University of Reading

Department of Computer Science

Individual Project - CS3IP16

Creating an Artificial Intelligence for Minishogi Using Deep Neural Networks

Ivan Syrovoiskii
24000001

Supervisor: Dr. Xia Hong

29th April 2019

Abstract

todo

Acknowledgements

I would like to thank my project supervisor Dr Xia Hong for the input she provided for this project. I would also like to thank my family and friends for providing an ongoing support over the course of the year.

Contents

Glossary of Terms and Abbreviations

MCTS - Monte-Carlo Tree Search

UCB1 - Upper Confidence Bound 1

UCT - Upper Confidence Bound applied to Trees

PUCT - Polynomial Upper Confidence applied to Trees

MDP - Markov Decision Process

AI - Artificial Intelligence

RL - Reinforcement Learning

CNN - Convolutional Neural Network

SGD - Stochastic Gradient Descent

MNIST - Modified National Institute of Standards and Technology

GPU - Graphical Processor Unit

1 Introduction

The first attempts to teach a computer to play a game have been made as early as 1949, when Claude Shannon published his thoughts on how chess can be played by a computer [1]. The first computer game-playing computer program was created for Checkers by Arthur Samuel in 1956[2]. A few years later, a computer program for Go was created[3], which was improved in 1969 by Alfre Zobrist [4]. Despite significant progress, Zobrist claimed that "the program appeared to reach the bottom rung of the ladder of human GO players".

While creating an AI for Go remained an unsolved task, a lot of progress was made with less complex games. In 1989, a checkers master was defeated by a program developed by a team at the university of Alberta [5] and a chess program called Deep Thought managed to beat a chess grandmaster Bent larsen [6].

During the same year, an algorithm that used the concept of neural networks was created [7]. This algorithm was remade in 1992 [8] to use the concept of Reinforcement Learning (RL), which deals with fining optimal moves in different situations based on the rewards the algorithm receives. The same algorithm, TP-Gammon, managed to achieve a superhuman level of performance by 1994 [9]. Numerous unsuccessful attempts were later made to apply the same approach to chess an Go [10] [11]. A first truly successful attempt to create a superhuman level chess program was made in 1997, when a supercomputer Deep Blue was created [12].

Since the invention of TP-Gammon, Reinforcement Learning has been successfully applied to other games including chess [13], Scrabble [14] and poker [15]. It has also been used for mastering Atari games [16] and for complex competitive multi-agent 3D environments [17].

In 2016, a major milestone in the history of Artificial Intelligence (AI) was achieved when Deep Mind's AlphaGo became the first computer Go program to win a professional 9-dan player without any handicaps. [18] The algorithm behind it was elegantly simple: a neural network was trained on some sample games to access game positions and predict the moves that professional players tend to play more often. After that, the algorithm was left to train while playing against itself until it achieved a superhuman level [19].

In December 2017, a paper on the successor of AlphaGo was released. [20] The new algorithm, named AlphaZero, contained numerous improvements compared to the original. One of the main changes was training the neural network entirely through self-play, without any supervised training on sample games. This approach implied that the algorithm behind AlphaZero could be theoretically applied to other games with perfect information with very little tweaking.

While there have been attempts to apply AlphaZero to simple games like Connect Four and Othello [21] [22], very little research has been done covering training optimisation or benchmarking games with large action space. A lot of training was performed on powerful GPU clusters with very little optimisation.

For this project, an algorithm similar to AlphaZero was applied to Minishogi, a Japanese chess-like game. The developed algorithm has several differences compared to AlphaZero, mainly including optimisation changes that allow it to learn basic strategies a lot faster. It

also includes changes to how self-play results are used, allowing it to focus more on endgame strategies. All of these changes will be considered in more detail in the Problem Approach section.

For the game, Minishogi was selected due to the fact that no evidence was found that the algorithm has ever been applied to it before. It was also chosen because of its complexity level, which lies somewhere in between simple games like Connect Four and advanced games with many possible strategies like chess.

A game engine was created which allowed easy solution benchmarking. The produced program include the support for WinBoard, a program that provides a Graphical User Interface for chess and shogi engines.

The main aim of the project was not to create a perfect algorithm for playing Minishogi, as that would require a lot of computational power for the neural net training. The primary objective was to create a proof of concept method that relies on Deep Learning and compare its performance to multiple other baselines, including other algorithms and human players.

The training cycle for the developed algorithms consists of three main phases: self-play, neural network training and network evaluation. Because of the methods used for this algorithms, no human input is needed for training - the algorithm essentially learns the best strategies for the game by itself.

A secondary aim of this project is to produce a customisable application that can be used for creating AI using other game engines. The produced application is highly customisable, and only requires a game class with rules in order to learn a different board game.

This report covers all stages of the project development. The literature review section offers an introduction to the state of the art Artificial Intelligence algorithms. It also provides an insight to numerous machine learning techniques and how they have been applied to the problem. The problem approach section discusses the produced solution and justifies decisions made during the development process. It also provides an overview of the tools that were used for project management. The implementation sections goes over the technical detail of how exactly the program was implemented. The testing process and project result are outlined in the Verification and Validation section. A reflection on the course of development of this project is outlined in the Contribution and Reflection section. The final section of this report goes over the social and ethical aspects of creating such an AI and using Machine Learning in general.

2 Problem Articulation & Technical Specification

2.1 Problem Domain

Shogi is a Japanese two-player turn-based board game with perfect information. The core rules of shogi are very similar to chess: each player has a collection of pieces and can move them one at a time. When a piece is moved into the same square as one of the opponent's pieces, the opponents piece is considered "captured" and gets removed from the board. The game ends when a player captures their opponent's King. While the underlying idea of chess and shogi is quite similar, there are numerous aspects of shogi that add to its complexity. The key difference is related to capturing pieces. In shogi, each captured piece (apart from the King) goes into the opponents hand. During any of the opponent subsequent turns, they can then put that piece almost anywhere on the board instead of moving one of their pieces that are already in the game. This difference significantly increases the average number of possible actions for each game position, also known as the branching factor[23]: in chess it is estimated as 35 compared to 80 in shogi. It also affects the average game length, which is equal to 115 in shogi and 80 in chess [24].

Just like chess, shogi has not been solved yet, which makes it particularly interesting from the research point of view. It has been proven, however, that solving it has exponential runtime [25]. In other words, shogi is hypothetically solvable by a deterministic Turing machine in $O(2^{p(n)})$ time, where $p(n)$ is a polynomial function of the board size.

As was mentioned before, a variation of shogi called minishogi was selected for this project. Minishogi uses the same set of rules as shogi, but uses a smaller 5x5 (Figure 1) board and only six pieces for each player instead of twenty in regular shogi. The starting pieces are:

- A Pawn, which can only move one square up
- A King, which can move to any adjacent square
- A Rook, which can move to any square in the same row or column
- A Bishop, which can move to any square on the same diagonal
- A Silver General, which moves like a King, except that it cannot move directly left, right or backwards.
- A Gold General, which moves like a King, except that it cannot move to either of the two squares diagonally backward.

Figure 1: Minishogi board. Ito, 2019 [26]

All of these movements have been summarised in Figure 2.

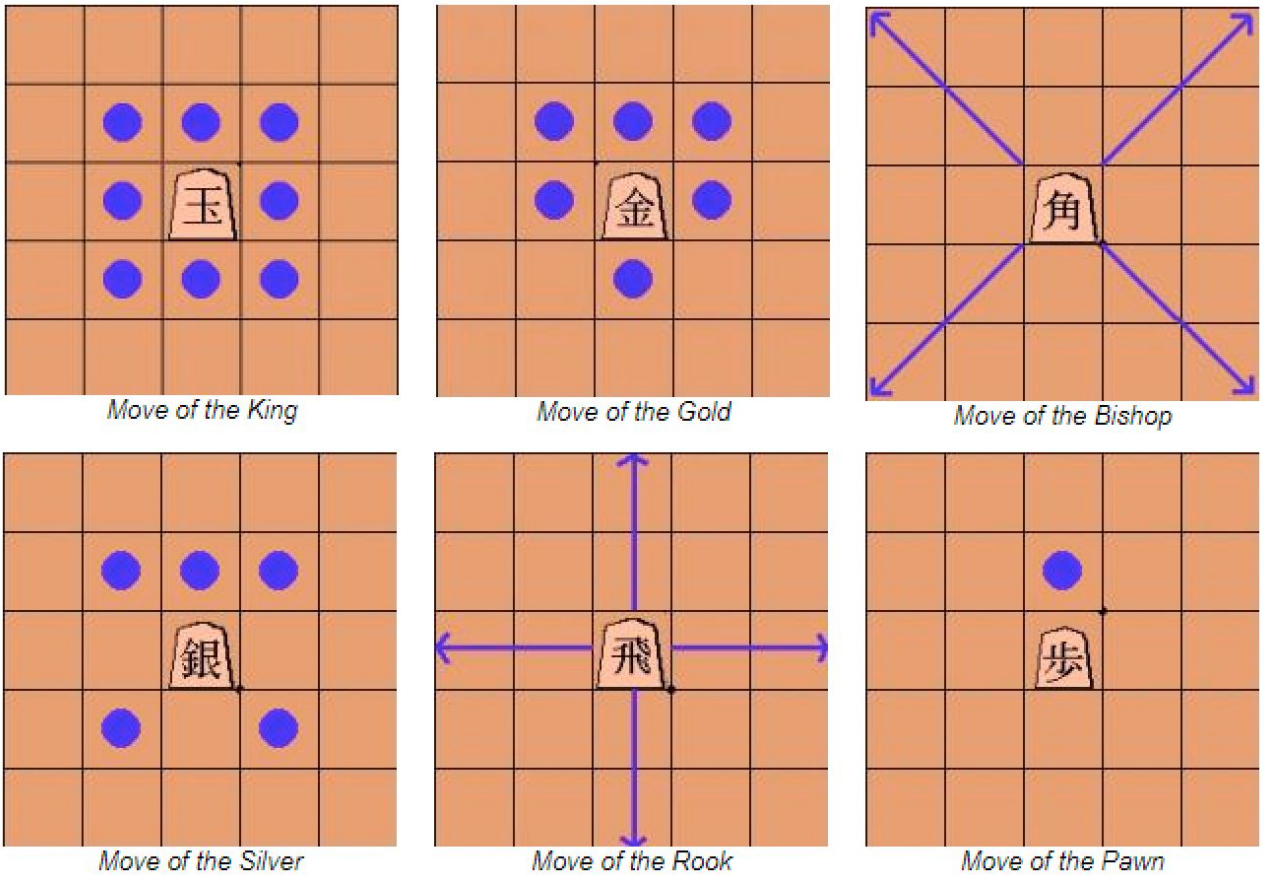

Figure 2: Minishogi Piece Moves. ancientchess.com [27]

Similarly to chess, a piece can be promoted once it reaches the promotion zone of the board, which is equal to the top row for minishogi. Pieces also can get promoted when leaving

the promotion area. When promoted, Pawns and Silver General replace their moves with Gold General moves. Rooks and Bishops also gain Gold moves, but keep their original moves as well. Gold Generals and Kings cannot be promoted. Once a piece is captured, it gets automatically demoted back to its original form. The list of moves for each promoted piece has been summarised in Figure 3.

