Reinforcement Learning for Game-Playing: Experiments with AlphaZero

Student Name: Stephen Church

Supervisor Name: Dr. Barnaby Martin

April 30, 2019

Abstract —
Background/Context: Reinforcement learning is a growing field within machine learning which aims to answer the question of whether machines can learn from interacting with an environment without the need for human intervention. AlphaZero is one of the most well-known successes in this field, however it was trained using a lot of computing resources, and its results are hard to verify. Furthermore, few experiments have been done to test the effect of different learning parameters and neural network architectures and this project explores how these affect results. Finally, many recent successes within computer vision have involved using pre-trained networks for new tasks, and this project aims to explore how this could be used to speed up the training of AlphaZero on more complex tasks.

Aims: The aims are: to implement AlphaZero and train the network on limited hardware for two board games to verify its effectiveness; to experiment with parameters and neural network architectures during training to understand their effect on the algorithm; to explore whether traditional fine-tuning methods can be used to scale up to bigger boards.

Method: AlphaZero has been implemented following the original research papers. A player using only Monte Carlo Tree Search (MCTS) has also been implemented for comparison. AlphaZero has been trained on Connect4 and a more complicated game - Pathwayz. Tests have been performed using different parameters and two different neural network architectures have been implemented (one using a convolution neural network, one using a residual convolution neural network) to broaden understanding of parameters and their effects on results. For both Connect4 and Pathwayz, a tournament was played between all trained AlphaZero players and the MCTS player. Scalability was tested by first training networks to play Connect4 on a 6x7 board for 50 iterations, and fine-tuning their weights to play Connect4 on a 8x9 board for 30 iterations, with this player being compared to one that trained on a 8x9 board from scratch for the same number of iterations.

Results: The best AlphaZero player for Connect4 wins 73% of games against a MCTS player with 200 searches per move after training over 5,000 games of self-play for 42 hours. In the Connect4 tournament, all AlphaZero players get better scores than the MCTS player, with the residual-based architectures performing better than the convolution-based architectures. The Pathwayz tournament verifies these results on a more complex game. Furthermore, results show how parameters such as number of convolutional blocks, learning rate, and learning optimiser effect player competence. It has been shown that fine-tuning results in a better AlphaZero player than training from scratch for a given amount of time, with the former player winning 82% of games against a basic MCTS player and the latter only 73%.

Conclusions: This project shows how AlphaZero can be implemented and trained using limited resources and obtain competent artificially intelligent players for Connect4 and Pathwayz. Furthermore, the scalability results indicate that using this in conjunction with fine-tuning allows more complicated instances of games to be learnt more quickly. However, further tests are needed for some of the parameters and there is room to explore different architectures in the future.

Keywords — Machine learning, reinforcement learning, board games, scalability, AlphaZero, AlphaGo, Monte Carlo Tree Search

I INTRODUCTION

Reinforcement learning aims to solve one of the main problems of the wider field of artificial intelligence: can a machine learn something without guidance? Its usefulness comes from the fact that it closely models how humans learn; when children interact with the environment, they receive information and use that to inform any future actions. Reinforcement learning copies this approach by having an agent that performs actions in an environment, and then calculates certain short-term and long-term reward values to evaluate the action compared to other possible actions, eventually learning the best action to take in each situation. This paradigm is well-suited to learning to play a game and this is where reinforcement learning has had most of its well-known successes, in games such as backgammon [1] and even Atari video-games [2]. A typical reinforcement learning approach uses self-play where an agent plays multiple games against itself to arrive at the best possible set of moves to win the game.

One of the most recent examples of reinforcement learning success is AlphaGo which beat the world Go champion, Lee Sedol, four games to one in March 2016 [3]. This approach used a combination of reinforcement learning and supervised learning using expert moves. Later, the same team created AlphaGo Zero which removed the need for supervised learning, thereby creating a system solely based on self-play reinforcement learning [4]. Finally, AlphaZero was created which generalised this approach to create superhuman players for the games of Chess, Shogi and Go [5], all trained from initial states with only knowledge of the game rules. It is this last approach that is so revolutionary as it is one step closer to the long-standing ambition to create a program that can learn for itself.

One area of contention within reinforcement learning is that currently it is hard to reproduce results to verify if improvements have actually been made [6]. To make this harder, the original AlphaZero approach made use of extensive computing resources, using 5000 first generation TPUs (a TPU is equivalent to a high-end GPU) to generate training games, 64 second generation TPUs to train the networks, and a combination of 4 TPUs and 44 CPUs while playing. This adds doubt as to whether it is an algorithmic improvement or just a hardware improvement. For this reason, this project focuses on whether it is possible to re-produce the results of AlphaZero and whether it is feasible using much fewer resources; in this case, one CPU and one GPU while training and playing. To extend on this, traditional fine-tuning will be explored; fine-tuning of pre-trained networks has proven to be very successful in the field of computer vision due to the general feature extractions of convolutional neural networks (CNNs). CNNs form the basis of the neural networks used within AlphaZero and this project explores if this fine-tuning can be applied to AlphaZero to allow a player that has been pre-trained on one instance of a game to easily be trained to play a bigger instance of the same game. The motivation behind this is to speed up the training of these players such that AlphaZero could be used to solve larger and more complex problems quicker than training from scratch, thereby increasing its effectiveness using limited computing resources.

New uses for neural network architectures are being found all the time, and it is hard to know exactly which architecture is best for the current problem, as is highlighted by the transition from convolutional neural networks (CNNs) in AlphaGo to residual CNNs (R-CNNs) in AlphaZero. For this reason, experiments will be performed with two different neural network architectures (CNN and R-CNN) to further understand how they affect AlphaZero. Alongside this, various training parameters will be tested to understand their effect on the results.

Section III explores the related work in this field, specifically the AlphaGo, AlphGo Zero and

AlphaZero research papers that this project is based on. Section IV explains the implementation and training in detail and Section V and VI present and analyse the results of the various tests, including tournaments between all trained AlphaZero players for both Connect4 and Pathwayz and comparisons of different training parameters. Section VII concludes the project and points to further work.

The results show that AlphaZero can be implemented effectively using limited hardware to beat a basic tree search algorithm (Monte Carlo Tree Search, MCTS) at Connect4 on a 6x7 board, with the former winning 73% of games when given 200 searches per move. Furthermore, this approach can be implemented for a more complicated game, Pathwayz, with relative ease. Results with fine-tuning were successful, with a fine-tuned Connect4 player for an 8x9 board beating a player trained from scratch in a tournament. This indicates that AlphaZero can be used to play simple games and then re-use that learning to speed up training on more complicated instances of the same game. Further work with more sophisticated transfer methods could explore the ability of the network to transfer learning between Connect4 and Pathwayz as they both involve similar strategies.

II DEFINITIONS

The definition of reinforcement learning used in this paper is taken from Sutton (1998) [7]. A brief summary is given here: in a given reinforcement learning task, there exists an environment and an agent that interacts with the environment at discrete time steps. There is a set of states and a set of actions. From any given state, the agent is able to perform any one action from a subset of actions. The agent follows a policy, which is a mapping from states to probabilities of selecting actions. Every time-step, the agent moves to a new state and is given a reward. The main principle of reinforcement learning is for the agent to learn how to change its policy to maximize the reward [8].

