Applying Reinforcement Learning with Monte Carlo Tree Search to The Game of Draughts

Effects of Multiple Trees and Neural Networks

Li Meng

January 24, 2020

Supervisors: Dr. Marco Wiering (Artificial Intelligence, University of Groningen)
and Matthia Sabatelli (University of Liège)

Artificial Intelligence
University of Groningen, The Netherlands

ABSTRACT

Draughts is a popular game among many regions in the world. In particular, international draughts has a game complexity around 10^{30}, which is currently considered unsolvable. Hence, applying the Monte Carlo Tree Search (MCTS) algorithm on international draughts and analyzing the playing strength are interesting research objectives.

Besides the baseline MCTS algorithm similar to AlphaZero, three different variations of the MCTS algorithm are compared in our experiment. Two of them use multiple neural networks inspired by domain-specific heuristics of draughts or the multiple search tree MCTS. The hybrid algorithm is a combination of both heuristics and multiple search trees.

The results of our experiments show that MCTS is indeed capable of improving its playing skills of draughts. All MCTS algorithms are capable of beating a random player, but no algorithms can stably best a player using the Alpha-Beta algorithm with depth 2. The most dominant parameters behind the bad performances are the size of the neural networks and the number of MCTS simulations. The number of input channels and the amount of training examples are also considered crucial.

The results of the method with multiple search trees are the best among all MCTS algorithms, which proves that the coordination of policies and values between different search trees can improve the performance of the MCTS algorithm. On the other hand, the usage of domain-specific heuristics is considered insufficient to offset the deficit caused by decreasing the size of neural networks.

Acknowledgements

First of all, I would like to extend my greatest appreciation for the excellent supervision and guidance to Marco Wiering and Matthia Sabatelli during this project. Meanwhile, I am grateful for the long lasting support and encouragement from my family. I also would like to thank the staff from the AI faculty and Peregrine HPC cluster who provided the necessary support to my project.

Contents

LIST OF FIGURES

LIST OF TABLES

1
Introduction

Artificial Intelligence (AI) is a research field that studies intelligent agents. Researchers have tried to define "intelligence" in mathematical formulas [1], yet the formal definition of universal intelligence did not give quantitative measures about how much an agent can be considered intelligent. On the other hand, studying intelligent behaviour of agents in games has been considered useful for gradually advancing towards true understanding of intelligence [2]. Games are in a much narrower scenario of intelligence compared to the complexity of the real world. In a game environment, there are typically well-defined rules about eligible actions agents can take under each state, and under what states the game ends in favour of which side of the players. This entails a limited action space and time space, making quantitative measures of intelligence applicable. As our way to study intelligence, the focus of this thesis is on applying the MCTS algorithm [3] to the game of draughts. Different variations of MCTS are compared by examining whether they are able to improve the training process and performances of agents in the draughts game or not.

1.1. Rules of Draughts

Draughts refers here to 10×10 international draughts. Illustration pictures of the draughts board are created using [4]. The 8×8 draughts version is considered a solved game after it was proven that the game can always end in a draw if neither side of the players makes mistakes [5]. On the other hand, international draughts is not solved and hence more interesting for our research.

In a draughts board, there are black and white sides. The black side is usually printed at the top of the board. Each player has 20 pieces at the beginning of the game. An illustration of the initial board position is shown in Figure **1.1**. The pieces can move diagonally and only 50 color grids of all 100 grids are used. Pieces can jump over the pieces of the other player. Jumped pieces are captured and later removed from the board. When players decide a move, they must always select the move that has the maximal amount of captured pieces. Only if there are multiple moves having the same capture amount, they can select a move among those. After a capture move, if the particular piece is able to continue capturing other pieces, the turn is not finished and it then must select a move among those legal moves. All captured pieces will only be removed after a sequence of moves is finished.

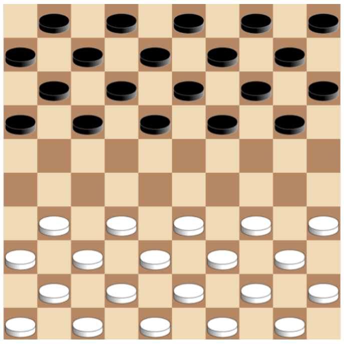

Figure 1.1: Initial board position

There are two kinds of pieces, ordinary pieces (men) and kings. Men can only move one distance at one time, but kings can fly at any distance. Kings can select any legal positions forward if not blocked by other pieces after a capture move, whereas men can only jump to the next grid. There are no kings at the beginning of the game. Men can be crowned to kings when it reaches the other side of the board, with an exception that the men need to jump backwards away from the last row if there are possible captures. Men are not crowned in this scenario. Men need to wait until the next turn to enjoy king's right, so it is not possible to jump using king's move path to capture other pieces immediately after being crowned.

The game ends whenever one player has no pieces to move, and then the other player wins. This can happen if one player has no pieces or all possible moves of pieces are blocked by opponent's pieces. A draw in draughts is also possible. The draw terms will be described in our game implementation.

1.2. THEORETICAL BACKGROUND

In many types of board games such as chess, AI players with Alpha-Beta search dominated the tournaments before the dawn of AlphaZero. Alpha-Beta search cuts the search complexity by limiting the search width of MinMax search [6], while MinMax search is a classic search algorithm in zero-sum games to find optimal moves by exhaustive search through all possibilities.

Alpha-Beta search usually performed much worse with neural networks because it propagates the maximal errors directly to the root of the tree. However, there are many optimization methods developed to improve the performance of Alpha-Beta search. Some of them focused on extra pruning techniques in addition to Alpha-Beta pruning, such as extended futility pruning. Extended futility pruning builds upon the traditional futility pruning technique and cuts the complete branch of a search tree based on mere static criteria, realizing forward pruning [7]. Some of them relate to the order of the moves, reordering move evaluations by using domain-dependent knowledge as heuristics. In the case of the famous chess program Deep Blue [8], an opening book created by human chess grand masters and a 700,000 game database used as extended book are utilized to select preferred moves when the game starts. In the game of draughts, game engines using Alpha-Beta search are also popular, yet those engines did not show dominant victory against human champions. There has been a match between the world champion of draughts Schwarzman and the program Maximus in 2012, in which Maximus lost against Schwarzman with 5 to 7 [9].

Deep learning has allowed the computer to play games in a manner similar to humans, which exhibited its great game-playing technique with temporal difference learning (e.g. deep Q-network) in games like Atari [10]. Nonetheless, the MCTS with deep learning method has outperformed the deep Q-network when playing Atari games, prior to AlphaZero [11].

AlphaZero is one of the most recent spectacular development in the AI field that deploys the method of MCTS with deep neural networks [3]. AlphaZero has defeated world champions from different game fields such as chess, Shogi, and Go. Many have believed that the way AlphaZero plays games is much more similar to humans than any other previous algorithm. AlphaZero has shown more generalized capability compared to its predecessor, the AlphaGo Zero program, which had previously defeated Lee Sedol, the world champion of Go. AlphaGo Zero first takes a database of human game plays as input, whereas AlphaZero trains the neural networks purely from the examples of its self-play.

Different from previous Alpha-Beta algorithm studies, there were also researches focusing on developing reinforcement algorithms based on self-play prior to the creation of AlphaZero. For instance, [12] studied temporal difference learning methods by comparing the learning from self-play and from the plays of experts in the game of backgammon. Those researches also extended to Othello. In particular, the idea of learning from the opponent's moves while playing against the player is inspiring [13]. AlphaZero focuses on a pure self-play learning approach and did not deploy domain specific heuristics. On the other hand, those kinds of heuristic techniques might also be found useful when the MCTS algorithms play draughts.

1.3. AlphaZero

The AlphaZero Program searches through the game-tree with MCTS only using generated simulations of games [3]. MCTS evaluates the value of each state using a trained neural network. The performance of MCTS improves dramatically as the search tree grows and the number of training epochs of the neural network accumulates.

Recent MCTS algorithms are typically combined with the Upper Confidence Bounds (UCB) [14] algorithm, which originates from the multi-armed bandit problem [15]. UCB strives to find an optimal balance between exploration and exploitation, and agents need to explore new actions that could lead to more profits but also to exploit empirical experience of the best actions so far. UCB applied to trees (UCT) [16] is a rollout based algorithm that samples episodes from the root of the tree repeatedly, and builds the search tree in a way that values from previous episodes are taken into consideration. UCT allows MCTS to converge towards optimality and has led to the breakthrough of applying MCTS on Go [17]. Other optimized algorithms based on probabilities include PUCB, which utilizes context to bias move decisions [18], and methods to narrow the search width based on probability distributions with the use of domain-dependent patterns [19].