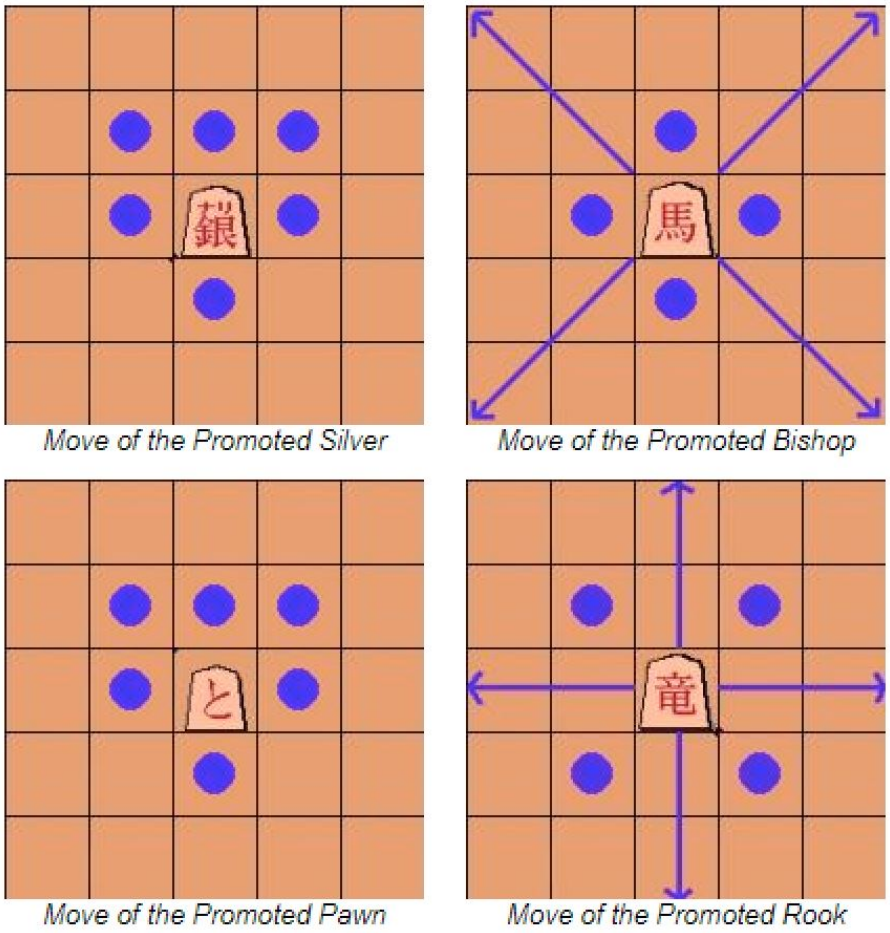

Figure 3: Minishogi Promoted Piece Moves. ancientchess.com [27]

The are also several additional rules regarding pawns:

- A pawn cannot be dropped onto a column containing another demoted pawn of the same player.
- A pawn cannot be dropped to give an immediate checkmate (but can be dropped if it leads to a check without an immediate mate).
- A pawn cannot be dropped in the top row, which would make it unable to move.

Another important aspect of minishogi rules is handling repeats. According to the rules, if the same position occurs more than three times, the game ends and the player that made the very first move in the game loses. An exception to this rule is when one player perpetually checks the opponent, in that case the checking player loses the game.

2.2 Technical Specification

The key aim of this project was to verify if using Deep Learning is a viable option for developing artificial intelligence in conditions with limited resources. The developed program needed to meet multiple objectives; some of them were defined in the Project Initiation Document (appendix A). They could be separated into two categories: functional and non-functional. The former describe what the algorithm should do and the later describe criteria that can be used to judge the operation of the system. Both groups have been summarised in the list below.

Functional

- For each game state, the program should be able list all legal moves
- The algorithm should be able to check for win/loss conditions
- The program should keep track of all moves in the current game
- It should be possible to update the state of the game after a certain move is taken (e. g. update the board, players' hands, etc)
- The program should provide an interface for testing the algorithms against different game agents, including a human player

Non-functional

- For each game, the algorithm should aim to select moves that would lead it its victory
- The algorithm should spend to much time accessing each move
- The algorithm should run on consumer-level hardware
- The game model produced by the algorithm should take a reasonable amount of time to prepare
- The program should log its progress in a convenient format

Multiple methods could be used in order to validate performance of the solution. It is assumed that the problem is adequately solved if the produced solution demonstrates a reasonably good performance against a human player. It should also demonstrate a significant superiority against other common model-free approaches, like Monte-Carlo Tree Search or choosing actions at random.

2.3 Stakeholders

Key stakeholders for this project have been outlined below.

Developer (Ivan Syrovoiskii)

The developer is responsible for conducting research, designing a solution, implementing it and running tests. It is also their responsibility to document their progress using a logbook and ensure that all deadlines are met.

Project supervisor (Xia Hong)

The project supervisor oversees the developer's work. Their role is to provide guidance to the developer and monitor their work. They are also responsible for marking the final report and providing feedback to the developer.

Shogi Players

Shogi enthusiasts might be interested in competing with the developed AI. They would require the program to be easy to use and have consumer grade system requirements.

Other developers

Other developers might be interested in applying this implementation of AlphaZero to other games. They would be particularly interested in code clarity and modularity, as that would make extending the algorithm easier.

3 Literature Review

3.1 Minimax

Over the years, many different approaches have been created for competitive multi-agent environments. Most of them use the concept of game trees, which are a directed graph where each turn is a vertex of the tree, and each branch indicates the players' successive choices [28]. "Artificial Intelligence: A Modern Approach" by Russell and Norvig provides an introduction to many AI fields including the adversarial search problems, also simply known as games. [29]. Their book defines Minimax as a common algorithm for deterministic, turn-taking, two-player, zero-sum games with perfect information. The algorithm relies on computing a "Minimax value" of each node, which can be defined as follows:

$$\text{minimax}(s) = \begin{cases} \text{Utility}(s) & \text{if Terminal-Test}(s) \\ \max_{a \in Actions(s)} \text{minimax}(\text{Result}(s, a)) & \text{if Player}(s) = \text{Max} \\ \min_{a \in Actions(s)} \text{minimax}(\text{Result}(s, a)) & \text{if Player(s) = Min} \end{cases} \tag{1}$$

In this equation, s represents the current state of the game and $Utility(s)$ is the value of the terminal node. For each state, its Minimax value is equal to the minimum value of maximum values of the following states. In other words, the value of each state is the smallest possible reward a player can get assuming that the opponent always selects the best possible move during their turn. The algorithm then calculates the Minimax value for each possible state of the game and selects the state with the highest value.

An example of applying Minimax to noughts and crosses is shown in Figure 4.

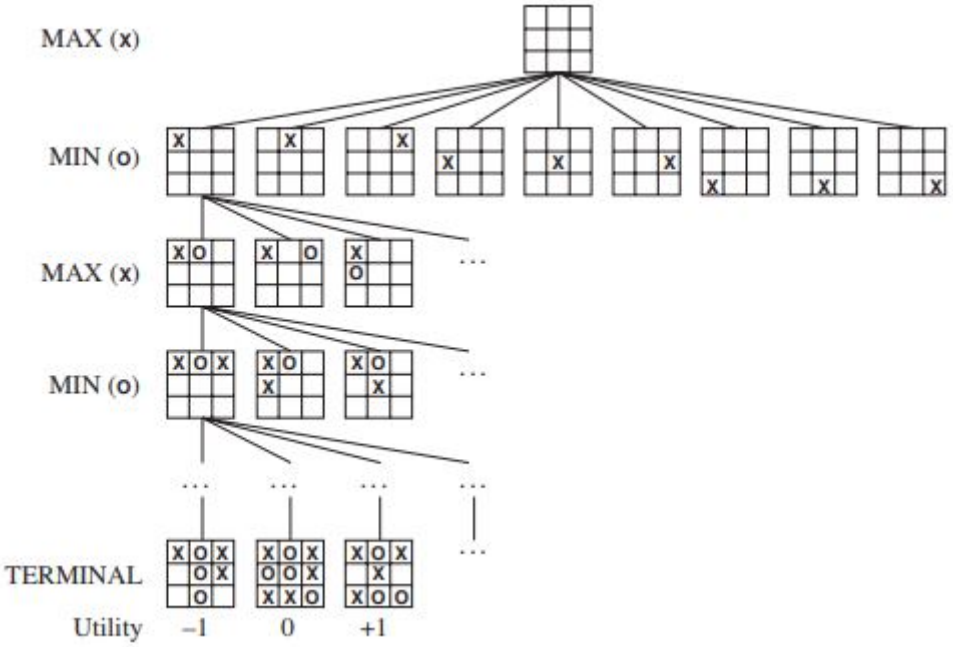

Figure 4: Applying Minimax to noughts and crosses. Russel, 2009 [29]

In their book, Russell & Norvig state that the problem with Minimax search is that the number of game states it has to examine is exponential relative to the depth of the tree - which, in case of shogi, is equal to 10^{71} [24]. To overcome the exponential time complexity, it is suggested to only consider a part of the game tree by cutting off the search after considering a specific number of nodes and then using a function to estimate the value of the leaf state instead of getting its actual value. The book states that this is usually done using a function that combines the weighted features of the state, like the number of different pieces and their positions. The weights and features can be identified either based on existing human experience or with machine learning techniques such as decision trees or artificial neural networks. It is also possible to reduce the number of considered nodes by stopping the evaluation when at least one possibility has been found that proves the move is worse than a previously examined move. This method is called Aplha-Beta search and it has been used before on Othello [30] and for creating DeepBlue [12].

3.2 Reinforcement Learning Techniques

"Reinforcement Learning" by Sutton and Barto [31] provides an introduction to the state of the art Reinforcement Learning techniques. In this class of methods, an agent uses a policy to select its next action, receives a reward for the action it takes, and then uses it in a feedback loop to update its policy function.

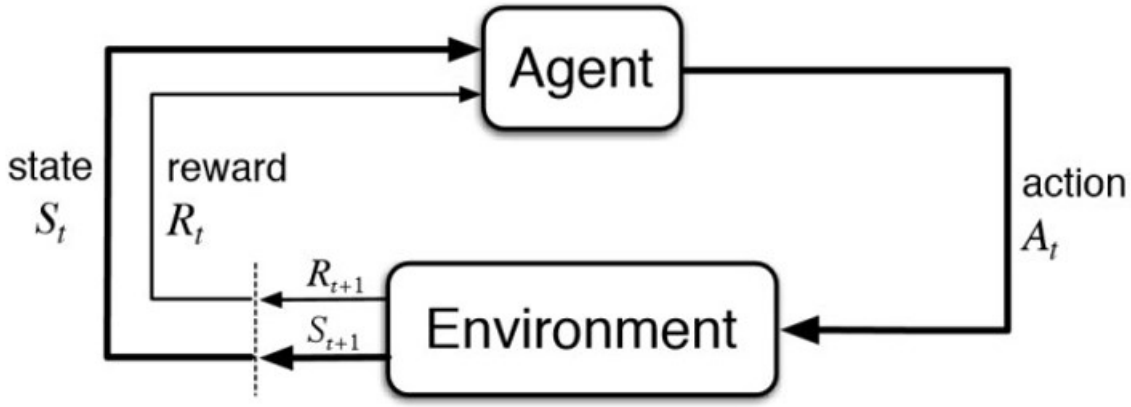

Figure 5: Reinforcement Learning loop. kdnuggets.com [32]

Compared to minimax, RL-based algorithms do not assume that the opponent has a particular way of playing. According to the book, "a minimax player would never reach a game state from which it could lose, even if in fact it always won from that state because of incorrect play by the opponent". Using reinforcement learning takes into account the fact that complex games might involve multiple different strategies, as there might not be a single perfect strategy.

The same book also defines a concept of Markov Decision Processes(MDP). A MDP is a tuple, (S, A, T, R), where S is a set of states, A is a set of actions, T is a transition function $S \times A \times S \rightarrow [0, 1]$, and R is a reward function $S \times A \rightarrow \mathbb{R}$ [33]. The transition function defines a probability distribution over next states and the reward function defines the reward received

when selecting an action from the given state. To solve a MDP, a policy function $\pi : S \to A$ that maximises the reward needs to be found.

Basic reinforcement learning problem can be modelled as a Markov decision process [34]:

- S is a set of states
- A is a set of actions of the agent
- $P_a(s, s')$ is a probability of transitioning from state s to state s'
- $R_a(s, s')$ is the reward after transitioning from state s to state s' with action a

One way to find an optimal policy is to find the optimal value function Q, which is the expected infinite discounted sum of the reward that the agent will gain if it executes the optimal policy. [35]. There are multiple approaches available that can produce such function.

One of the examples of a reinforcement learning algorithm is Q-learning, where the value function Q is used to determine the best possible move [36]. The equation for updating it is:

$$Q^{new}(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(R(s_t, a_t) + \gamma(max_a Q(s_{t+1}, a)) \tag{2}$$

where:

- $Q(s_t, a_t)$ is the previous Q function value
- s_t is the state of the game at time t
- a_t is the action that can be taken at time t
- α is the learning rate constant
- $R(s_t, a_t)$ is the reward for moving from state s_t into s_{t+1}
- γ is the discount constant
- $max_a Q(s_{t+1}, a)$ is the estimate of optimal future value.