III RELATED WORK

Reinforcement learning can be split into two main components: policy evaluation and policy improvement [7]. Policy evaluation concerns estimating the value function for a given policy, i.e estimating the reward that a player will receive if they follow the policy. Policy improvement concerns improving the policy with respect to the current value function, such that as the value function becomes optimal, the policy function will also become optimal. The three main approaches to solving reinforcement learning problems are: dynamic programming; Monte Carlo; temporal difference learning (TDL) [7]. Most research focuses on the latter two methods and so only these will be discussed.

A Monte Carlo approach estimates and learns the value function based on simulations of full games. An early example of its success was being used to solve a pole-balancing problem [9] building on even earlier work with a manual box-based method called MENACE [10].

TDL was first introduced in a Checkers-playing program [11] and aims to improve a value function by looking n steps in the future, as opposed to simulating an entire game. Q-learning, a specific TDL approach, was introduced Watkins (1989) [12] and can still be seen in current work. Instead of optimising a state value function, this approach optimises a state-action value function (also known as a Q-value function). This makes the process of policy improvement simpler as the policy can simply be greedy with respect to the Q-value function. One of the most famous TDL successes is a backgammon player called TD-Gammon [1]. This approach is one of the earliest examples of using a neural network to approximate a value function, with temporal difference learning being used to train the parameters for the network. This improved vastly on previous work and paved the way for neural-network and reinforcement learning combinations in the future, which eventually led to deep-learning-based reinforcement learning algorithms as hardware allowed neural networks to be larger.

Reinforcement learning is moving more towards integrating with deep learning to achieve the best results, and it is this approach that led to the design of the original AlphaGo system [3] and the subsequent AlphaGo Zero system [4], finally resulting in AlphaZero [5], which is the approach to be used in this research project. To understand how AlphaZero works, it is worth comparing it to not only the previous two iterations, but also other recent research papers that take a different approach to combining reinforcement learning and deep learning to solve similar problems.

A Main Method - Reinforcement Learning

Arguably the most important aspect of AlphaZero and other systems is the main method used for learning the policy that the player should take. Background research suggests a tendency in recent years towards temporal difference learning, specifically Q-learning which eventually resulted in the creation of Deep Q Networks (DQN) [13], [2], [14]. DQNs build on from the success of TD-Gammon [1], and use Q-learning to train a neural network to approximate the value function. This approach was able to achieve human-comparable results on 49 2D Atari videogames, using the same algorithm, network architecture and hyper-parameters for each player [14].

More recently, the Deep Recurrent Q Network (DRQN) has been proposed, and this approach has been shown to perform better than traditional DQN when playing a 3D game called DOOM [15]. This approach focused mainly on utilising a divide-and-conquer method with separate neural networks for each game phase and also on augmenting the DRQN with high-level game information, however it is the basic DRQN algorithm which is of interest here. This approach adds a Long-Short-Term Memory (LSTM; a neural network component that remembers values) architecture on top of an existing DQN, with the aim of mitigating the fact that in a partially observable environment (such as a 3D game world), an agent only receives an observation of the environment and does not know the full state. This approach is one the first instances of reinforcement learning being applied to partially observable environments; however in terms of being used to play a board game it is not as useful as many board games are fully observable.

Another recent research project also exploited the use of a DQN, but this time for the game "Super Smash Bros. Melee" [16]. Furthermore, this approach also made use of a second temporal difference learning method, called actor-critic. Unlike Q-learning, which solely focuses on learning an optimal value function, actor-critic methods attempt to learn both a direct policy function and a value function at the same time [7]. This is to counteract the issues with value-only-based methods and policy-only-based methods. AlphaZero, with its use of a policy network and a value network, in some way resembles this approach of using an actor-critic temporal difference learning method, which might sound counter-intuitive due to the fact that AlphaZero and its predecessors predominantly make use of a modified version of Monte Carlo Tree Search (MCTS), which is originally based on Monte Carlo reinforcement learning methods. MCTS, [17], is a decision-time planning algorithm, that searches a game tree and averages results from self-play simulations (rollouts) to determine the best moves to make from a given position in a game. A more detailed explanation is given in section VI.E. The modified version, known as Asynchronous Policy and Value MCTS (APV-MCTS), executes multiple searches in parallel and asynchronously backs up results [3]. This links to prior work which shows how asynchronous workers perform better when used with reinforcement learning than traditional approaches [13].

MCTS differs from traditional Monte Carlo reinforcement learning approaches because it is not attempting to learn a complete value function, instead it is estimating probabilities and action values for the current state according to a given rollout policy (often random) [7]. In this way, MCTS does not learn a value function for a game, it simply learns at decision-time an estimate for the best actions to take. Rollout algorithms are not actually often seen as reinforcement learning algorithms because they do not maintain long-term memories of values or policies [7].

AlphaZero extends this approach with a neural network that predicts values and probabilities of moves, and in this way AlphaZero does act like a traditional reinforcement learning algorithm as it is attempting to find a network that can learn the optimal value and optimal policy, much like the actor-critic method used within [16]. AlphaGo still uses rollouts, however AlphaZero replaces these with a value-estimate from a neural network. This means that within MCTS values are updated by comparison to the estimate of the value of the next state reached, and so this approach acts more like TDL than a Monte Carlo method; however, these methods are just two ends of the same spectrum, [18], [7].

Other papers have achieved promising results with the use of MCTS and neural networks, [13]; the difference is that AlphaZero and its predecessors use MCTS for both policy evaluation and policy improvement by predicting both probabilities and values.

AlphaGo trained with labelled data from expert moves, however AlphaZero was trained without the need for any supervised learning from expert data and it is this novel approach that opened the door to many avenues of research.

The approach of many of these papers is similar in that they combine the use of a traditional reinforcement learning algorithm, such as Q-learning or actor-critic, with neural networks to enhance results. Many of the approaches use DQNs or some variant and in many ways these methods, along with AlphaZero, are built on the foundation of TD-Gammon [1]. It is also clear that AlphaZero and its predecessors built on the progress of DQNs by extending the ideas to work with MCTS which has already been shown to work well with board games such as Othello [19]. MCTS approaches are several magnitudes of order slower than DQN agents, however they actually perform better in certain tasks, and it makes sense to use this approach in AlphaZero as board games allow more thinking time [13].

B Main method - Deep learning

All versions of AlphaZero (AlphaGo, AlphaGo Zero, and several unpublished versions as referenced in [4]) use similar concepts and neural network designs, but differ slightly in their configuration and use of neural networks. It is worth exploring how this has evolved throughout other research and how it compares to the networks and designs used by similar papers to understand why this is a state-of-the-art approach.

Architecture: In the original paper on AlphaGo, [3], MCTS was augmented with three neural networks to determine the following: a tree policy for searching the game tree; a rollout policy for choosing actions in the simulations; a value function for estimating the expected outcome of a game. In contrast, AlphaZero, [5], make use of only one neural network that determines both the value and policy at a given state. No rollout policy is needed as rollouts are not performed in AlphaZero.

