TsetlinGo: Solving the game of Go with Tsetlin Machine

Kristoffer Lomeland Magelssen
Trond Storebø

SUPERVISORS
Dr. Ole-Christoffer Granmo

Master's Thesis
University of Agder, 2020
Faculty of Engineering and Science
Department of ICT

Abstract

The Tsetlin Machine have already shown great promise on pattern recognition and text categorization. The board game GO is a highly complex game, and the Tsetlin Machine have not yet been tested extensively on strategic games like this. This thesis introduces *TsetlinGO* and aims to *Solve the game of Go with Tsetlin Machine*. For predicting the next moves a combination of Tsetlin Machine and Tree Search was used. In the thesis a 9x9 board size was used for the game of Go, to prevent the problem from becoming to complex.

This thesis goes through hyper-parameter testing for classification of the Go board game. A solution with Tree Search and Tsetlin Machine combined is used to perform self-play and matches between Tsetlin Machines with different hyper-parameters.

Based on the empirical results, our conclusion is that the Tsetlin Machine is more than capable for classification of the game of Go at various stages of play. Results from the experiments could be seen to achieve around 90%, while further climbing up to around 95% upon re-training.

From examining the clauses, strong patterns was found that gave insight into how the machine works. The Tsetlin Machine was able to play complete games of Go, making connections on the board through use of patterns from the clauses. It was found that the size of the clauses had great impact as clauses with large patterns had trouble getting triggered in early play. The high accuracy from classification was found to not correlate with how strong the Tsetlin Machine would perform during self-play. This may indicate that producing training data directly from self-play may be required to fine tune the assessment of board positions faced during actual play. We can conclude that this thesis provide a benchmark for further research within the field of Tsetlin Machine and the game of Go.

Table of Contents

List of Figures

List of Tables

Part I

Research Overview

Chapter 1

Introduction

1.1 Motivation

The problem outlined in this thesis is to solve the game of GO using the Tsetlin Machine and analyze patterns in the results.

The game of GO is a highly complex game and it would be interesting to see how the Tsetlin Machine would handle it using propositional logic. The Tsetlin Machine have previously shown great promise on pattern recognition [14] and text categorization [15]. The Tsetlin Machine have not yet been extensively tested on strategic games like Go, and that is a logical next step for the Tsetlin Machine.

The game of Go can consist of both small and larger patterns crossing the board. While the rules are simple it is largely a tactical game where short term losses can give long term gains. The understanding of how neural networks work is generally very poor, and not being able to look closer at what happens within the network can lead to seemingly strong results being actually weak and a network that is easy to trick [24].

The game of Go have not yet been completely solved due to the high number of different moves possible. A 19x19 Go board game have a higher amount of possible moves than Chess [23]. A 9x9 board will be used in our approach to reduce the complexity allowing more resources to be used on optimizing settings and evaluating our findings. The Tsetlin Machines patterns can also more easily be evaluated as the machine is more transparent than other solutions currently available. Being able to closely examine the patterns gives more insight into the function of the machine and how it handles the input.

The approach outlined in this paper focuses on training a Tsetlin Machine to being able to play the game of Go. To play the game a tree search will be used instead of an algorithm in order to better test the Tsetlin Machine. To limit the complexity of the problem a 9x9 board is used, which is often played by lower tier players of the game. Further examination of the clauses and the patterns will give more insight into how the machine learns and how it decides.

1.2 Thesis definition

The primary objective being outlined in this thesis is to solve the game of Go on a 9x9 board using the Tsetlin Machine. The research is split into four goals following up with the thesis hypotheses.

1.2.1 Thesis Goals

Goal 1: *Optimize hyper-parameters for the Tsetlin Machine using full boards. Evaluate classification from a given board position using the Tsetlin Machine.*

Goal 2: *Play the game of Go from a given position using Tree Search*

Goal 3: *Implement self-play on Go using Tree Search with different Tsetlin Machine configurations.*

Goal 4: *Investigate the interpretability of the Tsetlin Machine by evaluating clauses and their patterns.*

1.2.2 Hypotheses

Hypothesis 1: *The Tsetlin Machine will be able to find patterns and predict the outcome of a game from various stages*

Hypothesis 2: *The Tsetlin Machine will be able to play the game of Go.*

1.2.3 Summary

The first goal of the thesis is to optimize the hyper-parameters for the Tsetlin Machine, and analyze the results from classifying a Go board game from a given position using the Tsetlin Machine. Second, is to play the game of Go from a given position using Tree Search. The third is to implement a solution for self play using Tree Search and Tsetlin Machines with different configurations. Lastly, the final goal is to analyze the interpretability of the Tsetlin Machine by evaluating the clauses created and their patterns.