An article by Bertsekas [37] explores the idea of combining Q-learning with another algorithm, Monte-Carlo Search, which produces better results than a pure Q-learning agent, but worse results than the pure Monte-Carlo Tree Search algorithm.

Monte-Carlo Tree Search is a best-first search technique which uses stochastic simulations [38]. It is based on the idea of making its decisions based on random sampling - which was proven to be quite effective by Kearns and others in 2002 [39]. The Monte-Carlo Tree Search algorithm consists of four stages:

- Selection: starting from the node for the current state, select successive child nodes until a leaf node (a node from which no simulation has yet been initiated) is reached.

- Expansion: unless the leaf node is a terminal node, select one of its children
- Simulation: complete one random playout from that child node until a terminal node is reached.
- Backpropagation: use the result of the playout to update the win/play ratio for each node in the path from it to the root

The algorithm has been successfully applied to many games including Shogi [40], Othello [41] and Connect Four [42]. An simplified example of a single iteration of Monte-Carlo Tree Search is shown in Figure 6.

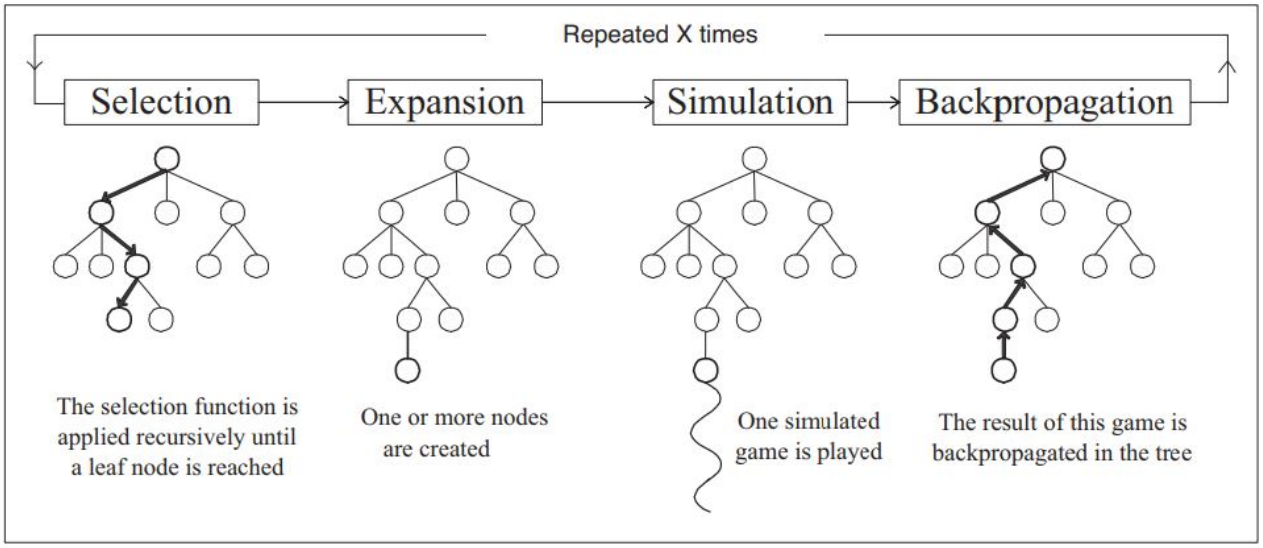

Figure 6: An iteration of MCTS. Chaslot, 2008 [38]

An important part of the algorithm is selecting the child for expansion, which can be done using an Upper Confidence Bound for Trees (UCT) [43]. This technique is a special case of a more general method, Upper Confidence Bound 1, which was introduced by Auer in 2002 for the Random Bandit Problems [44].

In their paper, Kocsis and Szepesvari propose to choose a child node for which the following expression has the highest value:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}} \tag{3}$$

where:

- w_i is the number of wins for the node
- n_i is the number of visits for the node
- N_i is the total number of simulations from the parent node, which is equal to the sum of n_i values of each child

- c is a hyperparameter which is chosen empirically

The UCT formula provides a balance between the exploitation of nodes with known rewards and unvisited nodes with unknown rewards. The rate of exploration can be controlled using the hyperparameter.

"A Survey of Monte-Carlo Tree Search Methods" [45] provides an overview of more variations and improvements for MCTS. It suggests other formulas for selecting children (UCB-Tuned, Bayesian UCT, EXP3), but claims that UCT is still considered to be the most popular choice. It also briefly discusses parallelisation, which is discussed in more detail in the proceedings article from a 2008 conference [46]. Three methods are proposed in it:

- Leaf parallelisation, where multiple random games from the same leaf node are executed in parallel
- Root parallelisation, which involves building multiple MCTS trees in parallel and then merging them
- Tree parallelisation where one MCTS tree is shared among several threads

The same article introduces a strength-speedup measure, which corresponds to the time needed for a parallel algorithm to achieve the same strength as a single-threaded version. The best strength-speedup measure of 8.5 was achieved for tree parallelisation with 16 concurrent threads.

TD-Learning is another Reinforcement Learning algorithm that assigns utility values to states alone instead of state-action pairs. [47] It uses the a formula[48] which is very similar to the formula for Q-learning:

$$V(s'_t) = V(s_t) + \alpha(R(s_t) + \gamma V(s_{t+1}) - V(s_t)) \tag{4}$$

where:

- $V(s_t)$ is the value of state s_t
- s_t is the state of the game at time t
- α is the learning rate constant
- $R(s_t)$ is the reward for moving into state s_t
- γ is the discount constant

This method has been applied to games like backgammon in 1995[8] and chess in 2000[49].

3.3 AlphaGo

In 2016, DeepMind released an article explaining the method that they used for developing AlphaGo, an algorithm that managed to defeat the human European Go champion in 2015 [50]. The article revealed that AlphaGo used and enhanced version of Monte-Carlo Tree Search: instead of performing expensive rollouts followed by backpropagation, a neural network was used to predict a potential reward for each action. It also used a second neural network to predict the probability distribution for all legal actions, which was used to direct the Monte-Carlo Search towards moves that are more likely to be played by a professional player.

Instead of using UCT, AlphaGo Zero used an algorithm called Polynomial Upper Confidence applied to Trees (PUCT) [51]:

$$a_t = (argmax(Q(s_t, a) + u(s_t, a)) \tag{5}$$

In this equation $Q(s_t, a)$ is the reward from the MCTS. It starts from zero and then gets updated using the following formula:

$$Q^{\text{new}}(s_t, a) = \frac{N(s_t, a) \cdot Q(s_t, a) + v(s_t, a)}{N(s_t, a) + 1} \tag{6}$$

The second term, $u(s_t, a)$, is a function that returns the prediction from the neural network, defined by the following equation:

$$u(s_t, a) = c \cdot \frac{P(s_t, a)}{1 + N(s_t, a)} \tag{7}$$

In this formula c is an exploration constant and $P(s_t, a)$ is a prediction returned by the neural network. Increasing c allows to put more weight towards the exploration term, while decreasing it make the algorithm prefer MCTS results over unexplored moves.

Once the simulations are complete, the action is selected based on a probability distribution over all actions, where the probability of selecting an action is proportional to the exponentiated visit count for each move:

$$\pi_a \propto N(s, a)1^{\tau} \tag{8}$$

where τ is a temperature parameter and controls the degree of exploration. AlphaGo Zero uses $\tau = 0$ for the first 30 moves and then uses $\tau = 1$ after that, picking the move with the largest visit count.

In 2017, a generalised version of AlphaGo, named AlphaGo Zero, was released. Compared to its predecessor, it was trained solely by self-play reinforcement learning, without any supervision. It also only used stone positions, while AlphaGo was trained from human data using handcrafted features. Another significant change was the use of a single neural network for both policy and value predictions. AlphaZero will be discussed in more detail later in this report.

3.4 Convolutional Neural Networks

The network used in AlphaGo Zero used a special type of neural networks: Convolutional Neural Networks, also known CNNs. CNNs are a specialized kind of neural network for processing data that has a known grid-like topology [52]. They were first created by Yann LeCunn in 1998 [53] and were initially applied to a document recognition problem.

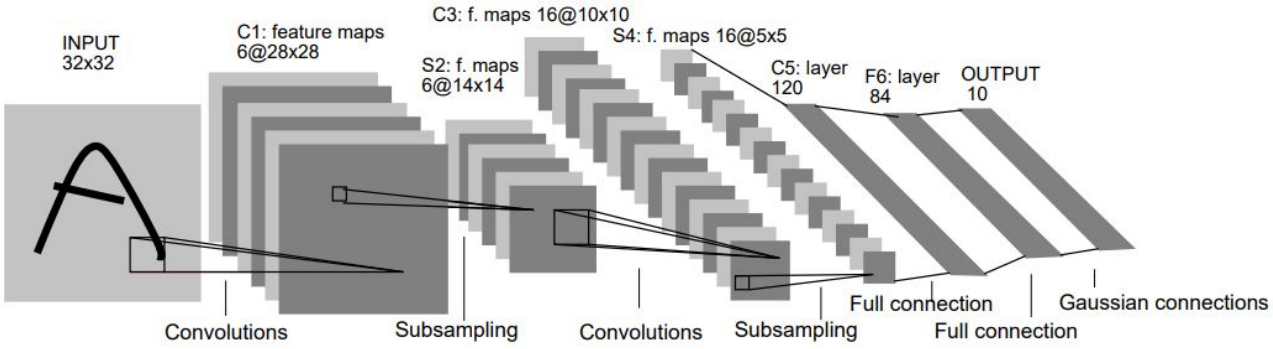

Figure 7: LeNet-5 Architecture. LeCunn, 1998 [53]

The network proposed by LeCunn (Figure 7) consisted of a two parts: convolution and polling layers for learning input features and fully connected layers for classification. Each convolutional layer consists of a combination of elements called filters (or kernels). Each filter extends through the full depth of the input volume starting from the top-left corner of the input image, and maps a $n \times n$ area of the input area it is currently processing (which is known as a receptive field) to a single output, as shown in Figure 8. As the filter is sliding around the input image, it is multiplying the values in the filter with the original values of the image [54].

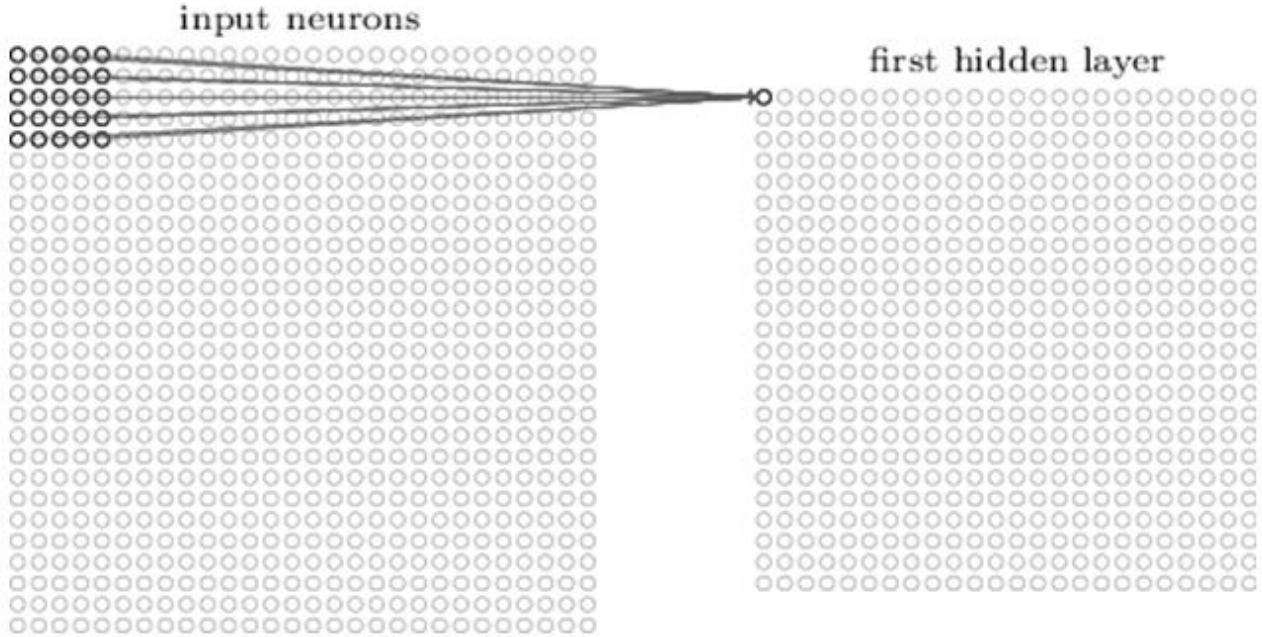

Figure 8: Visualisation of a 5x5 filter. Deshpande, 2016 [54]

The second type of operation used in CNN is pooling. Like with the convolutional layer, the main aim of pooling is to reduce the spacial size of the convolved feature [55]. A pooling layer takes a filter, applies it to the input volume and outputs the maximum (or average) number in every subregion that the filter convolves around. [54]. An example of applying pooling is shown in Figure 9.