From a high-level view, having to only train one network simplifies the process. The networks used are very similar in architecture, both making use of the recent successes of CNNs for playing games, [13], [20]. AlphaZero uses deep neural networks and additionally makes use of residual blocks to simplify the learning of the policy and value functions [21]. Furthermore, combining two neural networks into one neural network with two separate heads has been shown to be better than separate policy and value networks in learning Go, [4], and was also shown in other work to improve the long-term learning of a LSTM for playing a 3D first-person shooter game [15]. The approaches of AlphaZero and this other work are very similar, however the latter makes use of two neural networks (both with the split-head architecture): one is used for navigation and one is used for action/fighting. The reason for this is to modularise the networks and fine-tune each for a specific purpose, allowing the networks to be trained with fewer resources. This approach is important in the realm of video-games but for simple board games, only one network is needed.

Approach: The learning method for AlphaGo differs from AlphaZero as the latter incorporates MCTS into the learning phase. AlphaGo uses a training pipeline consisting of supervised learning to initialise the policy network, policy gradient improvement via self-play to improve the policy network and generate training data, and finally a second round of supervised learning to train the value network using the data generated in the previous step. AlphaZero, on the other hand, combines self-play learning with MCTS to fine-tune the policy and value network as follows: the result of the policy network is used as the tree policy for a MCTS search whose final result is an improved policy. This policy is then projected back onto the function space of the policy network. Similarly, the result of each game is projected back onto the value network. This approach can be traced back to Guo et al. (2014) [13], which also projected the output of a MCTS search onto a neural network, however that approach simply used a fixed MCTS policy so was simpler than the approach used for AlphaZero [5]. Both AlphaGo and AlphaZero make use of self-play to generate data to train a network using supervised learning, but AlphaZero simply extended this idea to remove the need for expert data, thereby shifting the focus from supervised learning to reinforcement learning. The use of MCTS during learning for this reason is more aligned to the fundamental idea of reinforcement learning and is why it is such an important new approach. Furthermore, AlphaZero performed better than the original system when playing Go [4], and the generalisability of the approach to other games of the same type was proven by its use in Chess and Shogi [5].

Lample (2017), [15], makes use of a combination of a DQN and a DRQN to train the neural networks, both of which build on the original Q-learning algorithm [12]. This approach is more aligned to AlphaZero than AlphaGo as it is more pure reinforcement learning whereas AlphaGo uses supervised learning.

C Scalability

In much research, scalability is focused on designing neural networks and algorithms that can scale up to problems of different sizes without changing the network architecture. The most relevant research in this area concerns evolutionary methods using neural Turing machines [22]. This approach deals with encoding memory accesses in such a way that the networks can train and be used on inputs of different sizes, with different output lengths. However, scalability in this project can be more closely associated with fine-tuning. Fine-tuning is commonly used within computer vision with the aim to re-use existing classifiers on new problems. An example success is using a pre-trained CNN-F (Fast Convolutional Neural Network) to recognise different items of food [4]. This is done by replacing the final layer with one that has the correct number of outputs and succeeds due to the fact that the target classification problem is similar to the original classification problem. Fine-tuning relies on the fact that early layers in a CNN encode general features, and later layers encode more specific features so only these higher level features need re-learning. However, this leads to one of the main problems with fine-tuning which is that if the new and original tasks are quite dissimilar, fine-tuning on the new task will cause the network to forget how to solve the original task [23]. Progressive networks aim to solve this and can be used to train a neural network on multiple different tasks in sequence without the risk of the network forgetting old task-specific features [24]. The main aim of progressive networks is to learn different tasks, and as such could be used to transfer learning between games.

Scalability specifically related to reinforcement learning in board games is hard to come by, but a multi-dimensional LSTM (MDLSTM) has been introduced [25], which is shown to be more invariant than CNNs to warping and scaling to large image sizes. In this way, MDLSTMs may be suited to the task of scaling up board sizes. However, for a project of this complexity traditional fine-tuning approaches should be sufficient as the original and target tasks are so similar [23]. Furthermore, due to time constraints, the other two more methods were unable to be explored.

IV SOLUTION

A Tools and Libraries

Development: All code was written using the JetBrains PyCharm IDE, running with Python 3.6. **Python Packages:** Pytorch was used to code the neural networks. Specifically, version 0.4.1 was installed for windows 10 and integrated with CUDA 9.0 to enable training on GPUs. The only other modules installed were numpy 1.15.2 for efficient array operations, matplotlib 3.0.0 for plotting results, and tqdm 4.28.1 for displaying progress bars during training and playing. **Training:** All training and testing was performed on a laptop running on an Intel Core i7-8550U CPU with an NVIDIA GeForce MX150 graphics card.

B Game - Connect4

The main game chosen for this project is Connect4; a two-player connection game. A typical board consists of six rows and seven columns. Each player takes turns dropping one of their coloured pieces into a column. A player wins when they have four of their pieces connected in a line, either horizontally, vertically or diagonally. This game was chosen for its simplicity; there are only seven possible moves on a standard board and so more time could be spent exploring the implementation of AlphaZero. The implementation was modularised such that the specific game logic could be changed without significantly changing how other parts of the system interact with it. A player interface was implemented that has been extended to create four different players: human, random, MCTS, AlphaZero. A Game class was also implemented; this takes a board and two players and guides one or more games between these players by asking each player for a move in turn until the game has ended.

C Game - Pathwayz

After most of the testing was completed for Connect4, the project shifted focus to Pathwayz. This was done for two main reasons: it allows the implementation to be verified further and shows the power of AlphaZero by showing that the approach can be generalised to different, and more complicated, games. The rules are simple: both players place pieces on the board, with the aim of making a fully connected path from one side of the board to the other. Each player can place a regular piece of their colour or a permanent piece of the opponent's colour; this piece also flips the colours of all the regular pieces around it so adds a complex layer of strategy. The original board size is 8x12, but a board of 4x6 was used for this implementation; even with such a small board, the first player has 48 possible moves as opposed to seven in Connect4.

D Overview of AlphaZero

AlphaZero consists of two main components: a MCTS algorithm and a neural network, both of which will be explained in more detail in later sections. The basic idea is that the MCTS uses the neural network to predict probabilities (for taking certain actions from certain states) and values (estimates of how likely a player is to win from a certain state). This removes the need for full rollout simulations that are present in the original MCTS implementation [17]. During the training phase, two player agents using MCTS with the current neural network parameters play many games against each other, using the results from the output probabilities of the MCTS and the results of the games to train the neural network with traditional backpropagation. During playing, the AlphaZero player uses MCTS with the best neural network parameters to guide the search.

E Monte Carlo Tree Search (MCTS)

MCTS is a method of using simulated self-play to learn the best actions to take for any given state [17]. The idea is to have a tree representing the state-space of the problem, including all states and actions possible, where a state describes the board configuration at a given time. Within the search tree, this is shown via the use of nodes and edges. Nodes store a specific state along with extra functions and properties, and edges are drawn between nodes, representing state-action pairs which have associated values relating to how likely a player is to win the game if they take the specific action from the given state. MCTS can be broken up into four main stages:

- **Select:** the search tree is followed from the root node, and actions are chosen using the upper confidence bound (UCB1) formula. The formula for a specific state-action pair is as follows: $\frac{w}{v} + c\sqrt{\frac{\ln vp}{v}}$, where w is the number of searches that took this action from this state and resulted in a win, v is the number of searches that took this action from this state, and vp is the number of searches that entered this state. The first component represents the state-action value and is the exploitation component which is larger for nodes with greater values; the second component is the exploration component which favours nodes which have not been visited as often; C is simply a weighting that determines how much to explore or exploit. The action chosen is that which maximises this formula and this continues until a leaf node is reached.

