et al. 2016). The DeepMind team developed a tree search that used a neural network to guide exploration and exploitation of the game tree. A neural network is used to reduce the effective depth and breadth of a search by evaluating positions using a value network and sampling actions using a policy network (Silver et al. 2016). The value network takes a board as input and outputs the probability of a win from that board while the policy network takes a board as input and outputs a vector of move probabilities, such that the higher the probability the more exploration should be done through that position. The policy network was trained via supervised learning from expert moves which was then optimised using the policy network by adjusting the policy towards the current goal of winning games. The value network is only trained through self play. This approach achieved a win rate of 99.8% against other Go programs and a 5-0 victory against a human European and former world champion Go player. This was the first time a computer program had defeated a human professional player in a full sized game of Go (Silver et al. 2016).

A.2 AlphaGo-Zero

The ultimate goal of artificial intelligence is an algorithm which learns tabula rasa, without human knowledge. Previously the AlphaGo algorithm used supervised learning on expert moves to accurately predict the policy vector for a given board. Expert data set are often expensive, unreliable or often simply unavailable (Silver et al. 2017a). With the implementation of AlphaGo-Zero, however, this all changed. The two previous neural networks, policy and vector, were combined into a single neural network with two output layers creating a single neural network. This neural network is trained over a series of games where at each point the neural network makes an evaluation of the current board and returns a policy vector for the board and its value. The value associated with this node returned from the neural network is propagated back up the tree. After a number of simulations, or an elapsed time period, the best move according to a search function is returned as the next move. At the end of the game the real result of the game Z (1 for win, -1 for loss and 0 for draw) is known. By using backward propagation we can train the neural network to adapt the predictions of the boards seen in this game to reflect the true game value Z. A similar process is repeated for the policy vector such that it reflects the number of visits for a child node divided by the sum of all visits of the parent. The neural network would only be updated if the new neural network wins by a margin of 55% over the old neural network. This algorithm achieved superhuman performance, winning 100-0 against the previously published champion-defeating AlphaGo (Silver et al. 2017a).

A.3 AlphaZero

Up until now the algorithms had been tailored to the properties of a specific game and exploiting them in training. These properties include symmetries in the board on which it is played (Silver et al. 2017b). AlphaZero was developed as a general approach that could solve a series of games at superhuman level without any specific domain knowledge of the games except the game rules. This means unlike AlphaGo-Zero training data cannot be augmented using symmetries and the Monte Carlo tree search cannot flip or rotate the board before evaluating it. This is so the algorithm can accommodate a wider variety of games such as chess which is asymmetric. AlphaZero also only maintains a single player that is updated continually, unlike AlphaGo-Zero where the new neural network had to beat the old network by a margin of 55%. The underlying neural network, however, remains unchanged. The results of this program showed that a single general purpose algorithm could be effective across multiple games as it beat world champion programs in chess, shogi and Go (Silver et al. 2017b).

B Objectives

I plan on using the AlphaZero algorithm to form the basis for an effective Othello player. The effectiveness of this player will be evaluated against a traditional pure Monte Carlo tree search across varying scenarios. First the model for the neural network will be trained over a series of self play games and from this point it will play a series of test games against a Monte Carlo tree search with the same amount of "$thinking\ time$" per move. This means both the Monte Carlo tree search and the AlphaZero algorithm will have the same time to perform as many simulations as possible before selecting what it believes to be the most fruitful move. From this point the "$thinking\ time$" for Monte Carlo will be increased until a point is reached in which it is an equal player to that of AlphaZero over a series of games. The minimum objectives

for this project is to be able to pit a Monte Carlo tree search against AlphaZero such that an evaluation of the algorithm can be performed. From there the intermediate objectives are that AlphaZero can defeat a Monte Carlo tree search with the same amount of "$thinking\ time$" and the advanced objectives are for AlphaZero to win against a Monte Carlo tree search with an increased "$thinking\ time$". I will then prove that my neural network design is true to the AlphaZero algorithm and it is in fact a general reinforcement learning algorithm. To do this I will apply the neural network to a series of games and again measure their performance against a Monte Carlo tree search. The games I have chosen to implement are Othello, Connect-4 and checkers.

II RELATED WORK

Over the years, many different attempts have been made to beat human world champions at Othello. The first came in 1997 with a convincing 6-0 win for Logistello over Takeshi Murakami, the then current world champion. At the time a win for a computer over the game of Go seemed impossible, but now Alpha Go has defeated the Go world champion and therefore I plan to apply these modern approaches to the game of Othello.

A Logistello

Logistello is a program made up of parts, a generalised linear evaluation model (GLEM), ProbCut and an opening book framework. GLEM combines Boolean features linearly. This approach allows an automatic data driven exploration of the feature space (Buro 2002). It does this using a two level feed forward network with binary inputs that 'ands' the inputs together according to some boolean functions and then the second layer takes the sum of the weighted result. This weighted output is then the value for a given board, which tells the program how good a board is. These boolean functions relate to whether the board contains a series of predefined features in a table. The second stage is ProbCut which can prune irrelevant sub-trees with a prescribed confidence (Buro 2002). The process performs a shallow search of the game tree and produces a value for a given node. This value is a good indicator of the value when a deep search of the game tree is performed. Therefore if this value is not as good as other sub trees then the position need not be searched any further as we have already found better candidates. The final part of Logistello is the opening book construction. In the opening plays of the game many search and evaluation functions show great weakness, to mitigate this the program uses a book consisting of move sequences or positions. These moves are consulted in the opening stages of game play. This book is constantly updated when the program loses to include the opponents winning opening sequence in the book. After initially beating the world champion at the time 6-0 Logistello went on to win a straight 22-win victory in its last computer Othello tournament in 1997 as well.

B AlphaZero Family

Logistello, though created almost 20 years prior, shows great similarities to the current state of the art implementations of AlphaZero. Both contain board evaluation functions that help traverse the game tree with elements of learning. The only real difference is that AlphaZero does not use a book of opening moves but its predecessor, AlphaGo, did.