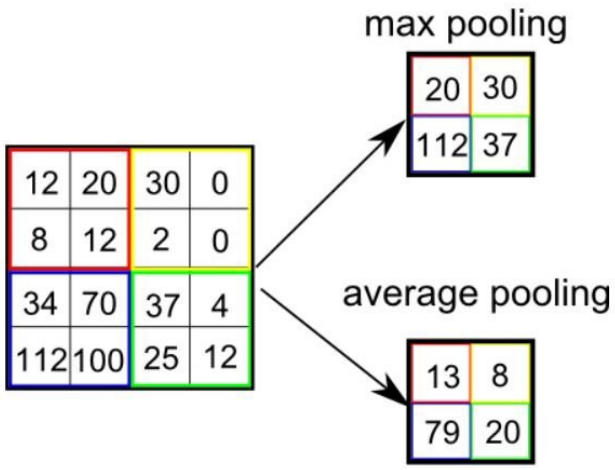

Figure 9: Visualisation of applying pooling. Saha, 2018 [55]

Convolutional Neural Networks have shown excellent result in multiple areas of image recognition. In 2012, an error rate of 0.23% on the MNIST database was achieved [56]. They have also achieved a low error rate in facial recognition [57] and 97.6% recognition rate for a 5600 image dataset [58].

A modification of traditional CNN called ResNet was developed in 2015 [59]. It introduced a concept of residual blocks, which were used to skips one or more layers, as shown in Figure 10

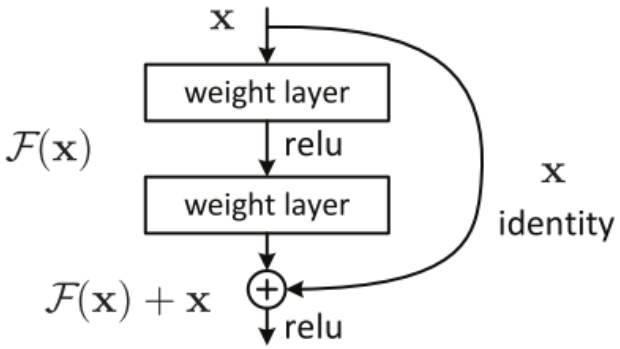

Figure 10: Residual Block. Fung, 2017 [60]

Deep networks are known to be prone to a vanishing gradient problem: as the gradient gets back propagated back through the network, the gradient becomes infinitely small due to repeated multiplication. [61]. According to the authors of ResNet, using residual blocks allows

to mitigate this issue. Testing the network on a CIFAR-10/100 dataset produced an error rate of 4.62%, which was lower than all other networks used as a baseline [62].

Another architecture based on Convolutional Neural Networks was released in 2017 [63]. Densely Connected Convolutional Networks continued the idea of skipping connections, but used significantly more complicated structure for each block, as shown in Figure 11.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Figure 11: Dense Convolutional Neural Network. Huang, 2017 [63]

Unlike ResNet, DenseNets do not sum the output of the previous block. Instead, the output gets concatenated, giving each layer access to all feature maps from preceding layers. According to the article, applying DenseNet to the CIFAR-10/100 dataset resulted in an error rate of 3.46%, which is lower than the one produced by ResNet.

3.5 Alpha Zero

The algorithm used for beating Lee Sedol was generalised and published by DeepMind in 2017 [20]. The paper introduced the result of applying a modification of AlphaGo Zero to two more games - Chess in Shogi. Similarly to its predecessor, a Residual neural network was used for simultaneous policy and value prediction. The paper also described the structure that was used to encode the game states and game actions. For all three games, the input for the neural network used one-hot encoding, creating a $N \times N \times MT + L$ image stack, where:

- N is the size of the board
- M is the number of feature planes that represent player's pieces, with one plane for each piece type
- T is the number of states passed to the network

- L is the number of extra features passed to the network, like the overall number of moves or current player's colour

The action policy for chess was represented by $8 \times 8 \times 73$ stack of planes. The 8×8 plane represented the position of the piece that can be moved. The 73 channels represented all possible types of actions, with 56 planes encoding possible Queen Moves (all moves that can be performed by a Queen), 8 planes encoding Knight Moves (8 possible moves performed by a Knight) and 9 for underpromotions (for pawn moves or captures in two possible diagonals, to knight, bishop or rook).

A policy for shogi was encoded using 139 channels, where:

- 64 planes were used for Queen Moves
- 2 for Knight Moves
- 64 for promoting Queen Moves
- 2 for promoting Knight moves
- 7 for Drops (one for each piece type)

The overall structure of AlphaZero consisted of the following steps:

- Training examples are generated through self-play, starting from a neural network with random weights
- After each game, a value (1 for winning and -1 for losing) and a matrix indicating the next selected move are assigned to each state
- The neural network is trained on the examples
- The untrained network is compared with the trained one
- If the trained network win rate exceeds a threshold, the old neural network is replaced with it

Another important addition to AlphaZero was the use of Dirichlet noise [64] for the root node policy vector. This was done to balance exploration, ensuring that the network does not over-fit and continues to explore actions even if the neural network returns a prediction close to zero.

According to the report, the loss function was defined as a combination of categorical cross entropy for the policy output and mean squared error for the value [52]:

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}_t \cdot \log(\vec{p}_\theta(s_t)) \tag{9}$$

where:

- $v_\theta(s_t)$ is the value prediction from the neural network for sate s_t
- z_t is the actual value of for this state, based on previous self play
- $\vec{\pi}_t$ is the policy vector, predicted by the network
- $\vec{p}_\theta(s_t)$ is the actual probability vector, also based on results of self-play

There are multiple optimisers that can be used. Adam [65] is a common optimiser with decent performance, however, an article from 2017 has demonstrated that in some particular cases Stochastic Gradient Descent (SGD) with Nesterov Momentum [66] demonstrates better performance [67]. A recent article from 2019 has introduced a new type of optimiser - Adabound, which is supposed to combine the speed of Adam and the quality of SGD [68]. According to the article, it showed superior performance compared to both Adam and SGD.

3.6 Multi-Agent Environment

A paper published by OpenAI [17] provides an overview of training agents in a competitive three-dimensional environment. A range of techniques are discussed in this paper, including Proximal Policy Optimisation [69]. The idea behind it is that it limits the policy gradient update for each iteration, which slows down learning but also helps to prevent the parameters from updating outside the range where the data has been collected - which can lead to instability during training. This method, however, only works with methods that rely on updating the policy gradient.

Another interesting approach that was used is ensemble of agents - during self-play, an opponent for each agent was selected from a pool of enemies, which lead to a bigger variety of strategies.

An exploration curriculum was also used speed up learning: instead of using self-play to learn complex motor behaviours like walking and jumping, a dense reward was introduced. This reward encouraged the agents to learn these behaviours, and was set to slowly diminish in favour of a sparse reward.

4 The Solution Approach

4.1 Proposed Solution

4.1.1 Training Cycle

An implementation of Monte-Carlo Tree Search reinforced with a Convolutional Neural Network was developed for this project.

The program consisted of three separate modules: the game engine, the Monte-Carlo Tree Search implementation for selecting optimal moves and a Deep Neural Network (Figure 12).

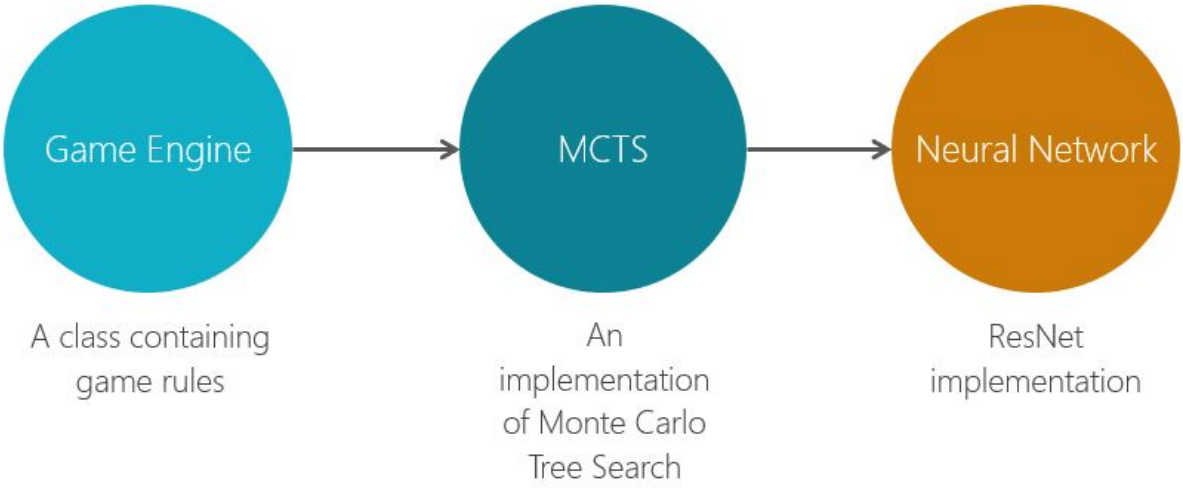

Figure 12: Program Structure

These modules were combined together to create a training cycle (Figure 13).

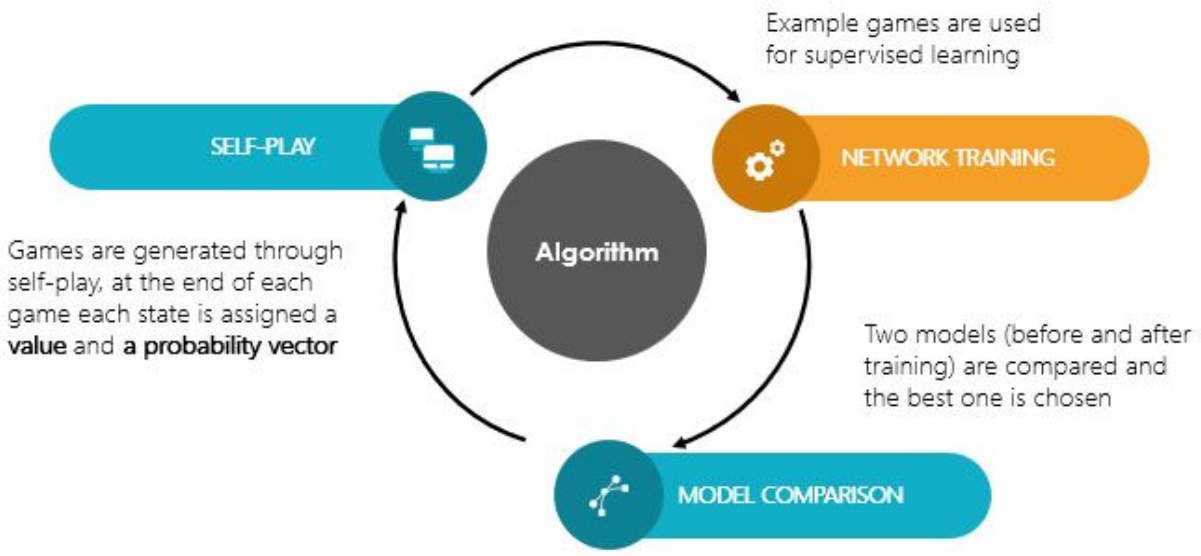

Figure 13: Training cycle

Before the start of the cycle, the neural network is randomly initialised using Xavier weight initialisation [70]. It is then used to make predictions during the first round of self-play. During self-play, Monte-Carlo Tree Search is used to run 800 simulations. For each simulation, the PUCT formula (Equation (5)) is used to determine the direction of the search.