- **Expand:** an action is chosen from the state in the leaf node that has not yet been explored and the state that results from taking that action is added to the search tree as a node, with a state-action pair edge added between the leaf node and this new node.

- **Simulate:** a simulation is run from the newly-added node until a terminal state is reached. The actions taken within the simulation are chosen based on a default policy (usually random).

- **Backpropagation:** the result of the game is calculated with respect to the current player in the root node, and is given as +1 for a win, -1 for a loss and 0 for a draw. This value is then propagated up through the edges from the node added in the expand step to the root node.

- **Final move:** The final action chosen from the root node is the action that has the highest associated value (wins/visits ratio).

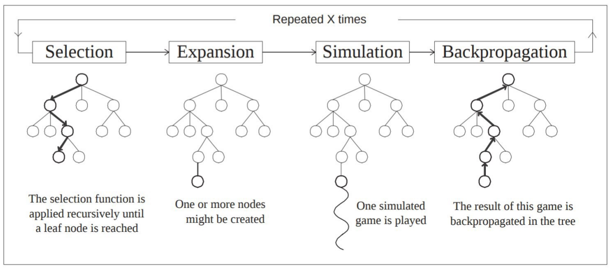

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes might be created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 1: Visualisation of MCTS, taken from [26]

AlphaZero uses a modified version of this algorithm called APV-MCTS. This significantly decreases training time by doing the following: training the network and playing self-play games simultaneously; improving the MCTS algorithm by searching the tree and sending boards to the network for evaluation asynchronously in batches. For this project, the basic MCTS algorithm without these changes was used as APV-MCTS adds a lot of complexity to the implementation and would not be fully effective without more complex hardware. The main changes are as follows:

- when expanding into a new node of the tree, the corresponding state is sent to the neural network to calculate a value for how likely the next player to move will win the game and this value is attributed to the state-action pair edge leading into the node. The neural network also returns a prior probability for each state-action pair leaving the expanded node;

- During the select phase, actions are chosen using the following formula: $\frac{w}{v} + c * p * \frac{\sqrt{vp}}{v+1}$, where v, w, vp, and c are defined as before, and p is the prior probability of taking this particular action from the current state. The value of $\frac{w}{v}$ is also not explicitly known as no simulations are performed, and so this is instead replaced by a q value which is the average value (estimated by the neural network) of states reached via exploring a path that passes through the current state-action pair;

- no rollout simulations are performed. After expanding to a new node following a certain state-action edge, the initial value of the state-action is backed-up through the tree;

- the final move chosen is based on the number of times each action is visited from the root node. A temperature variable controls how this is done; if the temperature is 0, the action that is visited the most is chosen, otherwise the action is chosen probabilistically where the probability of each action is given by $v^{\frac{1}{t}}$ where v is the number of times this action is taken from the initial state and t is the temperature parameter set by the user.

F Neueral Network

The AlphaZero implementation makes use of one convolutional block followed by 19 residual blocks (containing two convolutional layers with batch normalisation), and finally a split into two network heads - one to estimate value and one to estimate probabilities of actions. The main improvement of this network over those previously used in AlphaGo is the use of two network heads, such that probabilities and values can be computed using the same neural network architecture. This makes implementation easier and aims to improve learning due to both values and probabilities guiding the same network; this can be backed up by their own tests which confirmed that a dual-head network was indeed better. CNNs have long been known to have great success within the realm of board games and AlphaZero extends on this by making use of a relatively new technique called residual blocks which also first found great success in the field of computer vision [21]. The main aim of using residual blocks is to counteract some issues that deep networks have, specifically vanishing gradients whereby repeated multiplication during backpropagation results in infinitely small gradients, thereby making deeper networks harder to train. A residual block uses a short-cut connection to add the input to the output from the previous convolution layers; this lets the network learn the residual mapping rather than the real underlying mapping. In AlphaZero, residual blocks should perform better than only having convolutional layers [4].

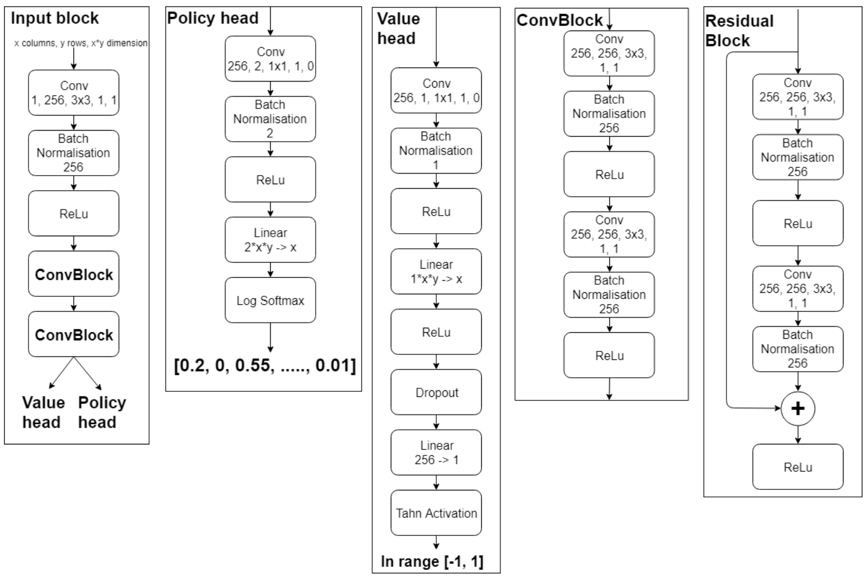


Figure 2: Structure of AZ-Res and AZ-Conv. In the residual network, the ConvBlocks are replaced with Residual blocks.

The idea of using two network heads was used in this implementation, however, the main AlphaZero network was designed for use with more complicated games such as Chess having a state-space of size 10^{47} compared to the relatively small state-space of Connect4 (10^{13}). Furthermore, a network of this size would take a very long time to train on the available hardware. For this reason, a smaller version of the AlphaZero network, AZ-Res, was trained and can be seen in figure 2. For testing purposes, this project also implemented a pure-convolution network, AZ-Conv, which is identical in every way, except with the removal of the skip-connections present in the residual network.

The networks use batch normalisation on convolutional layers and a dropout of 0.3 on fully-connected layers. This follows the original implementation [4], and was found to be the most effective through experimentation. While training, a batch size of 64 was chosen, and training was only performed for two epochs. It is unclear what learning optimiser is used for training AlphaZero, however following from the guidance of [27], and subsequent testing, the Adam optimiser was chosen [28]. The loss for each training batch was calculated as a combination of a mean-squared error loss for the values and a cross-entropy loss for the probabilities and the weights were initialised randomly, following the guidance from the original implementation.