B.1 AlphaGo

AlphaGo was the first computer program to achieve superhuman performance in Go (Silver et al. 2016). An achievement on par with IBM's Deep Blue algorithm win at chess over the current chess World Champion at the time, Gary Kasparov. Previous algorithms had traversed the game tree using hand crafted heuristics and recursively traversing the game tree according to these heuristics. This is infeasible in chess and Go as the search tree is b^d large where b is the game's breadth, the number of legal positions, and d is the depth, the game length. In chess $b \approx 35$ and $d \approx 80$ which is reasonably infeasible but with Go $b \approx 250$ and $d \approx 150$ (Silver et al. 2016). AlphaGo looked to reduce the game tree via two basic principles. The depth of the game tree can be reduced using position evaluation and the breadth can be reduced by sampling actions using a policy (Silver et al. 2016). Position evaluation is the process of removing the sub tree below a given node and replacing it with an approximate value that predicts the value of the game from that state. This has lead to superhuman performance in chess, checkers and Othello (Silver et al. 2016) but due to the large state space of Go it was deemed impossible to achieve superhuman performance with this alone. The breadth of the game tree can then be reduced by sampling actions from a policy. A policy is a probability distribution over the possible moves of a state, these show the probability that a given move should be taken at any point. This provided superhuman performance in backgammon, Scrabble and weak amateur level play in Go (Silver et al. 2016). In AlphaGo these policy and value functions are replaced by neural networks which are trained to approximate the function that describes the policy vector and value of a board. This is done by training the neural network on expert moves over a period of three months. It then combines the neural network with a Monte Carlo tree search to traverse the game tree according to the policy vector and value. AlphaGo achieved 99.8% win rate against other Go programs and defeated the human European Go Champion 5-0 before beating the World Champion 5-0.

B.2 AlphaGo-Zero

Previously the neural network had been trained against expert moves sourced from a database, this however was replaced with self training where the neural network plays against itself to generate training data. This proved to be effective with AlphaGo-Zero achieving superhuman performance, winning 100-0 against AlphaGo (Silver et al. 2017a).

B.3 AlphaZero

The approach above was generalised into AlphaZero where through self play it achieved superhuman performance in many challenging games. Starting from random play and given no domain knowledge except the game rules AlphaZero convincingly defeated a world champion program in the games of chess, shogi as well as Go (Silver et al. 2017b). The results of these games can be seen in Table 1.

	Chess			**Shogi**			**Go**	
	Wins	**Losses**	**Draws**	**Wins**	**Losses**	**Draws**	**Wins**	**Losses**
White	29%	0.4%	70.6%	84.2%	13.6%	2.2%	76.9%	31.1%
Black	2%	0.8%	97.2%	98.2%	1.8%	0%	53.7%	46.3%

Table 1: Results from Deepmind's AlphaZero paper

III SOLUTION

To evaluate the performance of a Monte Carlo tree search against an AlphaZero algorithm on the game of Othello an arena needs to be built. This arena must allow for a set of players to play against each other and themselves. One of these players will be a Monte Carlo tree search which will be implemented to play in the arena. The AlphaZero player will also have to be able to play in the arena and also communicate with the neural network so that it can query the network with a given board. This neural network must not be local to the game of Othello as it must also be available to a series of other games and therefore should be attached to a universal end point that can be queried. Finally the games need to prove the neural networks ability across a series of games and these need to be implemented in the same fashion as the Othello arena.

A Othello Program

The Othello program was written in object oriented Java. Supplementary packages were used throughout the project to further development and make this code as similar to real world production code as possible. Maven allowed me to simplify dependency management through the use of a pom such that I can include other packages with ease. This coupled with the ability to place all the configuration settings in a single properties file really improved the speed of testing as only a few values had to be changed for each test. One of the dependencies I include was Google's Guava, this is a package created by Google that brings in new functionality and types that previously Google had used internally. An example of this are the Immutable types. These types are, as the name suggest immutable, and therefore cannot be changed during execution meaning a new variable has to be instantiated if the variable needs to be changed. Though this may seem counter intuitive it means that important variables such as the list of lists that is the board cannot be changed accidentally. This means that I can be certain that the state of the board is true as it is in an Immutable type and I have test classes to prove its correctness. Throughout the project each class has a set of JUnit and Mockito test classes for unit and integration testing. These verify the functionality of functions at all times during development.

B Players

In this project a variety of players were implemented to help test the ability of the AlphaZero implementation, these players included heuristic, Monte Carlo tree search and human players.

B.1 Heuristic

The game of Othello has two main heuristics, stable counters and mobility. Stable discs are counters that due to their position on the board cannot be changed during the course of the game, a good example of a stable counter is a corner piece in Othello as it can never be between two opponent's pieces on any line. As stable counters cannot change colour throughout the game they are a good indication of who will win the game and therefore a good heuristic to maximise. Mobility is another interesting heuristic in Othello. By maximising the number of possible moves you have per turn and minimising your opponents, you can effectively control the game as you have maximum choice of moves while your opponent can only choose from few and often poor moves. These two heuristics can then be combined using a weighted average to create a player

that maximises the heuristic at each point. The heuristic function used is shown in Figure 3 where x,y, and z have been parametrised as 0.01, 1, and 10 respectively, though this can be modified.

$$h = x * (\#\ counters) + y * (\#\ stable\ counters) + z * (\#\ valid\ moves) \tag{1}$$

Figure 3: Where a stable counter is a counter that will not change throughout the rest of the game, an example would be a corner piece in Othello.

B.2 Monte Carlo Tree Search

A Monte Carlo tree search is a best first search technique which uses stochastic simulations. A Monte Carlo tree search evaluates a position by estimating the average outcome of several random continuations, and can serve as an evaluation function at the leaves of a min-max tree (Coulom 2006). This is how a Monte Carlo tree search can be used as an effective player on the game of Othello. Starting from a point in the game, a series of random games can be played and the results of each game are propagated back up the game tree. Such that at the point a decision has to be made on which child the current node will choose, where each child is a possible valid move, each child will have a pair of statistics such that the next best move can be determined. These statistics are number of wins from this node and the total number of simulations.