Similarly to the original paper [20], a Dirichlet distribution [64] $Dir(\alpha)$ is also used for each root node, where α is selected as 10/(avg Branching Factor). After running some tests it was established that, for early games, the average factor for minishogi is 25, hence $\alpha = 0.4$ was used. For each axis of the probability vector, the noise is then added using the following formula:

$$P(s,a) = (1-\epsilon) \cdot P(s,a) + \epsilon Dir(\alpha) \tag{10}$$

where ϵ is a constant that was set to 0.25, meaning that a contribution of 25% for the probability vector was coming from the Dirichlet noise rather than previous in-game experience. This was done to balance the search for the root node, initially making probabilities for all moves more similar.

Once the search is complete, the next action is selected based on number of visits per each node (Equation (8)). For this approach, the temperature value of 15 was used, which means that after 15 steps the agent started selecting the next move by selecting the node with the biggest number of visits rather than using a probability distribution over the number of visits.

At the end of each training epoch all visited states are saved and assigned to values: the action frequency distribution (a 1725-dimensional vector that describes the next selected move) and a value of the sate (-1 if the player who went into that state lost and 1 otherwise). These game example are then used for training of the neural network.

Two important things are done before the network is trained. Firstly, a sliding window method is used: only examples from last 5 comparison cycles are selected. This allows to

reduce the training time and also only focus on more recent, better quality training examples.

The second part of example preparation is example deduplication. During self-play, the program tends to generate a lot more states for opening moves, which increases the overall amount of training examples. To avoid this, all identical states are merged together by summing up their policy vectors and values, dividing them by the number of similar states and then saving the result as a new training set.

Once training is complete, two networks (before and after training) play with each other for 40 rounds, an if the new network wins at least 55% of the games it then replaces the old neural network. After that the training cycle repeats. During this stage the temperature value τ for both networks is set to 0, which means that both of them choose actions deterministically rather than using a probability distribution over possible moves. This is done to minimise the role of randomness during network evaluation.

4.1.2 Neural Network

ResNet [60] was selected as the model for the neural network. Its overall structure was made to resemble a simplified version of the original ResNet model. A total of 6 residual blocks was used, each with two convolutional layers with 256 3x3 filters. Each block also includes two batch normalisation layers to decrease overfitting and ReLU activation (Figure 14) [71].

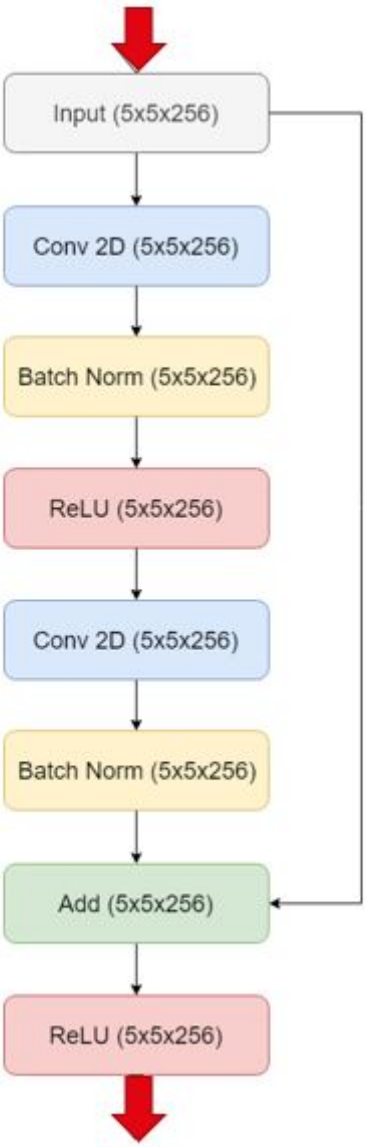

Figure 14: Residual block

L2 Regularisation (c=0.01) was also applied to each kernel to reduce overfitting.

The policy head of the neural net has a convolution layer with a 1x1 kernel and 85 filters, followed by batch normalisation, ReLU, and a densely connected layer with 1725 neurons.

The value head starts with a 1x1 convolutional filter, followed by batch normalisation, ReLU and two densely connected layers, with 256 and 1 neurons each.

Stochastic Gradient Descent with Nesterov Momentum was selected for as the optimiser [66]. Its learning rate was set to 0.001, decay to e^{-6}, and momentum to 0.9.

All training was performed over five epochs, with 128 elements in each minibatch.

4.1.3 Graphical User Interface

The Universal Shogi Protocol (USI) [72] was implemented to enable easier testing. This protocol allows shogi engines to communicate with Graphical User Interface (GUI) programs such as WinBoard [73], which was selected for this project (Figure 15).

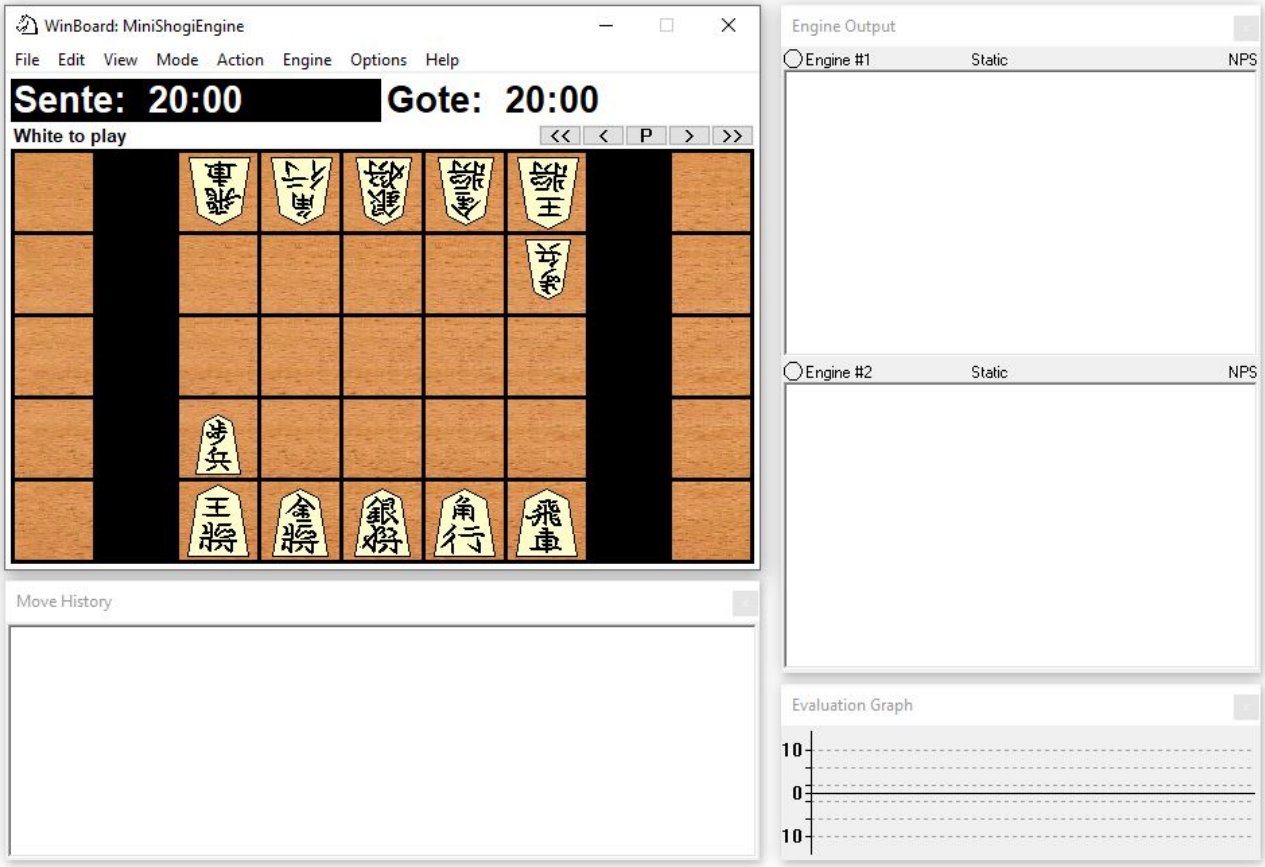

Figure 15: WinBoard configured to work with minishogi

WinBoard provides GUI with multiple different modes such as player vs AI and AI vs AI. The protocol itself will be discussed in more detail in the implementation section.

4.2 Alternative Approaches

4.2.1 Tree Search

As was mentioned before, the aim of this project was to verify if an algorithm for a game like minishogi can be learned using self-play and Deep Neural Networks. Multiple reasons were found in favour of this approach.

Minimax, which was discussed previously, cannot be applied to problems with a big search space because it requires to know the entire game tree. The improved version of Minimax,

Alpha-Beta search, has already been successfully applied to many board games, including chess. The downside of using it, however, are the large requirements for the computational resources to run the program. According to Russell [29], evaluating even a million nodes per second for chess only allows the algorithm to look five turns ahead, and considering the fact that the average branching factor for minishogi is unknown, it was decided not to use a method that depends so much on the branching factor being reasonably small.

Q-learning also was not selected for this project. According to Bertsekas [37], Q-learning is often impractical for problems with a large number of state-control pairs because there may be too many factors to update. An article by Wang [74] demonstrates that, without enhancements, the win rate of a Q-learning algorithm against a random agent in a 5x5 noughts and crosses game is just around 55% - only a little bit better than random play.

TD-learning was considered as one of the possible problem approaches. However, according to Tesauro [8], "In more complex games such as chess and Go, one would guess that an ability to learn a linear function of the raw board variables would be less useful than in backgammon. In those games, the value of a particular piece at a particular board location is more dependent on its relation to other pieces on the board. A linear evaluation function based on the raw board variables might not give very good play at all - it could be substantially worse than beginner-level play". Due to this, TD-learning was not selected as well.

As a result, the AlphaZero approach was selected for this project. It was chosen because of low resources demand during runtime and also because the algorithm itself is a relatively new approach that has not been explored thoroughly. Another benefit of selecting MCTS with Deep Neural Networks is the fact that the solution is model-free and learns entirely through self-play - which would be a problem otherwise due to the difficulty of finding a large enough dataset of minishogi games. Finally, the implementation of AlphaZero-like algorithms relies on the implementation of Monte-Carlo Tree Search, which, by itself, can be used as a benchmark for testing.

4.2.2 Alternative Neural Networks

A simple multi-layer convolutional neural net and ResNet were selected as the neural network for the project. The former was selected because of its low complexity, mainly because it has fewer trainable parameters. ResNet was selected as a second option based on two factors. First of all, as was mentioned in the previous section, it has shown good performance when applied to image recognition tasks, partially because of dealing with the vanishing gradient problem. It has also been used in the original implementation of AlphaZero, which indicates that it can be trained to work with high-dimensional game data.

While other neural networks designs such as DenseNet [63] or AlexNet [75], it was decided to use ResNet instead due to their complexity.

SGD with Nesterov Momentum was selected for the optimiser due to its popularity and well-known good performance. [66]. Adabound [68] was not selected due to it being fairly new and, hence, not being natively supported by popular machine learning libraries.

4.3 Project Organisation

An agile approach was selected for the project development. Each development cycle typically lasted about a week and included implementation of the new features, testing and analysis. The vast majority of research was done during the autumn term, with several bits being researched during sprints.

Python was selected as the primary language for this project, mainly due to it being easy to use for prototyping and a wide support of machine learning libraries. Keras with Tensorflow backend was selected as the neural network library. Keras provides a user-friendly wrapper interface around Tensorflow but still allows access to low-level features like session management. Tensorflow also come with a tool named Tensorboard, which allows to build real-time graphs of the loss change, build visual representations of the model and view per-layer histograms of weight and output distributions.

Git was selected as a version control tool with GitLab being used as the remote repository. It was also used for issue tracking and scheduling.

The Object-Oriented Programming paradigm was selected for the program. This was done to simplify the development of new modules (such as different agents for playing the game) and to ensure that the program can be extended to other games as well.

5 Implementation

As was mentioned before, the implementation process consisted of three major steps: developing the game engine, creating an implementation of MCTS and creating a neural network. This section will cover all three stages from the technical point of view.

5.1 General Layout

The project has been organised into multiple modules. The modules used are:

- *games*: contains an abstract implementation of a game engine as well as an implementation of a minishogi engine.
- *agents*: contains numerous agents that define the logic of making decisions in the game.
- *mcts*: contains two implementations of Monte-Carlo Tree Search: with and without a neural network.
- *nnets*: contains multiple implementations of neural networks.
- utils: contains various additional utilities.