G Training phase

Training in AlphaZero is at its core similar to supervised learning. The difference is that the labelled data is generated on-the-fly by games of self-play using the modified MCTS algorithm described in section VI.D. During one training iteration, a number of episodes are performed to generate example data. An episode consists of one game of self-play and examples are generated in the form (board state, value, probabilities) where the probabilities are the probabilities returned by the MCTS search for actions from this board state and the value is the final result of this episode's game with respect to the player who is about to move in the current board state (i.e, +1 if the current player wins from this position). During this stage, the MCTS searches 100 times before choosing a move. Various values were chosen, ranging from 25 to 800, but this was found to be the best trade-off between training time and results. MCTS acts as a policy improver as it starts with the prior probabilities determined by the neural network, performs a number of searches to improve the estimates, and uses the refined probabilities to update the network weights. During each episode, a temperature parameter is used. This is different to the temperature parameter defined in Section IV.E. In this context, the temperature is the move number at which the MCTS temperature (Section IV.E) is changed from a value of 1 to a value of 0. Essentially, before this temperature threshold is reached, the probabilities the network learns for each action are all non-zero, but after the threshold the resulting probabilities are sparse encodings with the best move having a probability of one, which should speed up training.

After all episodes have been performed, the example set generated is added to a list of all example sets. These examples are combined and shuffled. Finally, 4096 samples are chosen uniformly based on [4] and the network is trained on these examples as described in the previous section. The updated network weights are then saved and the next training iteration performed.

While training, all examples in the example sets are saved after each iteration along with the network weights so that training can be continued at a later date by simply loading the network weights and the examples generated so far.

While looking at training examples, it was clear that some of the older examples generated by the early network weights were slowing down the training and it would be best to discard some of those examples as training progressed. [27] had a similar idea and their approach was followed: a sliding window of example sets is used so as to discard earlier example sets. The sliding window increases in size over time to preserve more example sets as the network learns more.

Dirichlet noise was added to the prior probabilities of state-action pairs from the root node of the MCTS search tree during training. Noise was created using a Dirac distribution, with an input parameter to control its distribution (this parameter will be called Dirichlet from this point). The amount of noise added to the root node probabilities is determined by an epsilon parameter, with a high epsilon value meaning little noise is added and a low epsilon value meaning more noise is added. An epsilon value of 0.25 was chosen for all tests [27]. It is not clear how best to choose the Dirichlet parameter but for Connect4 a value of 1.0 was used. Pathwayz used the lower value of 0.3 as it is closer in complexity to Go, which uses this value [5].

H Scalability

An interesting issue within the realm of reinforcement learning, specifically relating to game playing, is whether fine-tuning of networks can be used to scale up to bigger instances of the problem. For the purposes of this project, scalability was only tested on Connect4 using one final player, and the scaling was from Connect4 on a 6x7 board to Connect4 on an 8x9 board.

In the AlphaZero network, two layers within the output heads use the number of available actions that a player may take as a parameter when being defined; this causes issues when scaling up to a bigger board size as these two layers will be unable to deal with the new board size input. Furthermore, the ouput of the policy head should match the number of available actions and this differs depending on the board size, so the network has to be changed. The approach taken is as follows: the pre-trained network is loaded (trained on a 6x7 board), the two affected layers are replaced with new layers that have the correct number of outputs and then the network is re-trained.

I Implementation issues

Replicating the results of AlphaZero took considerable effort as many of the design choices are not fully explained in the original papers. One main area of confusion was the optimiser used while training the network. The original paper suggests the use of the stochastic gradient descent (SGD) optimiser, with learning rate annealing starting from a learning rate of 0.2 and ending up at 0.0002. However, when this configuration was used to train the AZ-Conv architecture described earlier for 20 iterations, the training loss did not decrease. This seems to be a problem with such a high initial learning rate and so SGD was tried again but with a static learning rate of 0.01. Using this network, the player could indeed learn. For comparison, the same network was trained with the same parameters using the ADAM optimiser and this performed better against MCTS than the previous player, resulting in the choice to make ADAM the default. This is just one of the issues that were encountered when trying to determine the best parameters to replicate the results of AlphaZero and this highlights the difficulty of the task.

J Testing, Verification and Validation

The three main working parts of this system are the game, the MCTS and the neural network. If any of these three components is not performing correctly it could skew results. For this reason, all three parts were tested and verified.

To verify the game implementation, simple unit-tests were created and run every time a change was made.

To verify the MCTS implementation, a different implementation was taken (from [29]) and used within this AlphaZero implementation for a few games. The resulting moves chosen by this MCTS implementation were compared to those chosen by the implementation used in this project to see how they both perform. Most of the moves chosen were the same from a given board state, and the resulting players performed similarlv.

Verifying the neural network is difficult, mainly for the fact that all of the data is generated by using the neural network in some way so it is hard to distinguish between a data error and a network error. Much early testing shifted to a simplified version of Connect4 played on a board with only five rows and six columns, with the aim to connect three pieces instead of four. Graphs were generated showing the value mean-squared error loss, probability cross-entropy loss and the combined loss to verify that the network was training.

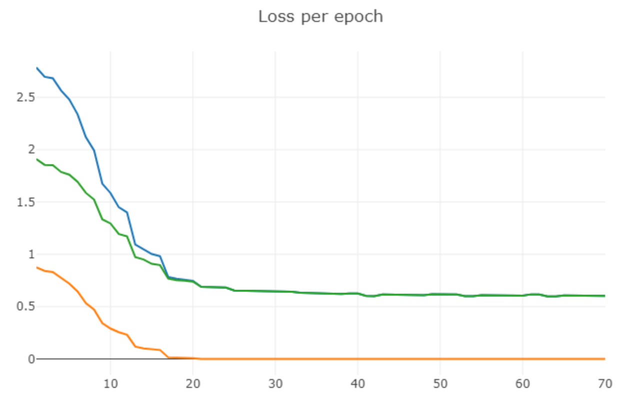


Figure 3: Early loss graph for AlphaZero using a CNN playing Connect3 on a 5x6 board (value loss in orange, probability loss in green, combined loss in blue)

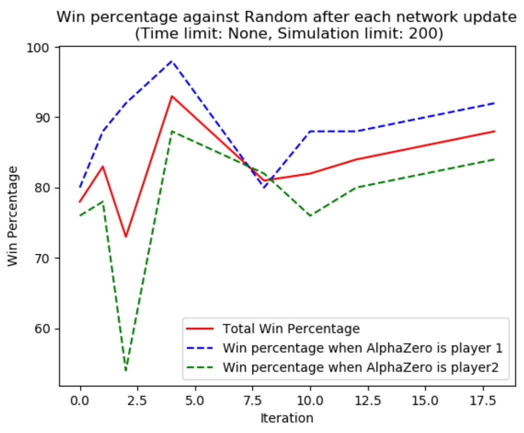


Figure 4: Early results of AlphaZero implementation vs Random opponent

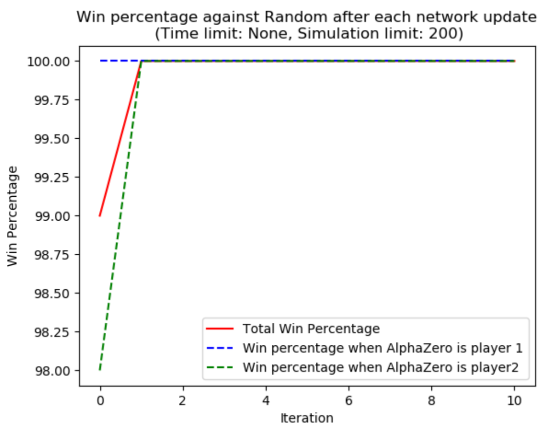


Figure 5: Later results of AlphaZero implementation vs Random opponent

From figure 3 it is clear that the network is learning something, but it is not clear whether this translates into a worthy adversary. Every newly trained network was validated by being used by an AlphaZero player that played one hundred games against another opponent; the results were then plotted on a graph. In early testing, the AlphaZero player could not even convincingly learn to beat a random opponent, as shown by figure 4, however after fixing various coding issues, the AlphaZero player agent could almost always beat a random opponent, as shown by figure 5.