A single cycle of the Monte Carlo tree search consists of four stages, selection, expansion, simulation and backpropagation. Expansion is the process of finding a previously unvisited node in the game tree, this is a node in which a simulation has not been started in previously, and selecting it for simulation. For the rest of the game actions are selected at random until the end of the game is reached and the end result of the game is recorded. This is the simulation phase. Then this value Z of the game (1 for win, -1 for loss, 0 for draw) is propagated back up the game tree back to the root of the tree. This means the number of simulations statistic is increased in all nodes visited in the simulation by one and the game result is added to the total number of wins statistic. The next node for simulation is then selected in the selection phase, this has to be done in such a way to balance both exploitation of researching good nodes to understand more for later games and also exploration such that less explored nodes are checked to remove uncertainty in their evaluation. This repeats for a number of cycles or until a time limit has elapsed from which the best child node of the current node, also known as the best possible move from the current board, is picked greedily to maximise the chance of winning. It does this by selecting the next move x according to the Upped Confidence Bound as defined below.

$$x = argmax\left(\frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log N(v)}{N(v_i)}}\right) \tag{2}$$

Where $Q(v)$ represents the number of wins associated with v, $N(v)$ represents the number of simulations associated with v, v_i represents a child in the tree of v and c represents an exploratory constant.

A Monte Carlo tree search has been shown in the past to generate the most powerful Go program at the time (Chaslot 2008). A Monte Carlo tree search has also been shown to be effective across numerous classical board games as well as modern board games and video games as an alternative to alpha-beta pruning. Alpha-beta pruning is a strategy for reducing the number of

evaluations performed during a search of a tree by applying a minimax algorithm. The minimax algorithm is a recursive algorithm which associates a value for each state of the game which indicates how good that state is using a heuristic function. For alpha-beta pruning to be an effective player an adequate evaluation function must exist and the game must have a low branching factor (Chaslot 2008). Therefore a Monte Carlo tree search is an excellent benchmark for AlphaZero.

B.3 AlphaZero

AlphaZero takes the basis of the Monte Carlo tree search and changes only two of the stages, selection and simulation. In the simulation phase instead of performing these random game roll outs, the algorithm instead makes a call to the neural network that returns a policy vector and a value for the given board. The value is propagated back up the game tree, much like the Z value in MCTS, and added to $Q(v)$. In the selection part the function for choosing the next move, x is adapted to include the policy vector to create the PUCT function.

$$x = argmax \left(\frac{Q(v_i)}{N(v_i)} + cP(v, v_i) \sqrt{\frac{N(v)}{1 + N(v_i)}} \right) \tag{3}$$

Where $P(v, v_i)$ is the probability of selecting a move from the current node v to the child v_i.

C Neural Network Architectures

The design of the neural network is the most important part of this project as the network must train effectively and produce high quality policy vector value pairs for a given board, but it must do this in a timely fashion. The larger a neural network, the more accurate at predicting the value of a given board the network will be. This is at the cost of the time taken for the board to propagate through the network. The longer the prediction time the less simulations the AlphaZero algorithm can achieve within its search window. Therefore, a trade off has to be made between the prediction accuracy on single board and the speed of prediction. This is because the Monte Carlo tree search part of the algorithm can compensate for a lack of accuracy in the neural network if it can perform enough simulations.

To test whether the algorithm is a true implementation of AlphaZero multiple games will have to communicate with the neural network. To achieve this the neural network is written in PyTorch, due to its prevalence and performance in the field, and hosted behind a Flask web server that accepts restful API calls for predictions and training. This allows multiple different programs to query the neural network allowing me to prove that my implementation is a general purpose reinforcement learning algorithm.

C.1 Convolutional Neural Networks

Recent work in image recognition has demonstrated considerable advantages of deep convolutional networks over alternative architectures. The power and generality of large and deep convolutional neural networks suggests that they may do well on other "visual" domains including playing computer games (Maddison et al. 2014). A convolutional neural network consists of a series of convolutional layers. Convolutional layers take an input and perform a convolution by sliding a filter across the input and produce an output by performing the dot product of this

filter and the input. The weights of this filter are learnt through training the neural network, this is a process of using back propagation to minimise the difference between the expected value and the value produced by the neural network according to a function. The advantage of using convolutional layers over linear layers is that information from spatially relevant cells can be considered when assessing one value, but most importantly it reduces the number of parameters of the neural network. This means that we can apply more layers in the network than a traditional fully connected neural network.

The initial architecture of this project was a 4 layered convolutional neural network where each convolution is followed by a batch normalisation then linear layers to reduce the output to a policy vector and a separate value output. This was a good starting point for the project and showed that with training on a 6x6 board of Othello that the AlphaZero player could outperform a random player 10-0. This showed that the neural network was learning some strategies of the game of Othello. When it came to competing with the Monte Carlo tree search, however, the AlphaZero player would need multiple integer times more "$thinking\ time$" to compete with the Monte Carlo tree search. Therefore a more accurate network was needed.

C.2 Residual Neural Networks

In the jump from AlphaGo to AlphaGo-Zero not only did the policy and value neural networks become one network but they also changed architecture. Previously AlphaGo had used 12 convolutional layers each for each neural network this was then adapted into a 40 block residual neural network. A residual neural network consists of blocks where each block is a series of convolutional layers, batch normalisation and activations. At the start of each block there are two paths, one path completes all the layers of the block while the other skips all the layers in the block. These two outputs are combined as the output of the block. These blocks are then placed together in series to form a residual neural network. Residual networks were found to be more accurate, achieving lower error and improved performance against AlphaGo (a convolutional neural network) by over 600 Elo, Elo is a rating system that calculates the relative skill levels of players in zero-sum games. Combining policy and value together into a single network slightly reduced the move prediction accuracy, but reduced the value error and boosted playing performance against AlphaGo by around another 600 Elo. This is partly due to improved computational efficiency (Silver et al. 2017a). In the AlphaGo-Zero paper the network consisted of a convolutional block at the start of the network followed by 19 or 39 residual blocks which were made up of 2 convolutional layers followed by batch normalisation and rectifier non-linearity layers.