AlphaZero uses a deep neural network, with value and policy networks as "head", and 19 residual blocks following a rectified batch-normalized convolutional layer as "body" [20]. Residual blocks are built using identity mappings that can serve as skip connections and after-addition activation. Residual Networks (ResNets) ease the training of deep neural networks particularly regarding vanishing and exploding gradient problems, and improve generalization. The CNN takes board representations as input and returns action probability vectors and the expected game outcome value (result value). The parameters of the neural network are trained from random initialization. In each Monte Carlo search, low-frequently visited moves with a high move probability and value according to the current neural network are selected. The search returns a policy vector based on the current state. The CNN is trained using the losses between the predicted probability vectors, the result value and those returned by the search tree.

1.4. RESEARCH QUESTION

The research question of our project is whether applying reinforcement learning (RL) with MCTS to draughts yields promising results similar to other board games or not. In other words, can it defeat traditional draughts algorithms with necessary training steps or not? How do the parameter settings in MCTS influence the performance of the algorithm?

Furthermore, how do the variations of MCTS affect the performance? Is it useful to create multiple MCTS search trees, or to use multiple neural networks guided by domain specific knowledge? Can the domain specific knowledge within draughts improve the performance of MCTS?

1.5. OUTLINE

The goal of this thesis is to examine the performances when the MCTS with deep neural networks method is applied to draughts, and to give insights on how different variations and implementations have an effect on the performances. Chapter 2 will describe our baseline MCTS algorithm that is similar to AlphaZero. Chapter 3 will describe our rationals to create the MCTS with multiple neural networks. Chapter 4 explains the algorithm of multiple MCTS search trees. The hybrid algorithm of multiple neural networks and multiple trees are described in Chapter 5. The implementation of our draughts game will be illustrated in Chapter 6. Chapter 7 details the parameter settings of our experiments and the reasons behind. The experiment results are shown in Chapter 8. Chapter 9 draws the conclusion of this thesis and discusses possible future work.

2 BASELINE METHOD

Our algorithms develop on the AlphaZero General, a lightweight general implementation of AlphaZero [21]. AlphaZero General provides a framework that can be used to build AlphaZero implementations of various kinds of games. Their learning algorithm was evaluated on 8×8 Othello and was able to beat random players and greedy players with one step look-ahead after 30 iterations [22]. In our implementation, the MCTS method combined with UCB is used to improve the policy that the CNN learns from the examples generated by self-play [23].

2.1. MCTS POLICY

MCTS searches from the root node s_{root} of the tree and two nodes s_i and s_j can have a directed edge $i \rightarrow j$ if there is a valid action a that can make state i transit to state j. The expected rewards of taking an action is denoted by Q-values $Q(s,a)$. During the procedure of self-play simulations, we maintain the numbers of visit counts of states as $N(s)$ and the numbers of taking an action a from state s as $N(s,a)$. The probability distributions of taking actions at the state s are denoted by $P(s)$, where $P(s) = \overrightarrow{p_\theta}(s)$ and $\overrightarrow{p_\theta}(s)$ is the prior probability distribution of taking actions at the state s that the CNN returns. Moreover, $P(s,a)$ is the probability of selecting action a in state s according to the probability distribution.

The MCTS search algorithm searches nodes with directed edges in the search tree by repeatedly selecting the action with the highest UCB value.

Our formula to calculate UCB values is slightly different from [14]. The formula is shown in Equation **2.1**. If $Q(s,a)$ does not exist, then it is initialized by only $N(s)$ and $P(s,a)$, as shown in Equation **2.2**.

$$U(s,a) = Q(s,a) + C_{puct} * P(s,a)\frac{\sqrt{N(s)}}{1+N(s,a)} \tag{2.1}$$

$$U(s,a) = C_{puct} * P(s,a)\sqrt{N(s)+EPS} \tag{2.2}$$

Where EPS is the minimal possible value, C_{puct} is a hyperparameter that controls the balance between exploration and exploitation. A higher C_{puct} results in a higher exploration behavior and a lower C_{puct} instructs the MCTS to search more profits according to empirical experiences obtained from previous searches. $Q(s,a)$ is updated each time after one entire MCTS simulation is over. The $Q(s,a)$ value is updated according to Equation **2.3**.

$$Q(s,a) = \frac{N(s,a) * Q(s,a) + v}{1+N(s,a)} \tag{2.3}$$

Here, v is the final result backpropagated from the end of the game to the current player after each game of MCTS simulation is finished. If $Q(s,a)$ is not initialized yet, then it is simply assigned the value v.

In each simulation, the MCTS search tree starts from the root node and continues its search so long as there are edges connecting the current node to the next node according to the selection of the highest UCB value. The search terminates once the arrived new node is not in the existing search tree and the CNN evaluates on this state, yielding the predicted prior probability distribution $\overrightarrow{p_\theta}(s)$ and predicted result value v at the state. The result value v is backpropagated towards the root and all Q-values that are visited during the search are updated as long as the search terminates, either by evaluation of the CNN, or by reaching the end of the game.

Parameter (τ) is a temperature parameter controlling the MCTS search. The $N(s,a)$ value is expected to give a good policy approximation after a number of MCTS simulations. To utilize the information provided by $N(s,a)$ more efficiently, $N(s,a)^{\frac{1}{\tau}}$ instead of $N(s,a)$ is used to generate the final target policy vector $\overrightarrow{\pi}$ of the tree search. If τ is 0, then the search algorithm simply selects the action with the highest $N(s,a)$ value and sets probabilities of other actions as 0.

The full algorithm of MCTS is shown in Algorithm **1**.

Algorithm 1: Bseline MCTS

```
Function MCTS(nnet, GameState, parameters):
    Init: $Qsa \leftarrow$ empty, $Nsa \leftarrow$ empty, $Ps \leftarrow$ empty, $Ns \leftarrow$ empty
    while $i <$ parameters.numMCTSsims do
        while not GameState.GameEnd do
            $s \leftarrow GameState.board$
            if  s not in Ns then
                $Ps[s],\ v \leftarrow$
                  predictNet($nnet$,$GameState.featureBoard$)
                $Ns[s] \leftarrow 0$, Break
            end
            for a in GameState.ValidMoves do
                if (s,a) in Qsa then
                    $u \leftarrow Qsa[(s,a)]+parameters.Cpuct*$
                      $Ps[s][a]*sqrt(Ns[s]/(1+Nsa[(s,a)]))$
                else
                    $u \leftarrow parameters.Cpuct*Ps[s][a]*$
                      $sqrt(Ns[s]+EPS)$
                end
                if $u >$ curBest then
                    $curBest \leftarrow u, bestAction \leftarrow a$
                end
            end
            $GameState \leftarrow$ NextBoard($bestAction$)
        end
        for v,a,s in GameState.steps do
            if (s,a) in Qsa then
                $Qsa[(s,a)]) \leftarrow$
                  $(Nsa[(s,a)]*Qsa[(s,a)]+v)/(Nsa[(s,a)]+1)$
                  $Nsa[(s,a)] \leftarrow Nsa[(s,a)]+1$
            else
                $Qsa[(s,a)]) \leftarrow v, Nsa[(s,a)] \leftarrow 1$
            end
            $Ns[s] \leftarrow Ns[s]+1$
        end
    end
    if parameters.temp equals 0 then
        return $probs[argmax_a(Nsa)] \leftarrow 1$
    else
        return $probs \leftarrow sum_a(Nsa)^{1/temp}/sum(Nsa))$
    end
```

2.2. TRAINING FLOW

The hyperparameter *iterations* controls the number of training iterations. During each iteration, the self-play procedure is executed the number of *episodes* times. For each episode, the game is initialized and the player selects the action according to the policy vector returned by MCTS. *tempThreshold* is a hyperparameter that controls the τ value, by setting τ to zero if the current game step exceeds *tempThreshold.* This entails that a higher *tempThreshold* value can result in more opening patterns and a lower *tempThreshold* decreases the divergence of opening selections.

After each game, the result value v is assigned to each player together with its corresponding policy vector $\vec{\pi}$ at each board state s. v is in the range of [-1,1], where the sign of v is reversed if its corresponding player changes as a win of one player is a loss of the other. Those examples are appended to the total history examples for the later training of the CNN. If the total number of history examples length exceeds the $retrainLength$, then examples of the first iteration entry are popped out. The examples are shuffled before fed into the CNN as input. The current neural network is saved for backup before the training. The new CNN will be recognized if it can beat the previous neural network by a percentage of $updateThreshold$ in $arenaCompare$ games. Otherwise, the previous neural network will be restored and the next iteration begins.

The performances of algorithms are evaluated using Elo scores that give information about the relative skill levels of players in a zero-sum game [24]. The Elo scores are calculated based on the current expectations of players winning next games. For example, If there are two players A and B, with Elo Scores S_A and S_B. Then the expectation of A winning the next games is calculated by Equation **2.4**. After playing one series of n games with i draws and j wins, the outcome R for A is calculated by Equation **2.5**. The new Elo score of A is updated by Equation **2.6**. K is a factor of maximal possible adjustment and is set to 32 for our experiment.

$$E_A = n * \frac{1}{1 + 10^{(S_B - S_A)/400}} \tag{2.4}$$

$$R_A = n * (0.5 * i + j) \tag{2.5}$$

$$S_A = S_A + K(R_A - E_A) \tag{2.6}$$

The flow of the simulation and network training is shown in Algorithm **2**.