V RESULTS

A *Parameters and architecture*

This project concerned many different aspects, one of which was to examine the effects of different configurations and parameters when training. The default parameters are as follows:

- Network: AZ-Conv with two Conv-blocks, Optimser: ADAM, Learning rate: 0.01, Batch Size: 64, Dropout: 0.3.
- Number of training examples: 4096, Number of episodes per iteration: 100, Number of searches per move in training: 100, C (exploration): 1.0, Dirichlet noise: 1.0, Epsilon: 0.25, Temperature: 18, Example window start size: 4, Example window maximum size: 35, Example window increase rate: every 2 iterations.

Each table shows the win percentage of an AZ-Conv player trained for five iterations, playing 100 games against a MCTS player with 200 searches per move with varying parameters. The player orders were swapped after each game to make the tests fair. Only the parameter in the table is changed between tests.

# Blocks	Win Percentage
1	36
2	40
3	36
4	45

Table 1: Varying number of blocks

C	Win Percentage
1	40
3	35

Table 2: Varying C

Optimiser	Win Percentage
SGD	40
ADAM	45

Table 3: Varying optimiser

Batch size	Win Percentage
32	38
64	37
128	50

Table 4: Varying batch size

Dropout	Win Percentage
0.1	48
0.3	37
0.5	46

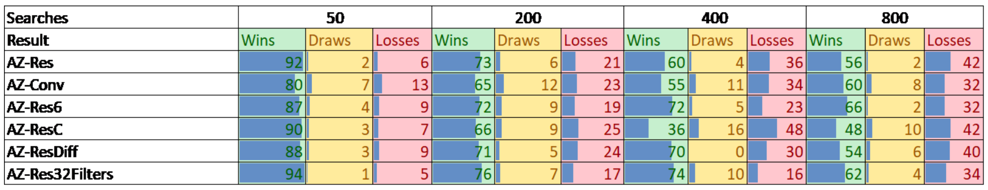
Table 5: Varying dropout

Table 1 shows a trend towards larger networks being able to play at a higher level, intuitively due to their increased learning capacity. The results from table 2 seem to suggest that increasing C decreases performance, however due to the guidance from Prasad (2018) further testing was needed [27]. Table 4 suggests an improvement with a larger batch size of 128. Table 5 is inconclusive as to a specific trend but seems to be best with a dropout of 0.1 or 0.5. Both network heads contain convolutional layers with 2 filters, however using 32 filters in the head resulted in an improvement from 40 percent to 57 percent wins against MCTS after five iterations of training. A further test was performed where the number of training examples per iteration was doubled, however this made no improvement to results.

B Connect4

Abbreviations for all the trained players are as follows: **AZ-Conv:** AlphaZero with convolutional architecture, using two convolutional blocks and default parameters; **AZ-Res:** AlphaZero with residual convolutional architecture, using two convolutional blocks and default parameters; **AZ-Res6:** AlphaZero with residual convolutional architecture, using six residual blocks and default parameters; **AZ-ResDiff:** AZ-Res with the following changes: dropout of 0.5, batch size of 128; **AZ-ResC:** AZ-Res with the following change: CPUCT value of three instead of one, **AZ-Res32:** AZ-Res with 32 filters in the convolution layers in the value and policy heads.

All players were trained for 50 iterations, which involved 5,000 games of self-play and completed in approximately 40-50 hours. All players were tested by playing 100 games against a basic MCTS player using the same number of searches per move. Various numbers of searches were used and the results are shown below:



Searches	50			200			400			800		
Result	Wins	Draws	Losses	Wins	Draws	Losses	Wins	Draws	Losses	Wins	Draws	Losses
AZ-Res	92	2	6	73	6	21	60	4	36	56	2	42
AZ-Conv	80	7	13	65	12	23	55	11	34	60	8	32
AZ-Res6	87	4	9	72	9	19	72	5	23	66	2	32
AZ-ResC	90	3	7	66	9	25	36	16	48	48	10	42
AZ-ResDiff	88	3	9	71	5	24	70	0	30	54	6	40
AZ-Res32Filters	94	1	5	76	7	17	74	10	16	62	4	34

Figure 6: Results against MCTS with varying search limit

This table suggests that an AlphaZero player using any of the mentioned architectures, except AZ-ResC, will always be better than a basic MCTS player, verifying that AlphaZero works on limited hardware given enough training time. There seems to be a trend that the more searches each player is allowed, the less improvement each player has over MCTS; this is most likely because Connect4 is simple and so a basic MCTS player can be near-optimal with enough simulations. The results for games involving 50 and 200 simulations favour the residual architectures, but suprisingly with 800 simulations the convolutional architecture does not end up being the worst. However, it is more useful to compare players with a more limited number of simulations as this more accurately leads to a comparison of how well the networks have trained as opposed to relying on how well MCTS can optimise the searches.

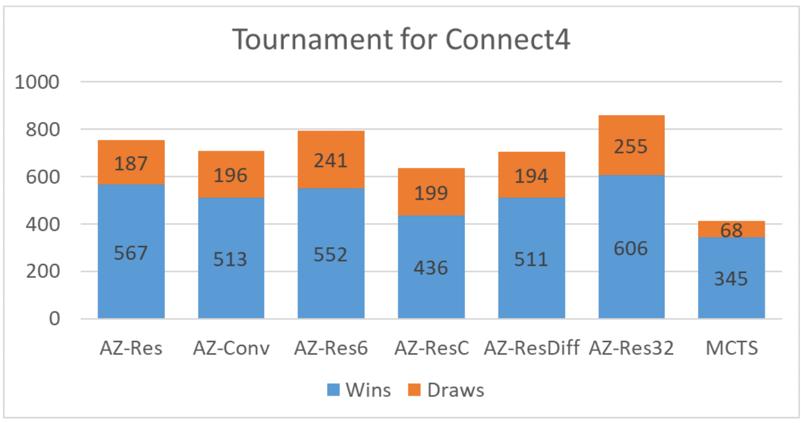


Figure 7: Results from all games played in tournament

To find the best player, all players were placed in a tournament in which each pair played 200 games, with each player having 200 searches per move. The player orders were switched between games. Due to the fact that most games between two AlphaZero players play out in the same way when repeated, it was decided to alter the games such that the first move for both players was chosen randomly. This meant the tournament found the player that could best win from a random start position, and hence the player that had learnt better generalised play. The results are shown in figure 7.