D Training

The most successful approaches are trained directly from the raw inputs, using lightweight updates based on stochastic gradient descent. By feeding sufficient data into deep neural networks, it is often possible to learn better representations than hand-crafted features (Mnih et al. 2013).

In AlphaZero we do this by pitting the neural network against itself and recording the board and policy vector at each point in the game tree. These boards and policies are then fed back into the neural network and the loss from the value predicted for a given board and the eventual end result is minimised. The result we are minimising is $\frac{Q+Z}{2}$. The end result of the game is denoted

by Z and we average this value of Z with the value of Q which denotes the average expected result for a given node, $Q = \frac{wins}{simulations}$. This true value is minimised against the predicted value using the mean squared error which measures the average of the squares of the errors and using backpropagation this error is minimised.

$$\frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 \tag{4}$$

The policy vector is trained against the fraction of simulations that went through a specific child and has the loss function of binary cross entropy.

$$\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i)) \tag{5}$$

This process is then repeated over a series of games until eventually the network can effectively distinguish a good board from a bad board using the value output and for each board give a good suggestion of the next move using the policy output.

D.1 Batch Learning

Training initially was done after every game in game order. This means the network always received the same start board first at every training interval and then would get boards in a predictable fashion. This generally caused the neural network to learn local minimums and when competing against a Monte Carlo tree search the lack of ability of the neural network started to show. This was adapted such that the neural network would train every x number of games, then it would randomly sample a batch of size y and train on that data set. This would repeat until the number of batches multiplied by the batch size was equal to the size of the training data. This would be one epoch and would repeat for z epochs.

D.2 Dirichlet Noise

By adding random noise to the search policy we can avoid getting stuck in local minima, especially after the first few rounds of training. After the first few rounds of training, the neural network has only taken a few paths down the game tree and will weight them heavily based on its previous training data. By adding noise, local minima can be avoided and new promising paths can be discovered. Dirichlet noise is used because it outputs a vector that sums to 1, therefore we can combine it with the previous policy vector and still have a vector that sums to 1. Dirichlet noise is also parameterised with alpha which determines over the vector how spread the values should be. If we have a vector of size 3 and a small alpha, the vector produced will have a higher probability to look like (0.98,0.01,0.01) while if the value of alpha is higher it is more likely to be (0.3,0.3,0.4). This means we can use alpha to control the amount of exploration in the Monte Carlo tree search. To add this noise to the previous policy value we follow the equation below to calculate the new policy.

$$x * p + (1 - x) * d \tag{6}$$

Where p is the previous policy vector, d is the Dirichlet noise and x is a hyper parameter set to 0.75 in the AlphaZero paper.

E Playing

During play, exploitation and exploration needs to be appropriately balanced at different points in the game. At the start of the game you want to understand as much as possible about as many branches in the tree as possible, even if only at a shallow level. This is so that many possible moves can be evaluated and the best move chosen. When nearing the end of the game, however, there will only be a few possible moves all with lots of prior simulation work already completed. At this point we can see from prior simulations which child is more fruitful based on its statistics. We therefore want to fully explore these fruitful choices to cement the theory that they are in fact fruitful, this means we want to weight move selection towards exploitation rather than exploration.

E.1 Temperature

In the AlphaZero algorithm we use the equation below to select a move at each point in the tree and in the game.

$$x = argmax\left(\frac{Q(v_i)}{N(v_i)} + cP(v, v_i)\sqrt{\frac{N(v)}{1 + N(v_i)}}\right) \tag{7}$$

We can divide the equation into two parts. $\frac{Q(v_i)}{N(v_i)}$ can be considered the exploitation part of the equation as it considers children with a high probability of wins over those with lower probability. While the second half $cP(v, v_i)\sqrt{\frac{N(v)}{1+N(v_i)}}$ can be thought of as the exploration part of the equation as it favours children with less simulations. We want to control the amount of exploration over exploitation throughout the execution of the program. We do this by introducing the variable τ to create the formula below.

$$x = argmax\left(\frac{Q(v_i)}{N(v_i)} + \tau \cdot cP(v, v_i)\sqrt{\frac{N(v)}{1 + N(v_i)}}\right) \tag{8}$$

For a number of turns τ is set to 1 such that exploration and exploitation are equal. After a number of turns that is predefined τ is set to an infinitesimally small number such that the exploration section of the equation only has the affect of settling draws. The number of turns until this happens is a hyper-parameter and therefore can be optimised. By weighting the exploration portion of the equation infinitesimally small the equation now weights move selection almost entirely on the fruitfulness of the children. This is perfect for end game situations where we only want to maximise the end result and care very little about exploring children with fewer simulations. This is because there are very few possible paths this late in the game and it is unlikely if 90% of the paths lead to a loss that the small set of unvisited paths lead to a win.

F AlphaZero - A General Reinforcement Learning Algorithm

AlphaGo-Zero achieved superhuman intelligence specifically in the game of Go. With AlphaZero we generalise this approach into a single algorithm that can achieve superhuman performance in many challenging games (Silver et al. 2017b). To show this I have taken the game framework from Othello and re-purposed it to the game of Connect-4 and checkers to allow AlphaZero to learn over three domains.

IV RESULTS

In this section I will evaluate how to maximise the playing strength of the AlphaZero player against a fixed Monte Carlo tree search player on the game of Othello. All of these experiments take place on a single consistent machine with specifications shown in Table 2. Both players were given 500ms of "$thinking\ time$", this means they have a maximum time of 500ms to perform as many calculations as they can before having to return a move to play. This is fixed throughout all experiments in this section unless stated otherwise. To compare the Monte Carlo tree search player against the AlphaZero player a series of 50 games is played in which the AlphaZero plays as black and the Monte Carlo tree search plays as white. This gives the AlphaZero player an advantage as black always goes first and this has been shown to be an advantage in Othello. This advantage, however, is consistent throughout the tests and therefore should have no effect on the results of experiments. A Monte Carlo tree search is a stochastic process and therefore two runs of the same values may produce different results. To mitigate this the overall performance of AlphaZero over these 50 games will be defined as the win ratio $w = \frac{wins}{losses - draws}$ from the perspective of the AlphaZero player. This should help mitigate against the effects of games that deviate from the global trend.