Algorithm 2: Baseline Algorithm

```
Function Baseline(nnet, parameters):
    Init: $TrainExamples \leftarrow$ empty
    while $iteration < parameters.iterations$ do
        while $episode < parameters.episodes$ do
            $GameState \leftarrow$ InitialBoard()
            while not $GameState.GameEnd$ do
                $\vec{\pi} \leftarrow$ MCTS($nnet, GameState, parameters$)
                if $GameState.step < parameters.tempThreshold$ then
                    $action \sim \vec{\pi}$
                else
                    $action \leftarrow argmax_a \vec{\pi}$
                end
                $GameState \leftarrow$ NextBoard($action$)
            end
            for $step$ in $GameState.steps$ do
                $examples \leftarrow$ append $step.v, step.pi, step.s$
                $examples \leftarrow$ GetFlips($examples$)
            end
        end
        $TrainExamples \leftarrow$ append $examples$
        if length $TrainExamples > parameters.retrainLength$ then
            $TrainExamples \leftarrow$ remove first entry
        end
        $pnet \leftarrow nnet$
        $nnet \leftarrow$ trainNet($nnet, TrainExamples$)
        $pwins, nwins, draws \leftarrow$ PlayGames($pnet, nnet,$
          $parameters$)
        if $nwins/(pwins+nwins) < parameters.updateThreshold$
          then
            $nnet \leftarrow pnet$
        end
    end
    return $nnet$
```

Here, InitialBoard() is a function that initializes the game board and NextBoard() returns the next *GameState* given the action as input. Get-

Flips() flips the color of the board and rotates the policy matrix accordingly so as to create more examples. PlayGames() is a function in which the two players play games $parmeters.arenaCompare$ times and the function returns the game results as output.

2.3. NEURAL NETWORKS

An illustration of our CNN structure is shown in Figure **2.1**. ResNets are used to build the neural networks. The residual blocks of our CNN structure are borrowed from [25]. There are 10 residual blocks following the first convolutional layer, with the number of channels 64, 3×3 kernels with stride 1, and padding 1 used for those layers. The policy head consists of 1 convolutional layer with 2 output channels and 1 fully connected layer with action size of output features. This convolutional layer uses 1×1 kernel with stride 1, padding 0. The value head consists of 1 convolutional layer using the same 1×1 kernel with stride 1, padding 0, and 2 fully connected layers. There are 128 output features of the first fully connected layer and 1 single output feature of the second fully connected layer. ReLU [26] is used as activation function and all convolutional layers are with batch normalization [27]. The Adam optimizer is used for training the loss [28].

The policy loss is calculated by the entropy loss and the value loss is calculated by the mean square error. The total loss function is defined by the sum of the policy loss and the value loss as shown in Equation **2.7**.

$$l = (v_\theta(s_t) - z_t)^2 + \overrightarrow{\pi}_t log(\overrightarrow{p_\theta}(s_t)) \tag{2.7}$$

Here, s_t is the current board state, $v_\theta(s_t)$ and $p_\theta(s_t)$ are value and policy estimations given the current model parameter θ at the state s_t. z_t and $\overrightarrow{\pi_t}$ are the game result value and the improved policy vector returned by tree search for the player at the state s_t.

The inputs of the CNN are the board features fed into different channels at the state s, and the outputs are a policy vector $\overrightarrow{p_\theta}(s)$ and the result value v. The input of each channel is of size 10×5, with each channel representing the draughts board. The choices of those channels are shown in Table **2.1**. The first four channels are of binary numbers representing the men pieces and king pieces of both the black side and the white side. The fifth channel is also binary numbers representing the pieces that are al-

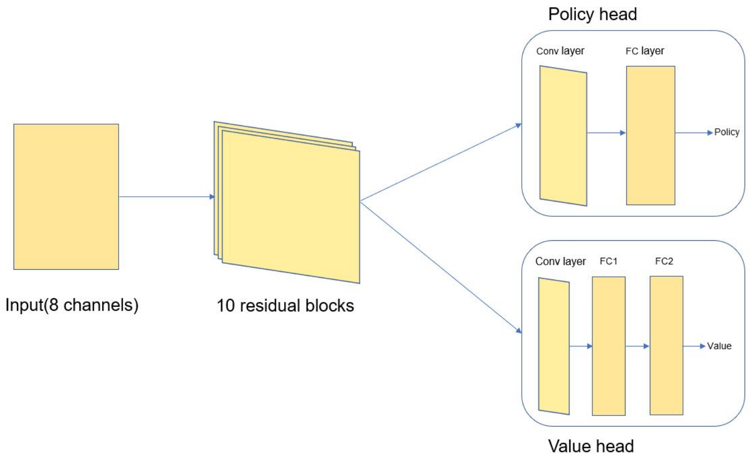

Figure 2.1: CNN structure

ready captured but have not yet been removed from the board. The sixth channel represents the current player, with the value 1 for player 1, or 0 for player -1. The seventh channel represents the total steps of the current game and the eighth channel is the number of steps of which there were no change in the counts of pieces for both players. Both the seventh channel and the eighth channel use non-negative integers as values.

Channels	Number	Value
men and kings	4	binary
captured pieces	1	binary
player	1	binary
total step	1	counts
no progress	1	counts

Table 2.1: Input channels for CNN

The action size returned by the CNN is $(10 \times 5)^2 + 1$, and each move is represented by its initial position on a board of size 10×5 and its new position as destination, no matter if it is a simple move or a jump. The moves of draughts are always distinct because the simple moves and jumps are of different board states, and there are no special rules to be taken care of, which otherwise may lead to ambiguities with actions under the same board state in other games, such as the underpromotion and the castling in chess. The extra 1 action denotes that no action is possible.

3 Three-stage Method

A previous research of training neural networks to play draughts [29] considered three kinds of input features of draughts, one is raw board positions, one is the structural features, the other is global features. Structural features are important in draughts, as the exchanges of pieces in certain setups become frequent. Global features are features that cannot be directly seen from the board positions, such as the difference between the numbers of pieces. Their test results have shown that the neural networks did not necessarily need raw board representations to achieve good performance when applying temporal difference learning methods. This is different with AlphaZero, where it takes the board channels as inputs and feeds them into the CNN. Possible structural features and global features also need to be formalized as channels of the board.

Their selection of features gives us inspirations about using heuristics of the draughts game itself to extract features and apply those features to different algorithms and models. As a result, the rules of draughts are utilized. A whole game of draughts can be divided into three stages. The first stage is defined as when there are more than 31 pieces and no kings on the board. The second stage is defined as when there are more than 8 pieces and no kings on the board. The third stage is recognized if there are less than or equal 8 pieces, or there are kings on the board. The differences between the first two stages and the third stage are straightforward, as there are initially no kings on the board and a piece only promotes to the king only after it reaches the other side of the board.

The action space is of size $(10 \times 5)^2 + 1$ for the baseline method, but the action size can be made much smaller for the stage 1 and stage 2 in the game of draughts. A smaller action space is selected for the stage 1 and stage 2, which is $10 \times 5 \times 4 + 1$. The detailed explanation about this choice of the action space will be described in Section **3.3**.

3.1. MCTS POLICY

The three-stage method made some changes in the MCTS policy based on the baseline method. The MCTS policy uses in total two different CNNs to generate a predicted prior probability $\overrightarrow{p_\theta}(s)$ and the result value v for stage 1, 2 and stage 3 separately. The $\overrightarrow{p_\theta}(s)$ returned by the CNN needs to be rescaled into the larger vector for the tree to use if it is stage 1 and stage 2. The differences between this three-stage algorithm and the baseline algorithm are shown in Algorithm **3**.

Algorithm 3: Three-stage MCTS

```
Function MCTS($nnet, GameState, parameters$):
    ...
    if s not in Ns then
        if stage equals 1 or stage equals 2 then
            $Ps[s],\ v \leftarrow$
              predictNet($nnet['s2'], GameState.featureBoard$)
            $Ps[s] \leftarrow$ InFlateProbs($Ps[s]$)
        else
            $Ps[s],\ v \leftarrow$
              predictNet($nnet['n1'], GameState.featureBoard$)
        end
        $Ns[s] \leftarrow 0$
        Break
    end
    ...
```

Here, $nnet['n1']$ is a CNN with the same structure as the baseline method that is used to predict the board state of stage 3, and $nnet['s1']$ is the smaller CNN that is used to predict on the board states of the stage 1 and stage 2. InFlateProbs() is a function that takes a probability vector of size $10 \times 5 \times 4 + 1$ and returns a probability vector of size $(10 \times 5)^2 + 1$. Each move in the small probability vector has a unique correspondence in the large vector given a certain board state.