The high-level structure of different classes an modules has been summarised in figure 16

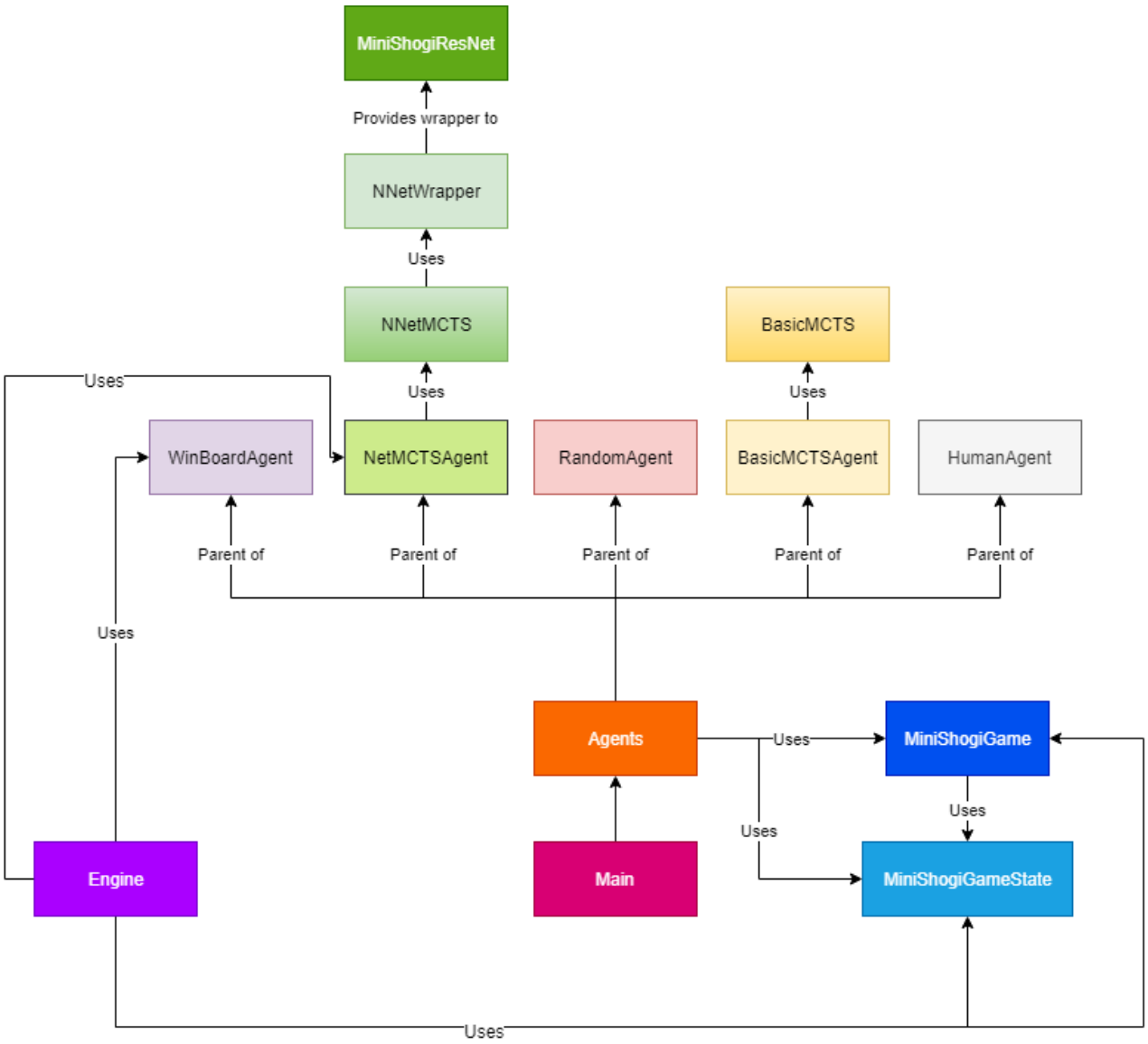

Figure 16: Program design overview

Apart from the aforementioned modules, the program contains two points of entry: *main.py* and *engine.py*. *main.py* is the point of entry for network training and testing it using various benchmark agents. *engine.py* loads the latest neural network and starts listening for USI commands. This script is only used for testing the network with WinBoard.

All communication between WinBoard and the game engine is done entirely through the standard process input and output, which gets redirected when the script is launched by WinBoard. The engine supports the following commands:

- *quit* - terminates the process
- *setboard* - initialises the game agent and loads weights of the most successful neural network
- *force* - tells the engine to send response straight after it receives input from WinBoard

- *go* - tells the engine to send its move
- *ping* - used for checking if the engine is running
- *protover* - requests supported features from the engine
- *usermove* - sends the engine users move using USI format

config.py, which is also located in the root of the project, uses a dictionary with various configuration settings, including setting for the neural network, exploration constant value and number of self-play games performed before each evaluation step. For convenience, this dictionary has been implemented as a dictionary with dot access, which allows to use a dot to access elements as if they were attributes of an object.

There are two extra directories, *logs* and *checkpoints*. *logs* contains log files from various modules of the program, as well as monitoring data generated by Tensorboard.

checkpoints contains snapshots of network weights and training data of each training epoch.

5.2 Game engine

The game engine class used in this project is an abstract class called *Game*, which has two methods:

```
class Game(ABC):
        def __init__(self):
                super(Game, self).__init__()

        @abstractmethod
        def take_action(self, action):
                pass

        @abstractmethod
        def move_to_next_state(self, next_state):
                pass
```

Both methods define transitions between different states of an abstract game. The first method, *take_action*, accepts an abstract action argument, while the second method, *move_to_next_state*, performs the transition based on the state argument.

The information about the current state of the game is stored inside another class, *GameState*. This class represents the state information in two ways: as a 3D stack of planes and as a collection of attributes. This is done because the neural network requires a matrix for its input, but checking rules and performing state transitions is a lot easier with an object with attributes. Because of that, this class contains two class methods, which allow to create it both from a matrix or from an arguments tuple:

```
@classmethod
@abstractmethod
def from_plane_stack(cls, stack):
        pass

@classmethod
@abstractmethod
def from_board(cls, args):
        pass
```

A similar thing has been done for actions. There are two ways to represent all legal actions for a single state: either as a 3D matrix (the exact encoding for minishogi will be discussed later in this report) or as an array of tuples (x, y, z), where each tuple represents one action. The former is used with the neural network, whereas the later can be easily used while implementing game logic. To accommodate for both methods, numerous support methods were created to perform conversions.

Overall, the class defines the following methods:

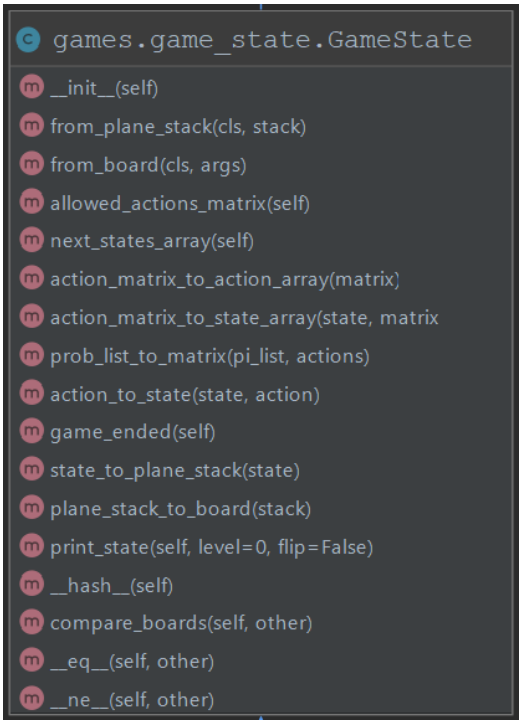

Figure 17: WinBoard configured to work with minishogi

The same package contains minishogi implementations of these classes. *mini_shogi_game.py* defines various aspects of the game such as board size, number of different actions, the size of the promotion zone, etc. It also uses a two-dimensional list to represent the starting layout of the game board. In this list, the following notations are used for different pieces:

- p - Pawn

- k - King
- s - Silver General
- g - Gold General
- r - Rook
- b - Bishop

A plus is added before the piece name to indicate that it has been promoted. All white pieces are represented with lower case characters and the black pieces are represented with upper case characters.

As was mentioned previously, a three-dimensional matrix is used as a way to represent states of the game. The size of this matrix for minishogi is 5x5x32, where 5x5 planes describe the following:

- First 10 describe positions of current player's pieces
- Next 10 describe positions of the other player's pieces
- 7 layers describe the current player's hand
- 7 layers describe the opponent's hand
- 1 layer describes the current colour
- 1 layer describes the amount of turns taken since the beginning of the game.

Piece layers use one-hot encoding: for each layer, a zero is used if there is no such piece in that position and one if there is. For example, at the beginning of the game the first layer, which describes all pawns that belong to the current player, is filled with zeros in all cells except the one at (0,3), which is where the pawn is located.

Both groups of hand layers correspond to a single piece type. All 25 cells are filled with the same number, which represents the amount of pieces of a particular type in the players hand.

The final layer is also entirely filled with the same number, the amount of turn since the start.

A similar approach is used to encode actions. The action stack of planes is a 5x5x69 matrix. In this matrix, each layer represents a specific action:

- The first 32 layers encode all possible actions: there are 8 directions of movement and each move move up to 4 squares
- The next 32 layers encode promotion moves, which are only available when a piece either enters or leaves the promotion zone

- The remaining 5 layers denote the drops of five different types of pieces from the players hand

The same class overrides two functions that are defined by its parent. Both *take_action* and *move_to_next_state* implement a high level transition between states: they add the next state to the state history and change the instance variable *game_state*, which is an instance of the *mini_shogi_game_state* class that keeps track of the current state of the game:

```
def take_action(self, action):
        next_state = MiniShogiGameState.action_to_state(self.game_state, acti
        self.state_history.append(next_state)
        self.game_state = next_state

def move_to_next_state(self, next_state):
        self.state_history.append(next_state)
        self.game_state = next_state
```

The same class also defines *piece_actions*, a class function that returns a 2D array that describes all legal actions for a game piece. This array two indices to encode these actions. The first one is $direction \in [0, 7]$, which describes 8 possible directions of movement, starting from top and continuing clockwise. The second index is the magnitude of the movement, which ranges from 0 (which indicates 1 step in that direction) up to the maximum magnitude, defined as $max(board\_width, board\_height) - 1$.

Two conversion methods, *get_coordinates* and *get_direction* are implemented in the same class as well. They are used for conversion between a tuple *(x, y, new_x, new_y)* and a tuple *(x, y, direction, magnitude)*, where each describes a single piece movement.

The same class also contains some miscellaneous methods that check and perform promotions and demotions of game pieces.

The *mini_shogi_game_state* class contains information that encodes a minishogi game state. It defines the following attributes:

- board - a 2D list describing the board
- hand1 - a list of pieces in the current player's hand
- hand2 - a list of pieces in the opponent's hand
- colour - current players colour (either 'W' or 'B')
- move_count - a number that indicates how many moves have passed since the beginning of the game.

The board list uses the same notations as the ones used in the Game class. It is also always stored relative to the current player - despite their colour, the top-left corner is always (0,0), their pieces are always lower case and the opponent pieces are always upper case.

The same class implements conversion methods mentioned in the description of *game_state*. All of them have a similar structure: a combination of three for loops is used to either create or traverse a 3D matrix.

The final part of this class is hashing. All states are later used as keys in various dictionaries, which requires from the object to be hashable. All hashing is performed using the the standard *hash()* Python method. For performance reasons, instead of calculating the hash inside the *__hash__()* function, it is only calculated when the object is modified and then the result is stored in an object attribute. This significantly improves performance of comparison operations, which will be further analysed in the Discussion section.

5.3 Monte-Carlo Tree Search

Two versions of Monte-Carlo Tree search have been implemented. The first version, *BasicMCTS*, performs simple MCTS search. Using a for loop, it looks at the current node's children and chooses a node to explore either using the UCT formula [43] or at random, depending on whether the children of the node have all been explored or not. The exploration stops either when the tree depth exceeds a limit or if a terminal node is found. Once it stops, the result of the simulation is propagated up the tree.