Using this set up I evaluate model optimisation and parameter optimisation for the AlphaZero player. Throughout this process if one parameter is being varied all other parameters are fixed to show the effects of the changing parameter more clearly. The parameters that will be optimised over these experiments are the model size of the neural network, $Cpuct$ and $temperature\ threshold$. The model size is the number of residual blocks in the neural network and $Cpuct$ defines the amount of exploration over exploitation for the Monte Carlo tree search. More exploration will look at more moves in the near future, while exploitation will focus on only the fruitful in the near future to allow it to look at more moves in the distant future. The $temperature\ threshold$ is the number of moves a player makes before it stops exploration.

Component	**Title of Component**
CPU	Intel Core i7-9700 @ 3.00GHz
GPU	GeForce GTX 1660 Ti 6GB GDDR6
RAM	8GB DDR4
OS	Windows 10 64bit

Table 2: Specifications of PC used to perform all experiments and training

A Model Optimisation

For the neural network it was proven in DeepMind's AlphaGo-Zero paper that a residual neural network outperforms a traditional convolutional neural network (Silver et al. 2017a). Therefore I have implemented a residual neural network which consists of an input layer, output layer and a series of blocks that lie between these two layers. Therefore, it is important to define how many blocks in the neural network are optimal. While training and evaluating the model the values of $Cpuct$ and $temperature\ threshold$ are fixed at 4.0 and 25 respectively throughout both phases. These values were selected so that diverse games are played to generate varied training data. The residual network used to play the game of Go in the DeepMind paper consisted of 40 blocks with 38 internal blocks (Silver et al. 2017a). Therefore, as the game of Go is far more

complex than Othello the maximum neural network size evaluated for Othello had 15 internal blocks. This 15 internal block neural network was evaluated against neural networks with 5 and 10 internal blocks at training intervals of 500 games, 1000 games and 1500 games. The neural network is trained for 100 epochs every 20 games, meaning each neural network had been trained 75 times by the final game and was evaluated after every 25 training instances.

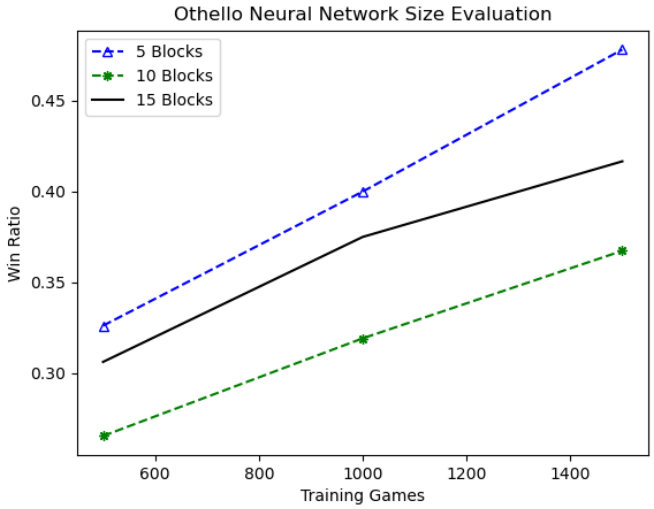

Figure 4: Graph of neural network size evaluation results

Blocks	**500 Games**				**1000 Games**				**1500 Games**			
	Wins	***Losses***	***Draws***	***Win Ratio***	***Wins***	***Losses***	***Draws***	***Win Ratio***	***Wins***	***Losses***	***Draws***	***Win Ratio***
5	15	31	4	0.3261	20	30	0	0.4000	22	24	4	0.4783
10	13	36	1	0.2653	15	32	3	0.3191	18	31	1	0.3674
15	15	34	0	0.3061	18	30	2	0.3750	20	28	2	0.4167

Table 3: The evaluation of the neural network size with fixed training and evaluation parameters of 40, 25, 500ms for $Cpuct$, $temperature\ threshold$ and "$thinking\ time$" respectively.

This shows that though all models improve with training, the 5 block and 15 block generate the best results. I believe this is due to the fact that the 5 block neural network is very quick at propagating values through the neural network due to its small size and therefore can perform more simulations than the 15 block neural network in a set amount of time. The 15 block may do fewer simulations, therefore it must be more accurate in its predictions than the 5 block neural network to overcome this shortcoming of the network propagation speed. I believe this is why the 10 block neural network did not perform as well as its larger or smaller counterpart as it was not quick enough to perform enough simulations, nor accurate enough in the simulations it did perform. The performance difference between 5 blocks and 15 blocks is consistent when the "$thinking\ time$" for both adversaries is increased to 1000ms for the 50 games, with all other parameters staying the same. The results of this test show that the gap in performance does not decrease even with greater "$thinking\ time$", it actually increases as shown in Table 4. Overall, this experiment shows that a 5 or 15 block network would be a reasonable adversary to a Monte

Carlo tree search but due to the longer training times associated with 15 blocks and the slight performance increase in the 5 block network I have chosen to persist with the 5 block neural network from this point.

Blocks	**1500 Games**			
	Wins	***Losses***	***Draws***	***Win Ratio***
5	18	31	1	0.3674
15	12	37	1	0.2449

Table 4: Increased "$thinking\ time$" confirms that 5 blocks is still more effective than 15 blocks.