3.2. TRAINING FLOW

In the training flow of the three-stage algorithm, the examples obtained from self-play need to be split into different stages. For the stage 1 and stage 2, the policy vector $\vec{\pi}$ in examples needs to be rescaled into the small action size. This step is necessary for the CNN to use the examples as inputs. Notably, all the examples are used to train the large CNN despite the fact that the CNN will only be used to predict on the board states of the stage 3, in order to gather more examples to train the neural network. Afterwards, examples for different stages are used to train the two networks separately. The differences between the training flow of the baseline algorithm and the three-stage algorithm are shown in Algorithm 4.

Algorithm 4: Three-stage Algorithm

```
Function Three-stage($nnet, parameters$):
    ...
    for step in GameState.steps do
        if step equals stage 1 or step equals stage 2 then
            $examples[1] \leftarrow$ append $step.v, step.pi, step.s$
            $step.pi \leftarrow CompressProbs(step.pi)$
            $examples[0] \leftarrow$ append $step.v, step.pi, step.s$
        else
            $examples[1] \leftarrow$ append $step.v, step.pi, step.s$
        end
    end
    $TrainExamples[0] \leftarrow$ append $GetFlips(examples[0])$
    $TrainExamples[1] \leftarrow$ append $GetFlips(examples[1])$
    if length TrainExamples > parameters.retrainLength then
        $TrainExamples \leftarrow$ remove first entry
    end
    $pnet \leftarrow nnet$
    $nnet['s2'] \leftarrow$ trainNet($nnet['s2'], TrainExamples[0]$)
    $nnet['n1'] \leftarrow$ trainNet($nnet['n1'], TrainExamples[1]$)
    ...
```

Here, CompressProbs() is a function that takes a probability vector of size $(10\times5)^2+1$ and returns a probability vector of size $10\times5\times4+1$. Each move in the large probability vector has a unique correspondence in the small vector given a certain board state and if there are no kings on the board.

3.3. NEURAL NETWORKS

The large CNN has the same structure as the baseline CNN. The small CNN used in the stage 1 and stage 2 has a similar structure but with 5 residual blocks instead of 10 blocks. Three are 64 instead of 128 output features in the first fully connected layer of the value head.

The action size returned by the policy head of the small CNN is $10 \times 5 \times 4 + 1$, each move is represented by its initial position on a board of size 10×5 and its 4 directions in which it takes a move or a jump. The extra 1 action indicates that no action is possible. There are no kings in the stage 1 and stage 2. The initial positions and directions for which men pieces move can sufficiently denote all unique actions because men pieces are not able to move more than one grid at one time in a simple move and men pieces can only jump to the next empty grid in the direction of capturing the piece of the opponent. Moreover, simple moves and jumps always have different board positions. Hence, the CNN is able to understand whether a move is a simple move or a jump by its input and $10 \times 5 \times 4 + 1$ is a sufficient number of the action size.

4 MULTIPLE POLICY VALUE METHOD

The Multiple Policy Value (MPV) method is a recently proposed algorithm to improve the MCTS tree search of AlphaZero [30]. The basic idea behind is to build a different MCTS tree T_s with a smaller CNN f_s and a larger number of MCTS simulations. Meanwhile, the other MCTS tree T_l is generated by the larger CNN f_l with a smaller number of MCTS simulations. T_l searches with the help of T_s, which serves as a guidance towards the moves to be searched by T_l at each state s.

The main advantage of MPV is to make the algorithm perform more MCTS simulations with the same amount of computational budget. Their results have shown that MPV has bested the original AlphaZero search algorithm when applied on NoGo and the two algorithms have the same amount of computational budget. This makes the algorithm interesting in our MCTS implementation on draughts since we have a limited computational budget and MPV can potentially improve the performance of our MCTS within the budget.

4.1. MCTS POLICY

The interactions between T_l and T_s are related to the asynchronous policy value MCTS [31], in which the rollout and the value network are combined for the MCTS to obtain better evaluations than by each strategy alone. To realize the idea of MPV that the T_s can facilitate the search of T_l and meanwhile T_l gives better predictions over the nodes T_s guides towards, the exchanges of the information between T_s and T_l during the execution of MCTS simulations need to be subtle. How the small tree T_s and the large tree T_l interact when the prior probabilities $P(s)$ and the result value $V(s)$ are predicted at the state s by both T_l and T_s are shown in Equation **4.1** and Equation **4.2**.

$$V(s) = \alpha V_{T_s}(s) + (1-\alpha) V_{T_l}(s) \tag{4.1}$$

$$P(s) = \beta P_{T_s}(s) + (1-\beta) P_{T_l}(s) \tag{4.2}$$

Here, α and β are constants that control the balance between f_l and f_s. They should be set smaller if the f_l to generate T_l is more reliable. Because f_l is the larger net that has more accurate predictions, the MPV method follows the settings of the asynchronous policy value MCTS and uses $\alpha = 0.5$ and $\beta = 0$.

T_l needs to expand the tree following the guidance of T_s, and meanwhile it has to share more reliable predictions with T_s through the above equations. Hence, the selection of the action during the expansion of T_l prioritizes the nodes that are mostly visited by T_s, to help T_s correct the predictions of $P(s)$ and $V(s)$ at each state s. If the selected search tree is T_s, or if the selected search tree is T_l but the current node is not expanded by T_s yet, the algorithm selects the best action using the UCB criteria that were shown in Equation **2.1** and Equation **2.2**.

The number of MCTS simulations for T_l is given by $numMCTSsims$. We have an extra parameter $budget$ to control the number of MCTS simulations that are used to generate T_s, which is $numMCTSsims * budget$. The total amount of MCTS simulations is $numMCTSsims * (budget + 1)$. Before an MCTS simulation starts, we decide whether to expand T_s or to expand T_l. Our approach is different with the method used in the original MPV, where the expansion is on a randomized basis. Randomly choosing the tree out of T_s and T_l could be problematic when we have a small number of $numMCTSsims$, because during the expansion of T_l, it might have scarce knowledge about what nodes T_s would like to visit most frequently. If the simulations of T_l run at the start or at the end of all the simulations, both trees will lose the opportunity to optimize the search through shared prior probabilities and result values. Thus, we utilize the current step of iterations as an index so that we can insert the expansions of T_l evenly across the total number of simulations.

The returned probability vector $\vec{\pi}$ is generated using the visit counts over $N(s,a)$ in T_s, as the number of visit counts in T_s is larger and T_l has already shared its more reliable predictions with T_s by Equation **4.1** and Equation **4.2**.

The differences between the MPV algorithm and the baseline algorithm are shown in Algorithm **5**.

Algorithm 5: MPV MCTS

```
Function MCTS(nnet, GameState, parameters):
    ...
    while i < parameters.numMCTSsims*(parameters.budget+1) do
        $Tree \leftarrow i \% parameters.numMCTSsims == 0 ? T_l : T_s$
        ...
        if s not in Ns[Tree] then
            $Ps[Tree][s], Vs[Tree][s] \leftarrow$
              predictNet($nnet[Tree]$, $GameState.featureBoard$)
            if s in Ns[$T_s$] and s in Ns[$T_l$] then
                $Vs[T_l][s] \leftarrow 0.5 * Vs[T_l][s] + 0.5 * Vs[T_s][s]$
                $Vs[T_s][s] \leftarrow 0.5 * Vs[T_l][s] + 0.5 * Vs[T_s][s]$
                $Ps[T_s][s] \leftarrow Ps[T_l][s]$
            end
            ...
        end
        ...
        for a in GameState.ValidMoves do
            if Tree is $T_l$ and (s,a) in Nsa[$T_s$] then
                $u \leftarrow Nsa[T_s][(s,a)]$
            else
                ...
            end
        end
    end
    if parameters.temp equals 0 then
        return $probs[argmax_a(Nsa[T_s])] \leftarrow 1$
    else
        return $probs \leftarrow sum_a(Nsa[T_s])^{1/temp} / sum(Nsa[T_s]))$
    end
```

4.2. TRAINING FLOW

The training flow in MPV is basically the same as in the baseline method, except that an extra neural network $f(s)$ is initialized and is trained using the same examples as $f(l)$.

4.3. NEURAL NETWORKS

The structure of $f(l)$ is the same as the CNN in the baseline method. $f(s)$ has a similar structure with $f(l)$ but is smaller. The body of $f(s)$ only has 5 residual blocks instead of 10 blocks. The first fully connected layer of the value head has 64 output features instead of the 128 output features as in $f(l)$.

5 Hybrid method of Three-stages and MPV

A hybrid approach of three-stage and MPV methods is designed as an extra variation of the MCTS algorithm. Our particular interest is in whether the hybrid method gives better performance than either of the combined methods alone, namely, whether the hybrid method can outperform the three-stage method and whether the hybrid method can outperform the MPV method.