The algorithm can be summarised with the following pseudocode:

```
ucb(state) = q[state]/n[state] +
        c * sqrt(ln(n[state.children()]) / n[state])

for i in range(max_depth):
        children = current_state.children()
        if children.explored():
                next_state = max(ucb(children))
        else:
                next_state = random(children)

        if game.ended():
                break
        else:
                current_state = next_state

for state in visited_states:
        n[state] += 1
        if state.game_ended() and player == winner:
                self.q[state] += 1
```

Here, n is the number of visits and q is the number of wins.

The second class, *NNetMCTS*, contains a modification of this algorithm. It uses four dictionaries to keep track of various variables:

- *n_sa*: keeps track of how many times an action has been taken from a state
- *q_sa*: q values for taking an action from a state, as define in Equation (6)
- *p_s*: keeps track of the probability vector returned by the neural net
- *n_sa*: keeps track of how many times a node has been visited

The search itself happens inside the *search* function. At the beginning of each iteration it check if the node has been visited by checking if it has a probability vector associates with it. If it doesn't, then the neural network is used to calculate it. Once the value from the neural network is obtained, it is then reshaped to match the shape of an the action matrix. After that, elementwise multiplication is performed to mask illegal moves and all remaining moves are renormalised. Once it is done, the loop is exited.

A slightly different approach is taken if, after masking, the probability vector is equal to zero in all dimensions. That means that the network completely failed to make an accurate prediction - usually due to overfitting or a particularly large learning rate. In such case a warning message is logged and all moves receive an identical value associated with them, which simulates a random choice.

If the current node is not terminal and also is not a leaf node, the CPUCT formula from Equation (5) is used to determine the next node to consider and the loop repeats again

Once a terminal node or a leaf node are reached, the algorithm updates the n-value and the q-value for each node, as defined in Equation (6). This is done for all nodes that were used in the search path.

Overall, the algorithm can be summarised with the following snippet of pseudocode:

```
for i < max_length:
        if s.game_ended():
                v = -1
                break

    if s not in p[s]:
        p[s], v = nnet.predict(s)
        break

        if s.root():
                apply_dir_nosie(p[s])

    for a in actions(s)):
                u = Q[s][a] + c_puct*p[s][a]*sqrt(sum(n[s]))/(1+n[s][a])

        next_action = max(u)

    q[s][a] = (n[s][a]*q[s][a] + v)/(n[s][a]+1)
        n[s][a] += 1
```

The same class also contains a function for gathering visit statistics per each node. It is used for choosing the next action after all MCTS simulations are finished. The function generates a list with a value for each action, where each value is proportional to the number of visits of the corresponding node and the sum of all values is equal to 1. An if statement is also used to check if the temperature value τ is equal to zero - if it is, the returned list simply contains 1 for the most frequent move and 0 for all other moves.

5.4 Agents

Multiple agents have been implemented to test the approach and perform training. All of them extend the same abstract class, *Agent*, which contains a single abstract method *act*. This method is used to simulate perform transitions between different states of the game.

5.4.1 MCTS Neural Network Agent

This agent performs three main tasks: model training, evaluation and move selection. The function called *train_neural_net* loads the file with training examples, as well as the file with optimal neural network weights. It then uses a loop to perform self-play and appends all training examples to the main example list. The examples are then renormalised using a function from the *util* module.

Once the normalisation process is complete, the network is duplicated using a temporary weight checkpoint. It is then evaluated using the *simulate* function, which pits the networks against each other and returns the win rate for each of them. This result is then either used to either accept or reject the new network.

The high-level pseudocode for this process looks like this:

```
for i < training_epochs:
        for i < example_games_number:
                agent.initialise()
                play_one_game()
                save_results()

        new_agent = agent.copy()
        new_agent.nnet.train(examples)

        win_rate = compare(agent, new_agent)

        if win_rate > 0.55:
                agent = new_agent
```

Another important part of this class is the *act* function. This function runs teh MCTS simulation the predefined number of times and then chooses an action using a probability

distribution vector returned by the search. It also has an if statement that checks if the class is being used with WinBoard, and if it is, it prints the selected action to the console.

5.4.2 WinBoard Agent

This agent is used entirely for parsing actions from WinBoard. It receives a USI-formatted action which is then converted into the format used by the program. Simple validation is also performed in order to let WinBoard know that the engine thinks that the move is invalid.

5.4.3 Other Agents

Numerous other agents were made for testing. *HumanAgent* provides a simple console-based interface that can be used for manual testing. *MCTSAgent* provides an implementation of a simple agent that makes its decisions based on the results it gets from the *BasicMCTS* class. The final agent, *RandomAgent*, chooses all actions randomly and is only used for testing.

5.5 Neural Network

Two classes were used for the implement the neural network. *MiniShogiResNet* contains a Keras implementation of ResNet, as defined in the Solution Approach section. The class simply consists of a layer-by-layer definition of the neural network, with all parameters loaded from the configuration file. It accepts a stack of planes as an input, and treats it as a 5x5 image with 32 different channels. The output of the neural network is a single value v and a 1725-dimensional action probability vector. The vector is then rescaled into a 5x5x69 matrix before being used by the MCTS class.

As was mentioned in the solution approach, the loss function of the network was a combination of categorical cross entropy and mean square error[52], as shown in Equation 9

The second class, *NNetWrapper*, provides a wrapper interface for the neural network. It defines functions for loading and saving network weights, preprocessing and postprocessing data and managing Tensorflow sessions.

The former need to be done manually because of two parallel neural networks being used for predictions at the same time. Because of how Tensorflow works, each Tensoflow section has exactly one model graph associated with it. To overcome it, a separate session needs to be started for every neural network. This was achieved by saving the session in an attribute and using this attribute before performing any operations with the network.

6 Testing: Verification and Validation

6.1 Testing

Due to the agile nature of the project, testing played a very important role in the development cycle. Considering the incremental nature of the feature addition, each new class needed to be tested, both separately and with integration tests.

Most tests were implemented as the project was being developed. These were usually simple scripts that compared the expected output of the module with the actual output. The following scripts were used: For *MiniShogiGame* and *MiniShogiGameState*, the following parts of the program were tested:

- matrix-to-object conversion of a state
- object-to-matrix conversion of a state
- tuple-to-matrix conversion of an action
- matrix-to-tuple conversion of an action
- action matrix to state conversion
- comparison function
- correctness of state transitions
- promotions/demotions
- validity of piece actions

For *NetMCTS* and *BasicMCTS*, all testing was performed after the corresponding agent classes were created. They were then tested both using pre-written tests and by analysing the log of self play and play against the random agent.

Logging was used to monitor all components of the project. While it was deleted from the final version for performance reasons, the older versions of the program contained numerous information messages that described statistics like the exploration ratio for each node, probabilities, Q-values and various other values.

Profiling was also used to measure the algorithm's performance. cProfile allowed to gather data that showed how many times different methods are executed and how long it takes to run them. This was done multiple times during the development stage to ensure that the performance of the program is optimised.

Due to the research nature of the project, only minimal testing was done to ensure that the solution is correct. Only necessary input validation was performed because the produced program in not intended for a direct consumer use. A bigger focus was given to ensuring the correctness of all methods that were used in the project, as well as ensuring the best possible performance.

6.2 Results and Validation

Due to the hardware and time restrictions, the training process was performed only for 48 GPU hours. The network was trained on a single GPU, with occasional breaks to avoid damaging the hardware. It total, the program went through 22 generations of neural networks and only 7 of them were accepted after showing better performance than their predecessor. Towards the beginning of the training process, the average difference in the win rate of accepted networks was about 70%, which went down to 57% as the program learned to defend against straightforward attacks. The total number of 42060 states were generated, with 31962 game states being left after removing duplicates. The total of 4400 games was played, making an average length of a single game being equal to 10 moves.

Multiple benchmarks were used to validate the result. As was mentioned before, Tensorboard was used to perform testing and monitoring during training. It provided real-time graphs that showed the change in loss over time, which was used for tweaking the number of training epochs, learning rate and minibatch sizes. The loss change over time is shown in Figures 18, 19 and 20.

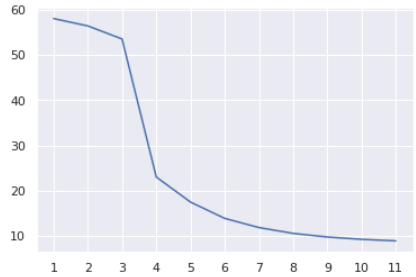

Figure 18: Combined loss

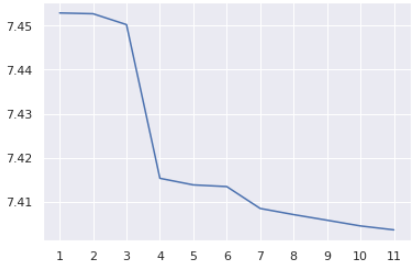

Figure 19: Policy loss

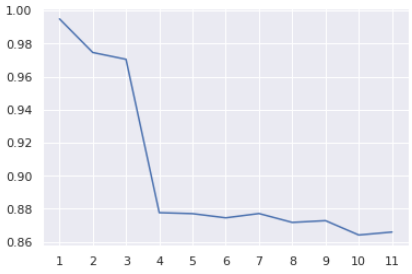

Figure 20: Value loss

A validation set with 0.1% of training examples was also used to monitor overfitting. Its loss was also carefully monitored throughout training (Figure 21).

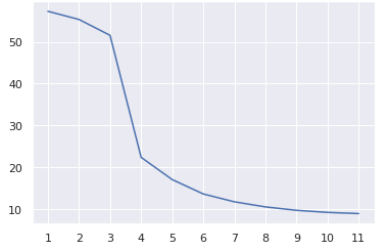

Figure 21: Validation loss

Tensorboard was also used to produce heat maps of network outputs. While these were not particularly useful due to the high dimensionality of the input, they were still used towards the beginning of training to ensure that the output of the neural network varies for different actions. It was also possible to see histograms of outputs of different layers, including two neural network heads. Outputs for the final version of the neural network have been shown in Figures 22 and 23.

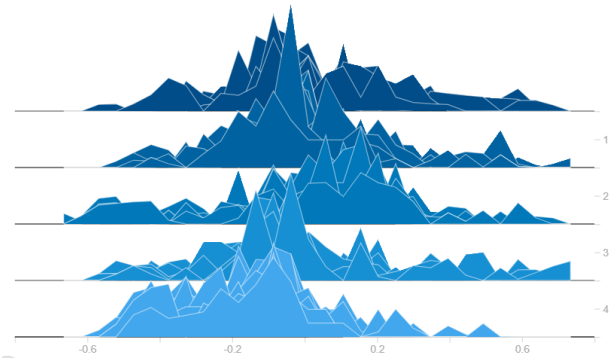

Figure 22: Value output distribution

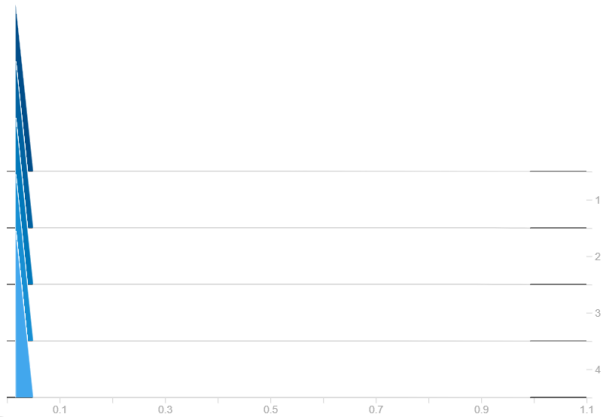

Figure 23: Policy output distribution

Four different agents were used to evaluate the solution. The network was evaluated multiple times against a random agent, which the expected win rate being close to 100%. The same applied to testing the network against a basic MCTS agent. To make this method fairer, the same amount of MCTS simulations was used.

The final type of validation involved testing the neural network against human players. Two levels of difficulty were selected. First of all, a person who had no prior shogi experience was selected. All rules for the game were explained immediately before the game, and 5 matches were played. For each match, 800 simulations were used for the program, which resulted in an average thinking time of 10 seconds. The second human baseline was a player with moderate shogi experience. The same number of simulations was used, but thinking time for the human player was also limited to 10 seconds.

All results of testing the network against other agents are presented in Table 1.