B Parameter Optimisation

The two parameters that need to be optimised for play are $Cpuct$ and $temperature\ threshold$. These two parameters control the level of exploration and the number of moves for which this exploration should be considered respectively in the PUCT function described in Figure 5. This means for effective play against an adversary these values need to be optimised to improve playing performance. In these experiments the neural network is kept consistent as the 5 block neural network trained for 1500 games from above with a fixed "$thinking\ time$" of 500ms.

$$x = argmax\left(\frac{Q(v_i)}{N(v_i)} + \tau \cdot cP(v, v_i)\sqrt{\frac{N(v)}{1 + N(v_i)}}\right) \tag{9}$$

Figure 5: PUCT function where c represent $Cpuct$ and τ represents $temperature\ threshold$

B.1 Temperature Threshold

The $temperature\ threshold$ controls how many moves the PUCT function considers on the exploration side of the equation. If the number of current moves for a player is less than the $temperature\ threshold\ \tau = 1$. The exploration side of the PUCT equation is considered equally to the exploitation side of the equation. Once the player's number of moves reaches and exceeds the $temperature\ threshold$ then $\tau = \frac{1}{\infty}$. This causes the exploration side of the equation to have almost no effect when deciding which move x to select next. The only effect it has is to settle draws between nodes with identical exploitations values.

The results of varying $temperature\ threshold$ with $Cpuct$, as shown in Figure 6, show that as the value of the $temperature\ threshold$ changes for different $Cpuct$ values the trend in win ratio is consistent across multiple $Cpuct$ values showing an independence of the parameters $Cpuct$ and $temperature\ threshold$. One clear spike in performance is around the 15 move mark which represents roughly the halfway mark in the game, as on average each player makes 16 moves in Othello. All $Cpuct$ values greater than 1 showed their peak performance at a $temperature\ threshold$ of 15 moves. The $temperature\ threshold$ will now be fixed at 15 moves.

B.2 Cpuct

$Cpuct$ is the weighting between the exploitation and exploration part of the PUCT function, shown in Figure 5. This function decides which move x should be selected next. A higher value of $Cpuct$ means the right side of the equation, the exploration part, has greater weighting and therefore the function favours moves that have not been explored as much. A smaller $Cpuct$ value means the exploration side is weighted less and therefore the left side, the exploitation part of the equation, is more dominant causing stronger moves to be explored more.

In this experiment the value of $Cpuct$ was varied from 2 to 8 with a fixed $temperature\ threshold$ of 15 moves. The resulting graph in Figure 7 shows how performance increases to a point after which it starts to decline. This point is a $Cpuct$ of 5 which will be used for all future evaluations. The graph shows that exploration is an important part of searching the game tree as you can see a steep increase in performance as $Cpuct$ is increased and thus exploration is increased. Exploring too much, however, can be seen to have a detrimental effect on the playing ability as shown by the decline in playing ability from a $Cpuct$ value of 5 onwards. This shows that too much focus on exploration and not enough focus on exploitation leads to a decrease in playing ability.

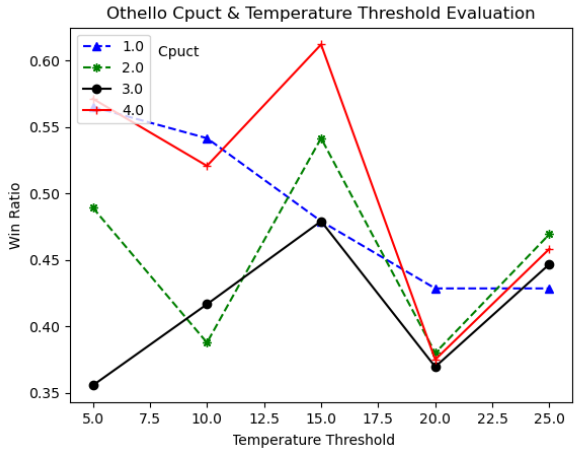

Figure 6: The win ratio for AlphaZero using a 5 block residual neural network with 500ms "$thinking\ time$" and varying $Cpuct$ and $temperature\ threshold$.

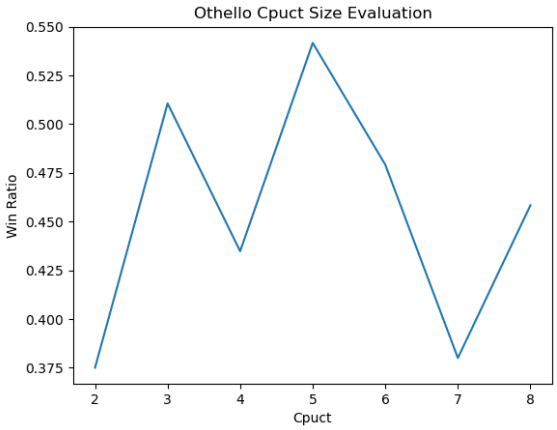

Figure 7: The win ratio for AlphaZero using a 5 block residual neural network with 500ms "$thinking\ time$", $temperature\ threshold$ of 15 and a varrying $Cpuct$.

C Training

In this experiment I evaluated the performance of a model trained over three 500 game intervals trained and evaluated using the parameters assigned above of 5.0 for $Cpuct$, 15 for $temperature\ threshold$ and a 500ms "$thinking\ time$" on a 5 block residual neural network. In this experiment I will be comparing the performance of a model after training intervals of 500, 1000 and 1500 games using different training targets for the value of a given board. I will be comparing the performance of training against Z or an average of Q and Z, where Z is the result of the completed game and Q is the expected result for a given node where $Q = \frac{wins}{simulations}$. It

seems intuitive to say that a board is as good as the result it leads to. So if a board is played, the neural network should use back propagation to minimise the value between the predicted value and the final game result. Due to the fact that a game is heavily influenced by randomness however this can be ineffective. A strong move at the beginning which is followed by a series of weak moves will be penalised when training against Z if this move lead to a loss even though it was the fault of later moves. To combat this we train the network on the average between Z and Q. By training against an average of Z and Q we combat the downfalls of training against either of the values individually but keep the positive aspects of both solutions. Using a training target of $\frac{Q+Z}{2}$ shows an increase in performance compared to using a training target of just Z alone, this is shown in Figure 8.