5.1. MCTS Policy

In case of the hybrid method, a slightly different MCTS search policy is used. For the stage 1 and stage 2 of the game, we use a smaller network to predict the prior probability distributions $\overrightarrow{p_\theta}(s)$ and the result value $V(s)$ as in the three-stage method. The predictions are directly assigned to both trees T_s and T_l. In the stage 3 of the game, however, $\overrightarrow{p_\theta}(s)$ and $V(s)$ are predicted separately for each of the trees. Those predictions in stage 3 are shared using Equation **4.1** and Equation **4.2**, as in the MPV algorithm.

The rational behind this implementation is that the smaller neural network used for the stage 1 and stage 2 in the three-stage method is already significantly small for both T_s and T_l, and thus there is no need to add an extra smaller network for the small tree. On the other hand, it is necessary, in stage 3, to create a different CNN so that the smaller CNN is used to predict on T_s and the larger CNN is used to predict on T_l.

The highlights of the hybrid method are shown in Algorithm **6**.

Algorithm 6: Hybrid MCTS

```
Function MCTS(nnet, GameState, parameters):
    ...
    if s not in Ns then
        if stage equals 1 or stage equals 2 then
            $Ps[Tree][s], Vs[Tree][s] \leftarrow$
              predictNet($nnet['s2']$, $GameState.featureBoard$)
            $Ps[Tree][s] \leftarrow$ InFlateProbs($Ps[Tree][s]$)
            $Vs[T_l][s] \leftarrow Vs[Tree][s]$
            $Vs[T_s][s] \leftarrow Vs[Tree][s]$
            $Ps[T_l][s] \leftarrow Ps[Tree][s]$
            $Ps[T_s][s] \leftarrow Ps[Tree][s]$
        else
            $Ps[Tree][s], Vs[Tree][s] \leftarrow$
              predictNet($nnet[Tree]$, $GameState.featureBoard$)
        end
        ...
    end
    ...
```

5.2. TRAINING FLOW

The training flow in the hybrid method is similar to that of the three-stage method, except for that there is an extra small CNN initialized and trained by the same examples for the large CNN.

5.3. NEURAL NETWORKS

There are in total three neural networks used in this method. The first two are a large CNN and a small CNN with the same structures as in the three-stage method. The third is a small CNN that has the same structure as the small CNN in the MPV method.

6 GAME IMPLEMENTATION

We implemented our game of draughts according to the draughts rules, but also made some small adjustments for the purpose of smoothed MCTS tree search.

6.1. REGULAR PROCEDURE

Referring to Algorithm 1 and 2, the $GameState$ is initialized by Initial-Board() at the start of each game. All the valid moves are shown to the player by a vector $GameState.ValidMoves$. $ValidMoves$ is given by searching all the moves that are possible for each movable piece. Whether the piece needs to make mandatory consecutive jumps or not is also to be tested. After the player chooses an action, NextBoard() returns the next $GameState$ and we check whether the game has reached its end using $GameState.GameEnd$. The game continues if the $GameEnd$ is not giving a value indicating the end. Otherwise, $GameEnd$ gives the result value of the game and the game ends.

6.2. MCTS SPECIFICATIONS

For the implementation of MCTS, each board is represented by the feature board $GameState.featureBoard$, which includes 8 channels of different features. During the proceedings of the game, necessary values for each state are stored in $GameState.steps$, which includes the action a, the state with current player s, and the backpropagated result value v from the end of the game. Those values are used to update the Q-values after each MCTS simulation finishes. In the training flow, the policy vector $\vec{\pi}$ is recorded instead of the action a and those values are used to generate example files for training.

During the game steps and especially those steps with consecutive jumps, it happens that there is only one valid move possible in the vector $GameState.ValidMoves$ for the player to take at that state. There is no reason to search those states and to use the CNN to predict a prior probability distribution. Thus, those states are directly omitted in the MCTS search and the player will automatically take the only possible move. The training files also do not include the examples from those states. This speeds up both the training and the tree search. However, this implementation also introduces a problem that the CNN will not be able to return a result value prediction v at those states when the new leaf expands. To prevent the case that the update of Q-values has no game result, the $GameEnd$ value is used as a heuristic for the game result.

We have set the maximal game steps to 150, as a typical draughts game would not exceed this number. A search depth limitation of MCTS is set to 50, as such deep search trees given our small number of $numMCTSsims$ indicate repetitions of game states. The number of steps where there is no progress for both players is also used to set a limitation. The game ends if this number is larger than 20. Those specifications adapt slightly different draw conditions compared with the draughts rules. The values of those game endings indicate draws for game plays in the arena, but not for games in MCTS simulations and training episodes. Those values are used for the tree search and to generate example files in order to search and train faster.

The values of $GameEnd$ are in the range of [-1, 1], -1 indicates the winning of player -1 and 1 indicating the winning of player 1. Any value other than -1 and 1 indicates a draw. For all the situations that the game does not play to its end but was discontinued, a heuristic value is assigned to $GameEnd$ according to Equation **6.1**.

$$v = \frac{\sum men * player + \sum kings * player + \sum captured}{numPieces} \tag{6.1}$$

Here, $player$ is a value of either 1 for the current player or -1 for the opponent player, men is the value of men pieces, which is 1, $king$ is the value of king pieces, which is 2, and $captured$ is the value representing the captured pieces yet not removed from the board, which is 1. The value of king pieces is also possible to set to 3 in some other practices. $captured$ indicates that the current player is on a consecutive jump, which will certainly have advantages at the current state. $numPieces$ is the total number of pieces, which is 40 for 10 × 10 draughts.

6.3. ALPHA-BETA PLAYER

A random player and an Alpha-Beta player have been built to play against the models [6]. The Alpha-Beta algorithm is shown in Algorithm **7**.

Algorithm 7: Alpha-Beta Algorithm

```
Function Alpha-Beta($GameState, depth, \alpha, \beta, player$):
    if GameState.GameEnd not 0 or depth equals 0 then
        return GameState.GameEnd, -1
    end
    for move in GameState.ValidMoves do
        $GameState \leftarrow$ NextBoard(move)
        if GameState.Player not equals player then
            $newDepth \leftarrow depth - 1$
        end
        $temp \leftarrow$ Alpha-Beta($GameState, newDepth, \alpha, \beta$
         $GameState.Player$)[0]
        if player equals 1 and temp > $\alpha$ then
            $bestAct \leftarrow move, \alpha \leftarrow temp$
        end
        if player equals -1 and temp < $\beta$ then
            $bestAct \leftarrow move, \beta \leftarrow temp$
        end
    end
    if player equals 1 then
        return $\alpha$, bestAct
    end
    if player equals -1 then
        return $\beta$, bestAct
    end
```

The basic idea of the Alpha-Beta algorithm is to trim the branches which are not searched but it is known that there are no optimal values in those branches according to the limit constraints of value α and β.

Due to the special rules of draughts, the definition of depth 1 here is not per move, but a combination of moves. The number of depth changes only if the $player$ switches. The depth will not change if the player is performing consecutive jumps.

The evaluation function of the Alpha-Beta algorithm also uses the heuristics from Equation **6.1** because those heuristics are based on the current situation of the pieces and give a good approximation of the evaluation.

7
EXPERIMENT SETUPS

As the training of MCTS with deep neural networks is time-consuming, a balance must be made between the performance of the model and the time cost of training. The authors of [32] have experimented on the influences of those parameters and categorized them as time-sensitive or time-friendly parameters. The performance of MCTS is related to time-sensitive parameters heavily and the performance can be worse if we set time-sensitive parameters lower. On the other hand, setting time-friendly parameters lower does not significantly influence the performance of MCTS. Hence, it was best of our interests to set time-friendly parameters lower when we tried to determine the hyperparameters for our models. Considerations of trade-offs between the time cost and the performance were also made when tuning hyperparameters that are time-sensitive.

7.1. TRAINING PARAMETERS

The settings of training hyperparameters for the training process and MCTS are shown in Table **7.1**. Those parameters are:

tempThreshold sets a threshold and if the game step is larger than tempThreshold, it will make τ zero. which in turn makes the model perform the search only on moves with the largest visiting count. We set the value of tempThreshold to 10 as it is a time-friendly parameter. However, low tempThreshold might lead to less opening patterns for the examples.