	Random Agent	MCTS Agent	Novice Human Agent	Experienced Human Player
NNetMCTS Wins	30	30	3	1
NNetMCTS Losses	0	0	2	4
Win Rate	100%	100%	60%	20%

Table 1: Results

7 Discussion: Contribution and Reflection

7.1 Discussion

TODO

Numerous games were also played with the program to study the strategies developed by the neural network. It was found that soon after training the network learned a simple move (Rook e2-e4), which presented immediate danger to the opponent's King (Figure 24).

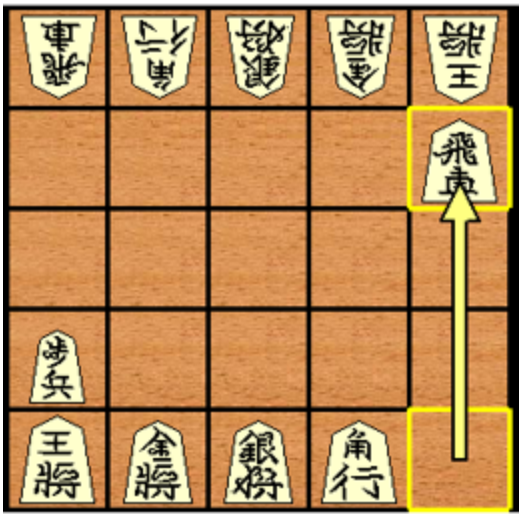

Figure 24: White Rook attacking Black King

Even though this move does not present any danger to any experienced player who can simply capture the Rook with its King or Golden General, the network needed to learn how to defend against this move on training examples before being able to counter it. This led to a significant increase of the value estimation of this move for the several generations of the network. Over time, the network managed to learn to counter this move, and the estimation of such action went down as it gave a substantial disadvantage for the attacker who simply loses one of their pieces.

Another observation was found during the competitive matches against human opponents. The network seemed to learn basic strategies for opening moves (such as positioning pieces so they 'cover' each other, e. g. Figure 25), but seemed to struggle towards the endgame part of the matches. It was able to avoid direct losses by moving the King and placing pieces between the King and the attacking pieces, but this was not enough to avoid the eventual loss.

Figure 25: A raised Silver General being covered by two other Pieces

It was also found during training that the network did not fully learn to pick up some behaviour which seems intuitive for a human player.

7.2 Reflection

500

8 Social, Legal, Health & Safety and Ethical Issues

1k

9 Conclusion and Future Improvements

1k

References

[1] C. E. Shannon, "Xxii. programming a computer for playing chess", *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950.

[2] A. L. Samuel, "Some studies in machine learning using the game of checkers. ii-recent progress", in *Computer Games I*, Springer, 1988, pp. 366–400.

[3] H. Remus, "Simulation of a learning machine for playing go.", in *IFIP Congress*, 1962, pp. 428–432.

[4] A. L. Zobrist, "A model of visual organization for the game of go", in *Proceedings of the May 14-16, 1969, spring joint computer conference*, ACM, 1969, pp. 103–112.

[5] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, "Chinook the world man-machine checkers champion", *AI Magazine*, vol. 17, no. 1, pp. 21–21, 1996.

[6] T. Anantharaman, M. S. Campbell, and F.-h. Hsu, "Singular extensions: Adding selectivity to brute-force searching", *Artificial Intelligence*, vol. 43, no. 1, pp. 99–109, 1990.

[7] G. Tesauro and T. J. Sejnowski, "A parallel network that learns to play backgammon", *Artificial Intelligence*, vol. 39, no. 3, pp. 357–390, 1989.

[8] G. Tesauro, "Temporal difference learning and td-gammon", *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[9] G. Tesauro, "Td-gammon, a self-teaching backgammon program, achieves master-level play", *Neural computation*, vol. 6, no. 2, pp. 215–219, 1994.

[10] S. Thrun, "Learning to play the game of chess", in *Advances in neural information processing systems*, 1995, pp. 1069–1076.

[11] M. Enzenberger, "The integration of a priori knowledge into a go playing neural network", 1996.

[12] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu, "Deep blue", *Artificial intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[13] M. Lai, "Giraffe: Using deep reinforcement learning to play chess", *arXiv preprint arXiv:1509.01549*, 2015.

[14] B. Sheppard, "World-championship-caliber scrabble", *Artificial Intelligence*, vol. 134, no. 1-2, pp. 241–275, 2002.

[15] M. Moravcik, M. Schmid, N. Burch, V. Lisy, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker", *Science*, vol. 356, no. 6337, pp. 508–513, 2017.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning", *arXiv preprint arXiv:1312.5602*, 2013.

[17] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, "Emergent complexity via multi-agent competition", *CoRR*, vol. abs/1710.03748, 2017. [Online]. Available: `http://arxiv.org/abs/1710.03748`.

[18] BBC news. (Mar. 2016). Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol, [Online]. Available: https://www.bbc.com/news/technology-35785875 (visited on 04/12/2019).

[19] 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. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge", *Nature*, vol. 550, p. 354, Oct. 2017. [Online]. Available: https://doi.org/10.1038/nature24270.

[20] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm", *CoRR*, vol. abs/1712.01815, 2017. [Online]. Available: http://arxiv.org/abs/1712.01815.

[21] P. Liskowski, W. Jaskowski, and K. Krawiec, "Learning to play othello with deep neural networks", *IEEE Transactions on Games*, vol. 10, no. 4, pp. 354–364, 2018.

[22] David Foster. (Jan. 2018). How to build your own AlphaZero AI using Python and Keras, [Online]. Available: https://medium.com/applied-data-science/how-to-build-your-own-alphazero-ai-using-python-and-keras-7f664945c188 (visited on 04/12/2019).

[23] Encyclopedia.com. (2019). Branching factor, [Online]. Available: https://www.encyclopedia.com/computing/dictionaries-thesauruses-pictures-and-press-releases/branching-factor.

[24] H. Iida, M. Sakuta, and J. Rollason, "Computer shogi", *Artificial Intelligence*, vol. 134, no. 1, pp. 121–144, 2002, ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00157-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370201001576.

[25] H. Adachi, H. Kamekawa, and S. Iwata, "Shogi on n x n board is complete in exponential time", *Ieice Transactions - IEICE*, J70-D:1843–1852, 1987.

[26] T. Ito and K. Tanaka, "Changes in cognitive processes upon learning mini-shogi", Apr. 2019.

[27] Ancientchess.com, *How to Play Japanese Chess - Shogi*. [Online]. Available: http://www.ancientchess.com/page/play-shogi.htm (visited on 04/12/2019).

[28] Encyclopaedia Britannica. (Oct. 2018). Game theory, [Online]. Available: https://www.britannica.com/science/game-theory (visited on 04/12/2019).

[29] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009, ISBN: 0136042597.

[30] M. Buro, "From simple features to sophisticated evaluation functions", in *International Conference on Computers and Games*, Springer, 1998, pp. 126–145.

[31] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 2nd. Cambridge, MA, USA: MIT Press, 2018, ISBN: 0262193981.

[32] Shweta Bhatt. (Mar. 2018). 5 Things You Need to Know about Reinforcement Learning, [Online]. Available: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html (visited on 04/12/2019).

[33] M. Bowling and M. Veloso, "An analysis of stochastic game theory for multiagent reinforcement learning", Carnegie-Mellon Univ Pittsburgh Pa School of Computer Science, Tech. Rep., 2000.

[34] M. van Otterlo and M. Wiering, "Reinforcement learning and markov decision processes", in *Reinforcement Learning*, Springer, 2012, pp. 3–42.

[35] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey", *CoRR*, vol. cs.AI/9605103, 1996. [Online]. Available: `http://arxiv.org/abs/cs.AI/9605103`.

[36] C. J. Watkins and P. Dayan, "Q-learning", *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[37] D. P. Bertsekas, *Dynamic Programming and Optimal Control, Vol. II*, 3rd. Athena Scientific, 2007, ISBN: 1886529302.

[38] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai.", in *AIIDE*, 2008.

[39] M. Kearns, Y. Mansour, and A. Y. Ng, "A sparse sampling algorithm for near-optimal planning in large markov decision processes", *Machine learning*, vol. 49, no. 2-3, pp. 193–208, 2002.

[40] Y. Sato, D. Takahashi, and R. Grimbergen, "A shogi program based on monte-carlo tree search", *Icga Journal*, vol. 33, no. 2, pp. 80–92, 2010.

[41] J. Nijssen, "Playing othello using monte carlo", *Strategies*, pp. 1–9, 2007.

[42] T. Cazenave and A. Saffidine, "Score bounded monte-carlo tree search", in *International Conference on Computers and Games*, Springer, 2010, pp. 93–104.

[43] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning", in *European conference on machine learning*, Springer, 2006, pp. 282–293.

[44] P. Auer, "Using confidence bounds for exploitation-exploration trade-offs", *Journal of Machine Learning Research*, vol. 3, no. Nov, pp. 397–422, 2002.

[45] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods", *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[46] G. M.-B. Chaslot, M. H. Winands, and H. J. van Den Herik, "Parallel monte-carlo tree search", in *International Conference on Computers and Games*, Springer, 2008, pp. 60–71.

[47] H. Mannen, "Learning to play chess using reinforcement learning with database games", 2003.

[48] R. S. Sutton, "Learning to predict by the methods of temporal differences", *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[49] J. Baxter, A. Tridgell, and L. Weaver, "Learning to play chess using temporal differences", *Machine Learning*, vol. 40, no. 3, pp. 243–263, 2000.

[50] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search", *nature*, vol. 529, no. 7587, p. 484, 2016.

[51] D. Auger, A. Couetoux, and O. Teytaud, "Continuous upper confidence trees with polynomial exploration–consistency", in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2013, pp. 194–209.

[52] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[53] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[54] Adit Deshpande. (Jul. 2016). A Beginner's Guide To Understanding Convolutional Neural Networks, [Online]. Available: `https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/` (visited on 04/16/2019).

[55] Sumit Saha. (Dec. 2018). A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way, [Online]. Available: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53` (visited on 04/16/2019).

[56] D. Cireşan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification", *arXiv preprint arXiv:1202.2745*, 2012.

[57] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach", *IEEE transactions on neural networks*, vol. 8, no. 1, pp. 98–113, 1997.

[58] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda, "Subject independent facial expression recognition with robust face detection using a convolutional neural network", *Neural Networks*, vol. 16, no. 5-6, pp. 555–559, 2003.

[59] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[60] Vincent Fung. (Jul. 2017). An Overview of ResNet and its Variants, [Online]. Available: `https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035` (visited on 04/16/2019).

[61] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient flow in recurrent nets: The difficulty of learning long-term dependencies", in *A Field Guide to Dynamical Recurrent Neural Networks*, S. C. Kremer and J. F. Kolen, Eds., IEEE Press, 2001.

[62] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks", in *European conference on computer vision*, Springer, 2016, pp. 630–645.

[63] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[64] S. Kotz, N. Balakrishnan, and N. Johnson, *Continuous Multivariate Distributions, Volume 1: Models and Applications*, ser. Continuous Multivariate Distributions. Wiley, 2004, ISBN: 9780471654032. [Online]. Available: `https://books.google.co.uk/books?id=EbPBXJ-N-m4C`.

[65] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[66] S. Ruder, "An overview of gradient descent optimization algorithms", *arXiv preprint arXiv:1609.04747*, 2016.

[67] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning", in *Advances in Neural Information Processing Systems*, 2017, pp. 4148–4158.

[68] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate", *CoRR*, vol. abs/1902.09843, 2019. [Online]. Available: `http://arxiv.org/abs/1902.09843`.

[69] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: `http://arxiv.org/abs/1707.06347`.

[70] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[71] A. F. Agarap, "Deep learning using rectified linear units (relu)", *arXiv preprint arXiv:1803.08375*, 2018.

[72] T. Romstadt. (2007). The universal shogi interface, [Online]. Available: `http://hgm.nubati.net/usi.html` (visited on 04/23/2019).

[73] F. S. Founation. (2007). Xboard, [Online]. Available: `https://www.gnu.org/software/xboard/manual/xboard.html` (visited on 04/23/2019).

[74] H. Wang, M. Emmerich, and A. Plaat, "Monte carlo q-learning for general game playing", *CoRR*, vol. abs/1802.05944, 2018. arXiv: `1802.05944`. [Online]. Available: `http://arxiv.org/abs/1802.05944`.

[75] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

10 Appendices

10.1 Appendix A: Logbook

todo

10.2 Appendix B: PID

todo