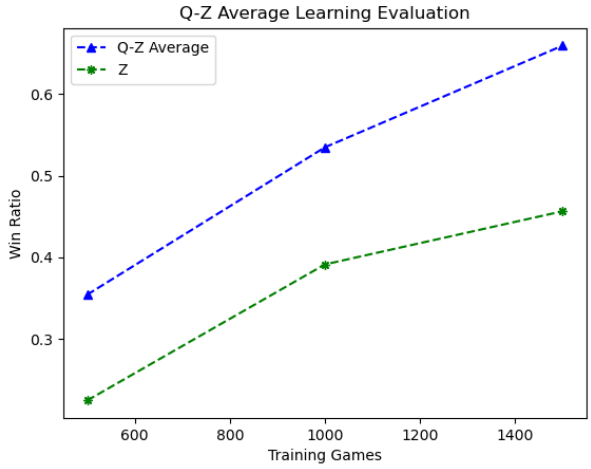

Figure 8: Varying training targets evaluated with 5.0 for $Cpuct$ and 15 for $temperature\ threshold$ using a 5 block residual neural network with 500ms "$thinking\ time$".

V EVALUATION

In this section an evaluation is made of an AlphaZero player against various opponents which include a heuristic, a Monte Carlo tree search and a human player. These players will be evaluated against a fixed "$thinking\ time$" AlphaZero player that has been trained on 5000 games of self played Othello with the parameters of 5.0 for $Cpuct$ and 15 for $temperature\ threshold$ and will play Othello with these same parameters.

A Heuristic Adversary

The heuristic player I have implemented, unlike a Monte Carlo tree search, looks only at the next possible move and uses a heuristic function to decide which of those boards maximises the heuristic function. The board that maximises the heuristic function defines which is the best move to make according to the heuristic function in Figure 3.

The heuristic player is a strong player as it is not time limited and is allowed to look at all possible moves and evaluate each one before making a decision on which move to make.

This is shown by its score on Othello against AlphaZero with 500ms *"thinking time"*. When AlphaZero plays as the black it wins 29-20 with 1 draw over 50 games and when it plays as white it wins 43-5 with 2 draws.

B Monte Carlo Tree Search Adversary

The Monte Carlo tree search is the most important evaluation as Monte Carlo tree search players have achieved Dan (master) level play at 9x9 Go and therefore are a true adversary to the program. As AlphaZero is based on a Monte Carlo tree search this evaluation will also prove whether the neural network outperforms random simulations. For this evaluation on Othello the AlphaZero player will have a fixed 500ms *"thinking time"* and the Monte Carlo tree search will get increasing *"thinking time"* until its play equals that of AlphaZero. The wins, losses and draws will be recorded over 2 sets of 50 games where both players will play 50 games as white and 50 as black and the results will be compared. The results of this evaluation are summarised in Figure 9.

It shows how the AlphaZero player defeats the Monte Carlo tree search convincingly at both 500ms and 1000ms where AlphaZero wins on average 69% of the games at 500ms and 58% at 1000ms while playing as black and white respectively. At 1500ms the limit of AlphaZero with 500ms *"thinking time"* is reached and AlphaZero narrowly beats MCTS with a win ratio of 52% for black and 51% for white. This shows with only 5000 training games that AlphaZero can outperform a Monte Carlo tree search three fold with just 5000 games of training.

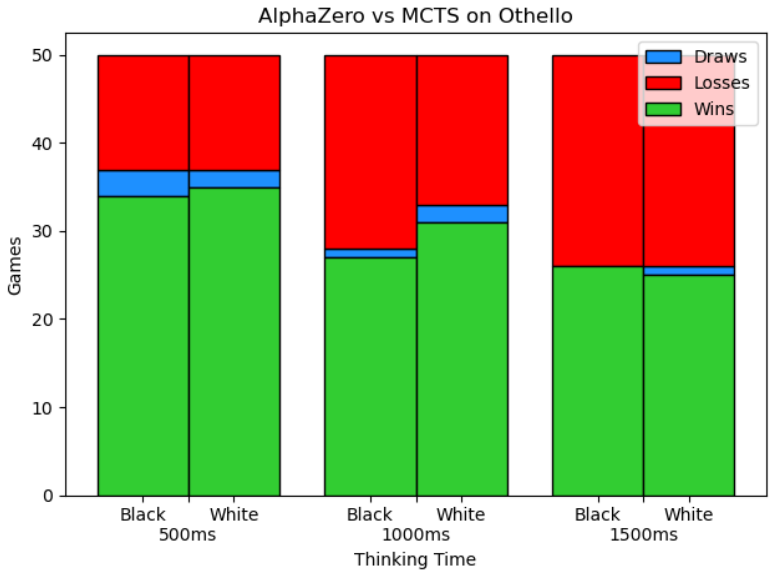

Figure 9: The results of a competition between AlphaZero and a Monte Carlo tree search, where AlphaZero has a fixed 500ms *"thinking time"* against a Monte Carlo tree search with increasing *"thinking time"*

C Human Adversary

I have been playing Othello for over 10 years and I would class myself as an average experienced player. Using the same set up as AlphaGo's game against Lee Sedol, I played a best of 5 games of Othello against AlphaZero first with 1500ms "$thinking\ time$". The advantage is given to AlphaZero by allowing it to play black first before swapping after each game. This resulted in 2 wins for myself, 2 wins for AlphaZero and 1 draw, with both my wins coming from playing as black. This shows that with only 1500ms that AlphaZero is equal in ability to myself. I did however notice AlphaZero's inability to play as white against a human. This is due to white being the weaker colour and that it requires slightly more computation in my implementation of AlphaZero to play because the board must be normalised for the neural network. When the "$thinking\ time$" is increased to 2000ms this downfall is mitigated and in a series of 5 games where I had the advantage for all games by playing as black, I lost to AlphaZero 3-2. This shows that with a "$thinking\ time$" of only 2000ms and with only 5000 training games AlphaZero has become an experienced average player.

D AlphaZero

The above evaluation has proved that AlphaZero is an effective Othello player with a relatively small number of training games. For an algorithm and neural network to be an AlphaZero implementation it must be applicable to other domains. To do this I have created two identical neural network to the one used for Othello and trained one on 2000 games of self play on Connect-4 and the other on 500 games of self play on checkers.