Cpuct controls the calculation of UCB when using Equation **2.1** and

Equation **2.2**. A large value of Cpuct favors more exploration over exploitation. It is also a time-friendly parameter and we set it to 1.

numMCTSsims is the parameter that controls the number of tree searches and thereby affects the depth of tree searches. It is time-sensitive and is crucial to improve the policy $\vec{\pi}$. A large numMCTSsims is typically needed for a game with a large action space. The numMCTSsims for the baseline and MPV algorithms are set to 50, which might not be sufficient for the tree search but higher numMCTSsims makes the time cost not affordable.

budget is the parameter that controls the numMCTSsims of T_s for the MPV and hybrid methods. A budget value of 8 gives the best performance according to [30]. In order to make the baseline and MPV algorithms have similar time cost, we set numMCTSsims of T_l to 10 and budget to 8, so it has a total of $10 + 10 * 8$.

retrainLength is the total number of history files that is kept for training. This parameter is time-sensitive and a larger retrainLength value can lead to more stabilized training with less fluctuations. However, setting retrainLength low can accelerate learning speed. retrainLength is set to 5 for all algorithms.

arenaCompare is the number of times the new CNN and the previous CNN play against each other in order to determine which side is more favorable. This parameter is time-sensitive and can usually be set smaller if updateThreshold is large. arenaCompare is set to 20 for our experiment.

updateThreshold is the win rate threshold of updating a new CNN. This parameter is required to be larger than 0.5 so it can help avoid using a worse CNN. The parameter updateThreshold is set to 0.55 for our experiment.

episodes is the total number of games the rollout plays to generate training examples in each iteration. This number is time-sensitive and is set to 100 for our experiment.

iterations is the total number of steps. For each step, the rollout runs $episodes$ times and then the CNNs are trained. This number is not fixed for our experiment as we run as much as it can. Thus, the numbers of iterations might also be different for the four algorithms.

Parameter	Type	Value
tempThreshold	time-friendly	10
Cpuct	time-friendly	1
numMCTSsims	time-sensitive	50
budget	time-sensitive	8
retrainLength	time-sensitive	5
arenaCompare	time-sensitive	20
updateThreshold	time-sensitive	0.55
episodes	time-sensitive	100
iterations	time-sensitive	-

Table 7.1: Training parameters of MCTS methods

7.2. CNN PARAMETERS

The settings of hyperparameters for the CNN are shown in Table **7.2**. Those parameters are:

learningRate: The learning rate of the model. A smaller learning rate is typically desired to avoid falling into a local optima. Meanwhile, the number should not be too small so as to let the model learn slow. This parameter is time-friendly and we set it to 0.001.

dropout is a parameter that controls the percentage of randomly dropping some of the weights during the training to prevent overfitting [33]. This parameter is time-friendly and is set to 0.3.

weightDecay is a parameter together with dropout to obtain better generalization of the model [34]. This parameter is time-friendly and is set to 0.0001.

epochs is the number of training epochs of the CNN. A large number of training epochs can lead to overfitting yet a small training epochs can lead to underfitting. This parameter is time-sensitive and we set it to 10.

batchSize is the batch size during the training. Larger batchSize typically decreases the performance of CNN. Meanwhile, a smaller batch size increases the number of batches and hence results in slower training. batchSize is time-sensitive and is set to 64.

numChannels is the number of channels outputted by each convolutional layer. This number is time-sensitive and is set to 64.

Parameter	Type	Value
learningRate	time-friendly	0.001
dropout	time-friendly	0.3
weightDecay	time-friendly	0.0001
epochs	time-sensitive	10
batchSize	time-sensitive	64
numChannels	time-sensitive	64

Table 7.2: CNN parameters of MCTS methods

8 RESULTS

To compare different algorithms in our experiments, all algorithms are tested against the random player and the Alpha-Beta player. Three-stages, MPV, and hybrid methods also need to play against the baseline algorithm separately in order to make a straightforward illustration of their strengths. During each comparison of games, two players play against each other for a total amount of 20 games and Elo scores are calculated by Equation **2.4**, **2.5**, and **2.6**. All players start the comparison with the same Elo scores of 0 and the search depth of the Alpha-Beta player is set to 2. All models are trained with similar time cost for each training iteration except for the hybrid method. The hybrid method takes more time cost because it has more networks to train than either method alone. Only neural networks that are updated by exceeding $updateThreshold$ are included in the arena comparisons. Notably, most results only include at most 50 iterations of training, as only the training iterations of the baseline method and the MPV method exceed this limitation and more training iterations do not provide obvious improvements to their performances.

8.1. TRAINING LOSSES

The training losses of four methods in the initial 50 iterations are shown in Figure **8.1**, **8.2**, **8.3**, and **8.4**. Losses of all training iterations are recorded, including iterations where the new CNNs do not update by not winning more than the percentage of $updateThreshold$.

It is clear that the large CNN and the small CNN with the same action size $(10\times5)^2+1$ have similar loss curves. Meanwhile, the small CNN with the smaller action size $10\times5\times4+1$ has a lower curve due to smaller losses in policy vectors, but with the same trends.

Losses from all methods decrease sharply at the beginning, and their decreasing speed decelerates over the training period. The MPV method has the slowest loss decreasing speed, which might be caused by its reliance on accurate predictions of the large CNN while the large CNN is randomly initialized at the beginning. The hybrid method has a faster loss decreasing speed than the MPV method, as the first and second stages of the hybrid method in the game do not use the coordination between two trees and hence rely less on the strength of the large CNN.

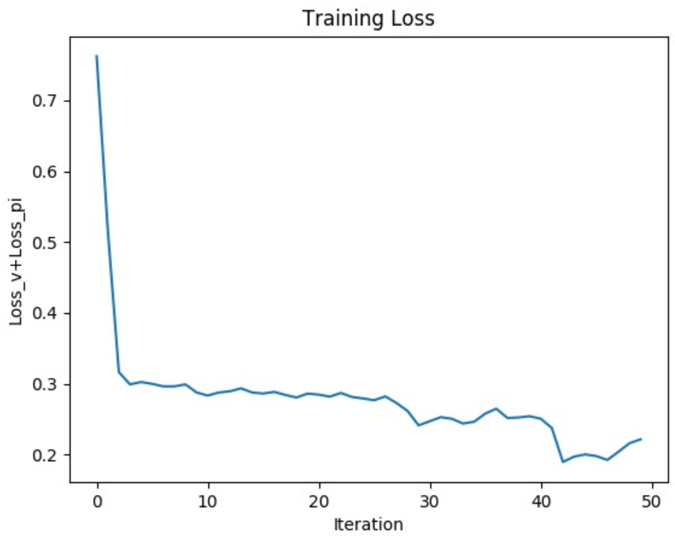

Figure 8.1: Training loss of the baseline method

8.2. AGAINST RANDOM AND ALPHA-BETA PLAYERS

The results of MCTS algorithms playing against the random and Alpha-Beta players are shown in Figure **8.5** and **8.6**. We can see that all MCTS algorithms are capable of beating the random player at the beginning of training. Elo scores of baseline and MPV algorithms continue rising to an

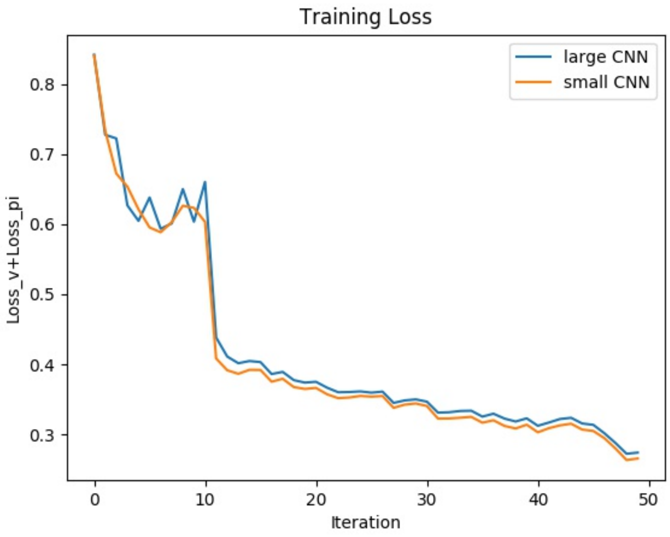

Figure 8.2: Training loss of the MPV method

amount where winning all 20 games against the random player only gives a marginal boost on Elo scores, due to the estimation calculated by Equation **2.4**. The hybrid algorithm, with slower training speed, gives promising prospects to perform as well as those two algorithms during the time of training. On the other hand, the three-stage algorithm alone does not keep increasing the Elo scores as fast as the other three, but maintains a relatively stable score at around 300.

While playing against the Alpha-Beta player, all MCTS algorithms start with losing to the Alpha-Beta player significantly and with Elo scores around -300. The baseline player then keeps losing to the Alpha-Beta player until it reaches the lowest Elo score around -500. Afterwards, the performance of the baseline player increases and reaches the peak around -150 at iteration 20. Meanwhile, this performance is not increasing any more with more iterations and fluctuates around -250.