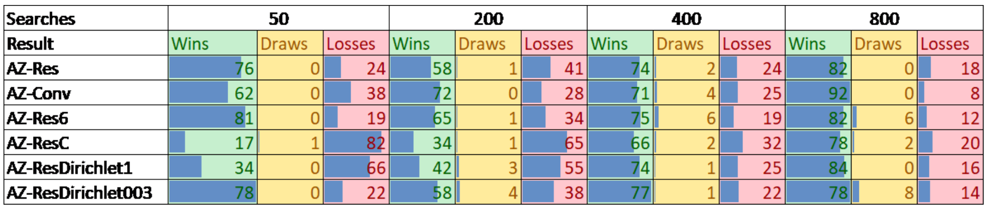
All of the trained players perform better in the tournament than the basic MCTS player as expected. In terms of comparing architectures, the performances are not massively different, however the basic residual architecture does seem to offer slightly better performance than the convolutional architecture, backing up the decision of the original authors to switch to the former

[5]. Both AZ-ResDiff and AZ-ResC perform worse than the basic AZ-Res, suggesting the default parameters are better. AZ-Res6 performed better than AZ-Res in this tournament, but still not better than AZ-Res32 which came out as the best player, intuitively down to its higher learning potential for the value and policy head outputs. It is possible that the small differences in results are down to the fact that Connnect4 is a simple game and all architectures are strong enough to learn it well, even after only 5,000 games.

C Pathwayz

The default parameters used for Pathwayz are almost identical to the defaults used for Connect4, with only the following changes: number of searches per move increases from 100 to 200; Dirichlet is 0.3. The players trained are defined in the same way as they are for Connect4, with the removal of AZ-ResDiff and AZ-Res32 (time constraints meant that it was not possible to test these on Pathwayz) and the following additions: **AZ-ResDirichlet1:** AZ-Res with Dirichlet value of 1.0; **AZ-ResDirichlet003:** AZ-Res with Dirichlet value of 0.03.

All players were trained in the same way as for Connect4, and the same tests were performed. The results are shown below:



Searches	50			200			400			800		
Result	Wins	Draws	Losses	Wins	Draws	Losses	Wins	Draws	Losses	Wins	Draws	Losses
AZ-Res	76	0	24	58	1	41	74	2	24	82	0	18
AZ-Conv	62	0	38	72	0	28	71	4	25	92	0	8
AZ-Res6	81	0	19	65	1	34	75	6	19	82	6	12
AZ-ResC	17	1	82	34	1	65	66	2	32	78	2	20
AZ-ResDirichlet1	34	0	66	42	3	55	74	1	25	84	0	16
AZ-ResDirichlet003	78	0	22	58	4	38	77	1	22	78	8	14

Figure 8: Results against MCTS with varying search limit

Apart from the anomalous cases of AZ-ResC and AZ-ResDirichlet1 with 50 and 200 searches per move, all players are able to beat a basic MCTS player with the same number of searches. The best player changes based on the number of searches per move, but surprisingly with 200 or 800 searches per move, AZ-Conv is the best player. This may seem like a contradiction against the previous results for Connect4, however the most likely cause is that the best noise and temperature parameters have not been found for Pathwayz, meaning that with only 200 searches per move while training, the players are learning in a sub-optimal way. The effect of noise can be seen clearly by the stark differences in win percentage between AZ-ResDirichlet1 and AZ-ResDirichlet003.

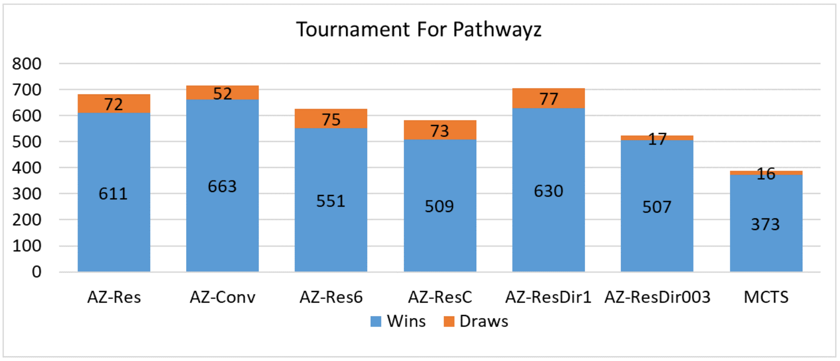


Figure 9: Results from all games played in tournament

Figure 9 reiterates previous results and clearly show the success of all AlphaZero players against MCTS in the tournament, no matter which parameters are used. Similarly to the Connect4 tournament, AZ-ResC performs poorly compared to other players, indicating that with the current default parameters, exploring rather than exploiting results in worse players; this could possibly be alleviated with more searches per move while training as the original AlphaZero implementation used 800 instead of 200 [5]. The results clearly show a difference between AZ-ResDir1 and AZ-ResDir003 however, surprisingly it is the former which comes out better here, which contradicts table 8. This can perhaps be explained by the added randomness in the tournament: AZ-ResDir1 is better at dealing with a random new situation whereas from an empty board start, AZ-ResDir003 has learnt the better optimal strategy. This result indicates more testing is needed with the noise parameters to understand which is optimal.

D Scalability

For scalability, the initial network taken was AZ-Res6 trained for 50 iterations, as shown in the Connect4 results section. This network was fine-tuned, using the method mentioned in Section IV.H, for a further 30 iterations on a 8x9 Connect4 board using a smaller learning rate of 0.001. Two versions of this were trained: in the first, all of the layers in the input block were frozen such that their weights could not be changed and only the value and policy head would be trained; in the second, no layers were frozen and all weights were learned. For comparison, an AZ-Res6 player was trained from scratch for 30 iterations on a 8x9 board using default parameters.

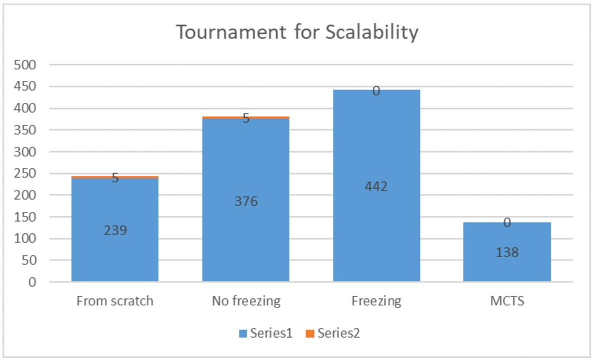


Figure 10: Results from all games played in tournament

These results again show the power of AlphaZero, specifically with the residual architecture, as the AZ-Res6 player trained from scratch is better in the tournament than the basic MCTS player. However, the main insight shown by this graph is that fine-tuning results in much greater players than simply training from scratch. It is worth noting that the pre-trained network without any fine-tuning achieves 31% wins against a basic MCTS opponent with 200 searches per move, whereas both fine-tuned networks achieve above 70% wins against the same player.

Furthermore, results suggest that only learning weights for the two output heads is better than fine-tuning all of the weights in the network, as the former player finishes the tournament with 442 wins compared to 376 for the latter.

VI EVALUATION

A Strengths

The strengths of this project lie mostly within its implementation. Many stages of forward planning and refactoring have resulted in a code-base that makes it easy to train neural networks and play games against various different players. Furthermore, the modularity facilitates fast proto-typing and testing by making it simple to add new architectures, implement different games and use different configurations of parameters. The usefullness of this can be highlighted by the fact that Pathwayz was successfully implemented and made ready for training after two days.