Connect-4 is a less complex game than Othello and it has been proven that the player that goes first, red in my implementation, can always win (Allis 1988). After 2000 games of self play on Connect-4 an evaluation of the AlphaZero player with a fixed 500ms "$thinking\ time$" against a Monte Carlo tree search was completed over 100 games. Each player played as red for 50 games and as yellow for 50 games, so that no player had an advantage. When both players had 500ms of "$thinking\ time$" AlphaZero outperformed MCTS 81-19. When the "$thinking\ time$" for MCTS was increased to 1500ms AlphaZero still outperformed MCTS 61-39. This shows an improvement on the results from an equivalent evaluation of Othello where AlphaZero only managed a 51.5% win ratio compared to the 61% win ratio in Connect-4 under the same parameters. AlphaZero can outperform MCTS up to 2000ms on Connect-4, that is 4x more "$thinking\ time$" for the MCTS, where a win ratio of 54% was achieved. AlphaZero is also a strong player against a human adversary on Connect-4, with only 500ms "$thinking\ time$" the AlphaZero player is equal to my ability where a series of 10 games lead to a draw. This is also an improvement over Othello where AlphaZero required 2000ms to achieve supremacy over myself.

Due to the complexity of checkers and the size of its state space the Monte Carlo tree search player becomes incredibly weak and struggles to finish a single simulation. To combat this, a maximum number of moves to find a terminal node is set before a basic heuristic of number of counters is returned. In a series of 20 games, AlphaZero with five times less "$thinking\ time$" than MCTS, gives an overall result of 6-5 with 9 draws, where a draw consists of neither player gaining an advantage in 100 moves. The dominant white player was shared equally.

These results over MCTS and humans across multiple domains shows that this algorithm is in fact an AlphaZero implementation. It can be applied to a series of domains and achieve success in multiple domains while only learning tabula rasa.

VI CONCLUSIONS

In this project I have developed an AlphaZero based player for the game of Othello that can outperform a strong Monte Carlo tree search player with three times less "$thinking\ time$" than its adversary. The AlphaZero player can also outperform myself with a relatively small "$thinking\ time$". This has all been achieved tabula rasa, without human knowledge, and has shown that the same neural network architecture can be applied easily to a series of games and achieve success in those domains. This includes defeating myself and MCTS with 4x more "$thinking\ time$" on Connect-4. Defeating MCTS with 5x more "$thinking\ time$" on checkers all while only having 500ms of "$thinking\ time$". This proves that my implementation is not only an effective player compared to heuristic, Monte Carlo tree search and human players on Othello, but also on Connect-4 and checkers. This shows that this is in fact a single general purpose algorithm that is effective across multiple games, which is the main characteristic of an AlphaZero implementation.

The solution could be improved using the same hardware, however, there are many other improvements that the current solution could not take advantage of including runtime engines and better hardware. David Silver, the lead author of AlphaGo, AlphaGo-Zero and AlphaZero states in a podcast with Lex Fridman that the only limit to these algorithms is time and computational power (Fridman 2020). This is because the neural network can always learn more about a game, all it requires is the time and power to do so, therefore these improvements are all about increasing inference and training speed via runtime and hardware improvements. If the hardware was to be limited to my current set up there is only one improvement that can be made which is using TensorRT. NVIDIA TensorRT is an SDK (Software Development Kit) for high-performance deep learning inference. It includes a deep learning inference optimiser and runtime that delivers low latency and high-throughput for deep learning inference applications (Nvidia 2020). By running a model in the TensorRT environment it has been shown to improve inference times for neural networks by at least 10% and up to 300% if INT8 quantisation is used compared to standard TensorFlow. This quantisation reduces the accuracy of weights within the neural network which increases inference and training speed at the cost of accuracy.

The improvements can also be achieved by increasing the amount and type of hardware used to evaluate and train the neural network. By creating a distributed system of GPUs the Monte Carlo tree search can be run in parallel across multiple devices each running their own simulations before combining the results at the end of a time period. This performance can then be improved further by switching out the GPUs for TPUs. Tensor Processing Units, TPUs, are Google's custom-developed application-specific integrated circuits used to accelerate machine learning workloads (Google 2020). These TPUs have been shown to be at least 30x faster than a state of the art GPU (Jouppi et al. 2017). Far superior to the single consumer GPU I used for training and evaluation.

References

Allis, L. V. (1988), 'A Knowledge-Based Approach of Connect-Four.', *ICGA Journal* **11**(4), 165.

Buro, M. (2002), 'Improving heuristic mini-max search by supervised learning', *Artificial Intelligence* **134**(1), 85 – 99.

Chaslot, G. (2008), 'Monte-Carlo Tree Search: A New Framework for Game AI', *Artificial Intelligence and Interactive Digital Entertainment Conference* .

Coulom, R. (2006), 'Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search'.

Fridman, L. (2020), '#86 – David Silver: AlphaGo, AlphaZero, and Deep Reinforcement Learning'. (Accessed on 15/04/2020).
URL: *https://youtu.be/uPUEq8d73JI*

Google (2020), 'Cloud Tensor Processing Units'. (Accessed on 15/04/2020).
URL: *https://cloud.google.com/tpu/docs/tpus*

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A. et al. (2017), 'In-datacenter performance analysis of a tensor processing unit', *in* 'Proceedings of the 44th Annual International Symposium on Computer Architecture', pp. 1–12.

Maddison, C. J., Huang, A., Sutskever, I. & Silver, D. (2014), 'Move Evaluation in Go Using Deep Convolutional Neural Networks'.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. A. (2013), 'Playing Atari with Deep Reinforcement Learning', *CoRR* .

Nvidia (2020), 'NVIDIA TensorRT'. (Accessed on 15/04/2020).
URL: *https://developer.nvidia.com/tensorrt*

R.C.Bell (1979), *Board and Table Games from Many Civilizations*, 2nd edn, Dover Publications.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. (2016), 'Mastering the Game of Go with Deep Neural Networks and Tree Search', *Nature* **529**(7587), 484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. & Hassabis, D. (2017b), 'Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm'.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. & Hassabis, D. (2017a), 'Mastering the Game of Go without Human Knowledge', *Nature* **550**, 354.
URL: *http://dx.doi.org/10.1038/nature24270*