The MPV method follows a similar curve shape to the baseline method, but is able to beat the Alpha-Beta player after 27 iterations, with a margin around 100. Afterwards, there is a dip in performance and the Elo scores are not able to further improve, and they finalize around -100. This result is better than the baseline since it has a higher finalized score and can

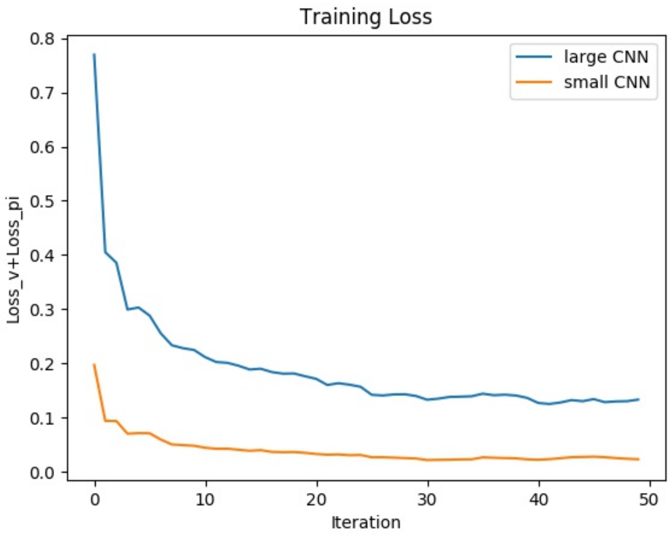

Figure 8.3: Training loss of the three-stage method

beat the Alpha-Beta player during training.

The hybrid method shows a similar pattern to the MVP method, yet the three-stage method gives much worse performances and shows no sign of beating the Alpha-Beta player. The hybrid method and the three-stage method also require a much larger number of iterations to train a new network that has the a winning rate passing *updateThreshold*, which in turn yields less iterations than the baseline and MPV methods in those plots.

The reasons that those algorithms are capable of beating the random player but not beating the Alpha-Beta player are possibly due to the insufficient amount of training, as we set some time-sensitive hyperparameters to small numbers due to the limitation of computational power. This is particularly true for the size of the neural networks and the parameter *numMCTSsims*, which are prominent time-sensitive parameters that have influences towards the strength of MCTS algorithms.

In our algorithms, the structure of large neural networks are typically with 10 residual blocks. 10 residual blocks are considered to be sufficient for most of the games, while the usage of small CNNs with 5 residual blocks in MPV and three-stage algorithms yields quite different results.

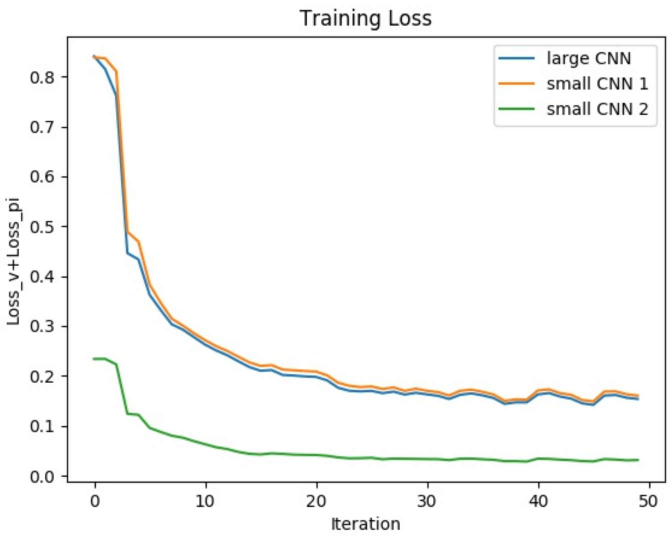

Figure 8.4: Training loss of the hybrid method

At the same time, our baseline method has a *numMCTSsims* of 50 and this amount is considered insufficient for most games with a large number of actions. The small *tempThreshold* of 10 might enlarge this disadvantage given by *numMCTSsims*, since it will generate less opening patterns.

It is notable that the input channels of CNNs include board representations from the previous 7 time steps in [3], while only one current board is fed into the CNNs in our experiment, in order to train and run CNNs faster. This might be another reason why our algorithms cannot stably beat the Alpha-Beta algorithm with depth 2.

However, it is also crucial that the performance of our Alpha-Beta algorithm in draughts with depth 2 can be much more powerful than in other games with the same depth. Two combinations of moves could easily unfold more than five jumps in draughts, whereas the search in MCTS needs to stop whenever a new leaf expands, no matter if this is during a consecutive jump or not, even if with only 1 possible current move. This gives the asymmetries between the depth in the Alpha-Beta algorithm and the *numMCTSsims* in MCTS algorithms. In general, the real search depth of Alpha-Beta player with depth 2 can be deeper, though unstable,

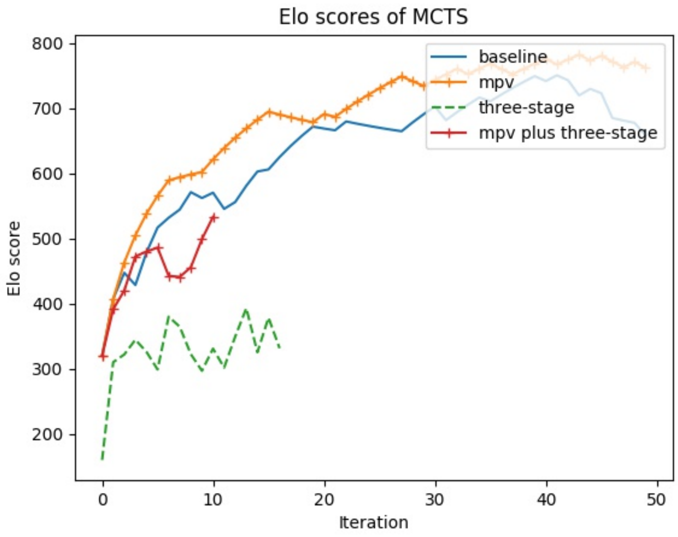

Figure 8.5: Elo scores against the random player

than the deepest expanded leaves of MCTS algorithms with $numMCTSsims$ 50.

8.3. COMPARISONS BETWEEN MCTS MODELS

In order to make straightforward comparisons between four MCTS models, the baseline model is made to play against the other three models separately. The results are shown in Figure **8.7**. We can see that the MPV method starts the comparison with the highest initial score around 200 yet this margin is not increasing stably during the whole training period, and there is a trend to make this margin slightly lower. Meanwhile, the hybrid method starts with an initial score 100 and the trend of decreasing this score is more significant than that of MPV. After certain iterations, the hybrid player is not capable of winning against the baseline player anymore. On the other hand, the three-stage method loses to the baseline method since the beginning and the Elo score keeps decreasing during the period of training, which indicates a worse performance.

As a result, the MPV method gives the best performance out of all models, and the three-stage method gives the worst performance. The

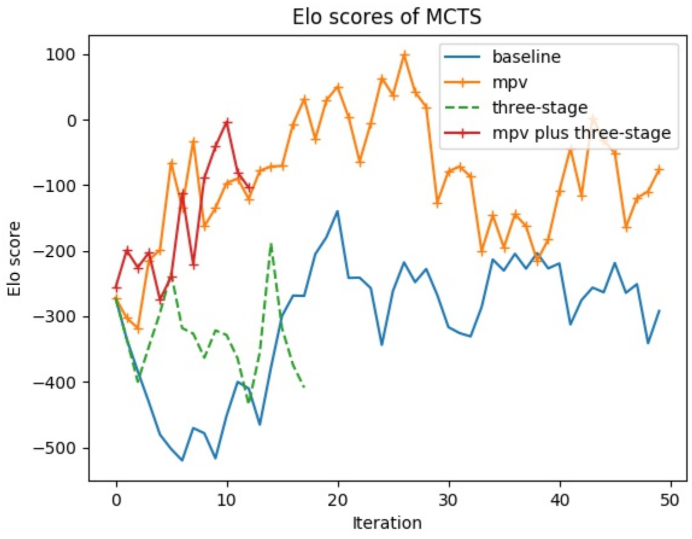

Figure 8.6: Elo scores against the Alpha-Beta player

three-stage method and the hybrid method also take more actual iterations, although not visible from the figure, to make an update of CNNs. The hybrid method gives a better performance than the baseline method at the beginning, but soon the performance becomes worse. The reason why this happens for the hybrid method might be that it is a combination of the good performance from the MPV method and the bad performance from the three-stage method.

The MPV method utilizes a smaller CNN f_s with 5 residual blocks to generate a tree T_s with more $numMCTSsims$. The $numMCTSsims$ for T_s and T_l in MPV are 80 and 10, respectively. The distinct MCTS policy not only gives it advantages during training, but also gives it higher starting Elo scores. On the other hand, the three-stage method also uses a smaller CNN f_s with 5 residual blocks, but for the training of stage 1 and stage 2, where the game has a smaller action space. The results in Figure **8.7**, hence, entail that the larger number of $numMCTSsims$ for T_s and the coordination between two MCTS trees could overcome the side-effects of the less prediction strength caused by smaller neural networks, whereas the usage of domain-specific heuristics in the three-stage method, unfortunately, cannot offset the effects of less prediction strength caused by

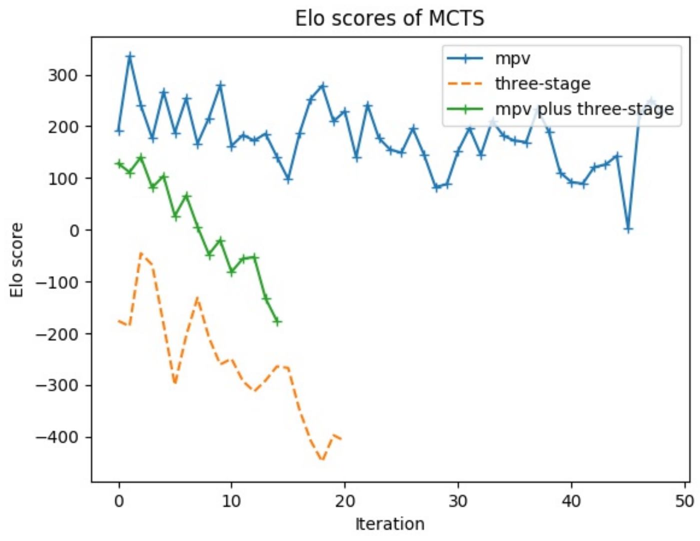

Figure 8.7: Elo scores against the baseline player

the smaller CNN. The shortage of training examples for the small neural networks might also play a role in the less capable performance in the three-stage method.