In terms of results, it is clear that AlphaZero has been successfully implemented for Connect4, showing a notable improvement over the basic MCTS algorithm. Promising results have also been found for the more complicated game of Pathwayz, thereby successfully verifying the generalisability of AlphaZero to different games, although more thorough testing with a wider range of parameters is needed. Additionally, results have successfully shown the effect of many different configurations of parameters and neural network architectures, going further to add knowledge to how these things affect AlphaZero learning. Furthermore, the scalability results show that traditional fine-tuning methods can extend to the realm of reinforcement learning for board-games and form one valid method for improving the performance of players trained using limited resources. Anecdotally it is also worth noting that no human player has managed to beat any of the AlphaZero Connect4 or Pathwayz players although no formal testing was done.

B Weaknesses/Limitations

The main limitation of the project was unsurprisingly resources, including computing power and time. The project aimed to explore whether AlphaZero could be feasible using limited hardware and although the results show very much that AlphaZero does work well for relatively simple games, the training required to reach that stage is not something to be ignored. All of the players trained took over 40 hours to train for 50 iterations, involving 5,000 games of self-play. For comparison, on the game of Go, AlphaZero trained for 34 hours over 21 million games. This difference is due to the following algorithmic and hardware improvements: a much greater amount of computing resources; the use of APV-MCTS. Furthermore, and playing 100 games between two players with 200 searches per move could take anywhere from thirty minutes to three hours to complete. This ultimately led to difficulties testing all of the different parameters meaning that the best set of parameters may not have been found. The solution implemented in this project is simpler than the one presented in the original AlphaZero paper, only in the sense that the original implementation was asynchronous and therefore faster. This would potentially have solved some of the timing issues, however would possibly require a more sophisticated hardware setup to fully benefit from. Furthermore, this project involved a lot of implementation and verification was hard enough without adding in the notoriously difficult asynchronous paradigm.

C Organisation of the project

The overall organisation of the project was flexible, with an overarching theme of experimenting with AlphaZero on a small scale, but with enough variation in the advanced project deliverables to allow for changes in exact direction. One major problem that became apparent was time: implementing the basic solution itself went mostly according to plan, but a myriad of implementation bugs arose and all parts of the system needed to be individually verified multiple times using time-consuming tests to make sure results could be trusted. The final verification results and players shown in the results section consist only of a small subset of tests that were run over the course of the project, with many players having to be re-trained when better parameters were found or bugs were fixed. Scalability was initially abandoned due to negative results, however after fixing various bugs, scalability was revisited later in the project cycle with more promising results. However, having to revisit old tests left little time to shift focus to more complex architectures and scalability paradigms and so this has to be left for future work.

VII CONCLUSION

The results clearly show that AlphaZero has been successfully implemented on limited hardware for the simple game of Connect4, with the best player achieving 73% wins against a basic MCTS player with 200 searches per move over 100 games with only 50 iterations of training, which consists of around 50,000 training games over 40-50 hours. Performance on the more complicated game of Pathwayz is promising, with the best player achieving 72% wins after the same number of iterations, however the performances of the players indicate more testing of certain parameters is needed. Furthermore, two different architecture types have been compared, with different configurations of each, with a residual convolutional neural network seemingly out-performing a purely convolutional neural network on Connect4. However, a contradictory result on Pathway, most likely due to sub-optimal noise parameters, highlights the need for more experimentation.

Perhaps the most striking result is that traditional fine-tuning methods prove advantageous over training players from scratch when scaling up Connect4 from a 6x7 board to an 8x9 board, highlighting one possible method to increase the effectiveness of training AlphaZero using limited hardware. Furthermore, freezing all but the final layers of the network proves better than fine-tuning all weights in the network.

It is clear that this is a project that would benefit with a lot more time, and this project acts as more of an opening to many different new areas of possible research including the following: more testing with Pathwayz to optimise parameters; exploring the use of LSTMs and MDLSTMs with comparisons to other architectures when training and fine-tuning; the use of progressive neural networks to transfer learning between Connect4 and Pathwayz.

References

[1] G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Commun. ACM*, vol. 38, pp. 58–68, Mar. 1995.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv:1312.5602 [cs]*, Dec. 2013. arXiv: 1312.5602.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. v. d. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016.

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. v. d. Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, Oct. 2017.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv:1712.01815 [cs]*, Dec. 2017. arXiv: 1712.01815.

[6] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep Reinforcement Learning that Matters," *arXiv:1709.06560 [cs, stat]*, Sept. 2017. arXiv: 1709.06560.

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.

[8] R. Bellman, "A Markovian Decision Process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[9] D. Michie and R. A. Chambers, "Boxes: An Experiment In Adaptive Control," in *Proceedings of the Second Annual Machine Intelligence Workshop*, (University of Edinburgh, UK), pp. 137–152, 1968.

[10] D. Michie, "Experiments on the Mechanization of Game-Learning Part I. Characterization of the Model and its parameters," *The Computer Journal*, vol. 6, pp. 232–236, Nov. 1963.

[11] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, vol. 3, pp. 210–229, July 1959.

[12] C. J. C. H. Watkins, *Learning from delayed rewards.* PhD Thesis, King's College, Cambridge, 1989.

[13] X. Guo, S. Singh, H. Lee, R. Lewis, and X. Wang, "Deep Learning for Real-time Atari Game Play Using Offline Monte-Carlo Tree Search Planning," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, (Cambridge, MA, USA), pp. 3338–3346, MIT Press, 2014.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[15] G. Lample and D. S. Chaplot, "Playing FPS Games with Deep Reinforcement Learning.," in *AAAI*, pp. 2140–2146, 2017.

[16] V. Firoiu, W. F. Whitney, and J. B. Tenenbaum, "Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning," *arXiv:1702.06230 [cs]*, Feb. 2017. arXiv: 1702.06230.

[17] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Computers and Games*, Lecture Notes in Computer Science, pp. 72–83, Springer, Berlin, Heidelberg, May 2006.

[18] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, Aug. 1988.

[19] M. v. d. Ree and M. Wiering, "Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play," in *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pp. 108–115, Apr. 2013.

[20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *arXiv:1602.01783 [cs]*, Feb. 2016. arXiv: 1602.01783.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385 [cs]*, Dec. 2015. arXiv: 1512.03385.

[22] J. Merrild, M. A. Rasmussen, and S. Risi, "HyperENTM: Evolving Scalable Neural Turing Machines through HyperNEAT," *arXiv:1710.04748 [cs]*, Oct. 2017. arXiv: 1710.04748.

[23] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," *arXiv:1411.1792 [cs]*, Nov. 2014. arXiv: 1411.1792.

[24] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive Neural Networks," *arXiv:1606.04671 [cs]*, June 2016. arXiv: 1606.04671.

[25] A. Graves, S. Fernandez, and J. Schmidhuber, "Multi-Dimensional Recurrent Neural Networks," *arXiv:0705.2011 [cs]*, May 2007. arXiv: 0705.2011.

[26] M. Liu, "General Game-Playing With Monte Carlo Tree Search," Oct. 2017.

[27] A. Prasad, "Lessons From Implementing AlphaZero, https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191," June 2018.

[28] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Dec. 2014. arXiv: 1412.6980.

[29] S. Nair, "A clean implementation based on AlphaZero for any game in any framework + tutorial + Othello/Gobang/TicTacToe/Connect4: suragnair/alpha-zero-general," Dec. 2018. original-date: 2017-12-01T02:55:15Z.