Nonetheless, our main hypothesis is that the three-stage method should be able to reach a better performance by using a large neural network with 10 residual blocks for the stage 1 and stage 2 in the game of draughts. This should be tested in future work.

9
CONCLUSION

9.1. CONCLUSION

The conclusion of our experiment is that MCTS methods with CNNs can indeed perform well on the game of draughts when playing against random players after a few training iterations. Meanwhile, the MPV method that utilizes two search trees performs the best out of all tested variations of MCTS methods. MPV plays almost as well as traditional prevailing methods, such as the Alph-Beta search.

On the other hand, the three-stage method does not show improved results, possibly due to the insufficient small CNN of the stage 1 and stage 2. The hybrid method of three-stage and MPV, shows more strength than the three-stage method alone but is not significantly superior than the MPV method. The domain-specific heuristics have shown less strength against the multiple search trees coordinating their policies and values.

9.2. DISCUSSION

There are also other possible explanations for our algorithms not stably besting the Alpha-Beta player, except for the mentioned assumptions such as $numMCTSsims$ and the size of CNNs. The authors of AlphaZero state that the training time and results can both significantly improve by exploiting symmetries of Go [35]. However, the positions of the board in draughts are not rotationally and reflectionally invariant, which indicates that the board of draughts is asymmetrical. The only thing we can do about examples without changing results in draughts is to switch the color of players and rotate the board accordingly. The number of examples in draughts is only one eighth that of games with symmetries. Meanwhile, our hyperparameter *episodes* was only set to 100 due to limited computational budget, which might result in worse performance and convergence for our algorithms.

Our experiment with different variations of the Monte Carlo tree search algorithm on draughts broads the understanding towards both the field of draughts and the algorithm itself, especially in terms of the connections among multiple neural networks, search trees, and draughts heuristics. Meanwhile, it is clear that MCTS is capable of learning how to play draughts and the MPV method has significantly improved the performance of MCTS with a similar time budget. The research results have added more material to the choices among different variations of MCTS methods. The research also contributes to the generalization of MCTS and the development of RL as a whole.

9.3. FUTURE WORK

There are criticisms about that AlphaZero cannot explain itself even though it can play those great moves in the game of Go and Shogi [36]. There are also questions about what humans could learn from such powerful AI and from its plays. Thus, managing to explain MCTS with CNN methods in more mathematical ways and coaching human playing those games are considered important directions in future research.

Meanwhile, current MCTS algorithms have limited generalization abilities towards common decision tasks. Those algorithms cannot handle a realistic scenario where there are a large number of decision examples but not a well-implemented simulator engine. The current methods of solving this scenario are often to use human knowledge to find Bayesian rules from empirical experience, and to build a simulator based on those extracted rules. This means that the quality of MCTS models might heavily depend on the soundness of a simulator if MCTS models are not playing a well-defined game.

CNNs have shown their strength with end-to-end learning in object detection tasks. It might also be possible to make the training of MCTS an end-to-end process since there are plenty of similarities in CNNs between RL and image recognition/object detection. For example, when the Regions with CNN features method (R-CNN) was initially proposed, the Selective Search method and Support Vector Machines (SVMs) had to be used together with the CNN in order to perform an object detection task [37]. However, Faster R-CNN was invented soon and makes a complex objection task an end-to-end process by using the Region Proposal Network (RPN) and ROI Pooler [38]. This gives us inspirations to make MCTS an end-to-end process, which can be a creative future work in the field of RL.

REFERENCES

[1] S. Legg, M. Hutter, Universal intelligence: A definition of machine intelligence, Minds and machines 17 (2007) 391–444.

[2] T. Schaul, J. Togelius, J. Schmidhuber, Measuring intelligence through games, arXiv preprint arXiv:1109.1314 (2011).

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al., A general reinforcement learning algorithm that masters chess, shogi, and go through self-play, Science 362 (2018) 1140–1144.

[4] lidraughts, lidraughts, `http://lidraughts.org`, 2018.

[5] K. Chang, Computer checkers program is invincible, The New York Times (2007).

[6] T. A. Marsland, Computer chess methods, Encyclopedia of Artificial Intelligence 1 (1987) 159–171.

[7] E. A. Heinz, Extended futility pruning, ICGA Journal 21 (1998) 75–83.

[8] M. Campbell, A. J. Hoane Jr, F.-h. Hsu, Deep blue, Artificial intelligence 134 (2002) 57–83.

[9] J.-J. van Horssen, Schwarzman vs. maximus: A man-machine match in international draughts, ICGA Journal 35 (2012) 106–119.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, Nature 518 (2015) 529.

[11] X. Guo, S. Singh, H. Lee, R. L. Lewis, X. Wang, Deep learning for real-time atari game play using offline monte-carlo tree search planning, in: Advances in neural information processing systems, 2014, pp. 3338–3346.

[12] M. A. Wiering, Self-play and using an expert to learn to play backgammon with temporal difference learning., JILSA 2 (2010) 57–68.

[13] M. Van Der Ree, 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), IEEE, 2013, pp. 108–115.

[14] P. Auer, N. Cesa-Bianchi, P. Fischer, Finite-time analysis of the multi-armed bandit problem, Machine learning 47 (2002) 235–256.

[15] D. A. Berry, B. Fristedt, Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability), London: Chapman and Hall 5 (1985) 71–87.

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

[17] M. Enzenberger, M. Muller, B. Arneson, R. Segal, Fuego—an open-source framework for board games and go engine based on monte carlo tree search, IEEE Transactions on Computational Intelligence and AI in Games 2 (2010) 259–270.

[18] C. D. Rosin, Multi-armed bandits with episode context, Annals of Mathematics and Artificial Intelligence 61 (2011) 203–230.

[19] R. Coulom, Computing "elo ratings" of move patterns in the game of go, Icga Journal 30 (2007) 198–208.

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

[21] S. Thakoor, S. Nair, M. Jhunjhunwala, Alphazero general, `http://github.com/suragnair/alpha-zero-general`, 2017.

[22] S. Thakoor, S. Nair, M. Jhunjhunwala, Learning to play othello without human knowledge, 2017.

[23] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of monte carlo tree search methods, IEEE Transactions on Computational Intelligence and AI in games 4 (2012) 1–43.

[24] A. E. Elo, The rating of chessplayers, past and present, Arco Pub., 1978.

[25] Z. T. Sun Haozhe, Alphadraughts-zero, `http://github.com/Tong-ZHAO/AlphaDraughts-Zero`, 2018.

[26] X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks, in: Proceedings of the fourteenth international conference on artificial intelligence and statistics, 2011, pp. 315–323.

[27] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, arXiv preprint arXiv:1502.03167 (2015).

[28] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).

[29] J. P. Patist, M. Wiering, Learning to play draughts using temporal difference learning with neural networks and databases, in: Benelearn'04: Proceedings of the Thirteenth Belgian-Dutch Conference on Machine Learning, 2004, pp. 87–94.

[30] L.-C. Lan, W. Li, T.-H. Wei, I. Wu, et al., Multiple policy value monte carlo tree search, arXiv preprint arXiv:1905.13521 (2019).

[31] 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 529 (2016) 484.

[32] H. Wang, M. Emmerich, M. Preuss, A. Plaat, Hyper-parameter sweep on alphazero general, arXiv preprint arXiv:1903.08129 (2019).

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, The journal of machine learning research 15 (2014) 1929–1958.

[34] A. Krogh, J. A. Hertz, A simple weight decay can improve generalization, in: Advances in neural information processing systems, 1992, pp. 950–957.

[35] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., Mastering the game of go without human knowledge, Nature 550 (2017) 354.

[36] I. Bratko, Alphazero–what's missing?, Informatica 42 (2018).

[37] R. Girshick, J. Donahue, T. Darrell, J. Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2014, pp. 580–587.

[38] S. Ren, K. He, R. Girshick, J. Sun, Faster R-CNN: Towards real-time object detection with region proposal networks, in: Advances in neural information processing systems, 2015, pp. 91–99.