1.3 Contributions

This thesis introduces further testing of the propositional logic in the Tsetlin Machine for the strategic game Go. As well as an analysis of the clauses and patterns from the results of using the Tsetlin Machine on the board game Go.

By looking at clauses and patterns from the Tsetlin Machine one can see how the Tsetlin Machine interpret the game, which is not possible by using a neural network. This thesis presents a method on how to solve the game of Go with Tsetlin Machine and pattern analysis of the results.

There is to the best of our knowledge no documented research on using the Tsetlin Machine to solving the board game Go, or analyzing the patterns from Go.

1.4 Thesis outline

Chapter 2 goes through the introductory background research used as method in the research later on. It will look into the Tsetlin Machine (2.1), the Go board game (2.2), Tree Search (2.3), and K-Fold Cross-Validation (2.4).

Chapter 3 will explore the current state-of-the-art for solutions in Go, where it will delve into AlphaGo (3.1), AlphaGo Zero (3.2), OpenSpiel (3.3), and lastly Monte Carlo Tree Search (3.4).

Chapter 4 outlines the data structure used later in the experiments. It will go through the original Dataset (4.1), Datasets for various game states (4.2), Data conversion (4.3), and Implementation of 10-Fold (4.4).

Chapter 5 introduces the proposed solutions for the goals defined in subsection 1.2.1. Section 5.1 outlines a conceptual model of how the implementation was solved. Section 4.3 describes the Data Conversion with Gomill. While section 5.3 describes how the Tree Search and Self Play was done for the experiments in the thesis.

Chapter 6, 7 and 8 presents and discuss results from experiments using the solutions presented in chapter 5.

Chapter 9 concludes the thesis' hypotheses and provides a summary of what has been done in the thesis. Section 9.2 introduces the future research related to the thesis.

Chapter 2

Background

In this chapter the background theory for topics related to the research performed later will be defined. Section 2.1 will go through how the Tsetlin Machine and the various versions work, and will move on to describe the Go board game in section 2.2, which is used in this research. Thereafter, section 2.3 will go through the Tree Search method. In the end, it will go through the K-Fold Cross-Validation method in section 2.4.

2.1 Tsetlin Machine

The Tsetlin Machine consist of a collective of Tsetlin Automatas, that use propositional logic to solve complex systems [14]. A Tsetlin Automata can be considered as a state machine that votes on the outcome and are either penalized or rewarded for the vote. In Figure 2.1 a two action Tsetlin Automaton is viewed with solid lines for rewards and dotted lines for penalty. Receiving rewards will move the state further along the action path it is on and to be more resolved in the action performed, while being penalized will move the state in the opposite direction towards the center and a possible action change.

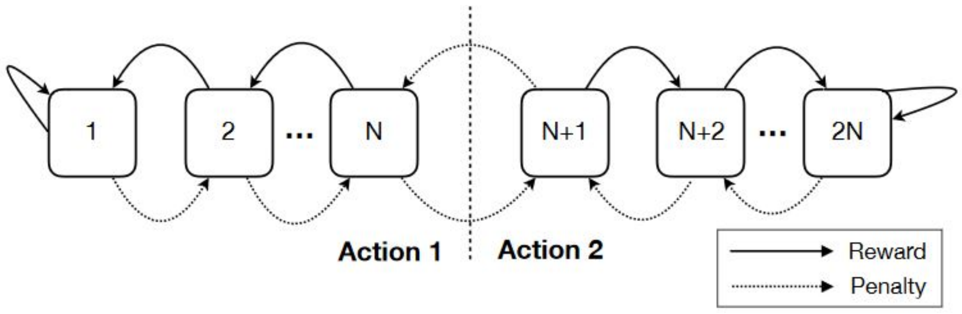


Figure 2.1: A Tsetlin Automaton with two actions reprinted from [14].

The system of a Tsetlin Machine is built up by *features* and *clauses* contained in one or more Tsetlin Machines. A feature is a distinctive attribute of the system that is to be modelled. Each input variable would be considered a feature. A clause, is a conjunctive clause, built up by feature vectors and the weights. A clause attempts to display a pattern logically and are controlled by the Tsetlin Automatas. Each feature contains two literals controlled by an automata, one will vote to include or exclude the feature in the evaluation. During training the automatas will adjust the patterns in the clause based on getting rewarded or punished. A generic example of a trained clause would be

$$C_1 = x_1 \land x_3 \land x_4.$$

To achieve even better results the automatas are not only looking for patterns that contain certain input, but also looking into what input a pattern should not contain. Thus, we can have more complex clauses, but also more complex modeling tasks can be performed

$$C_2 = \neg x_1 \land x_2.$$

The positive and negative evaluations are kept separate for easier handling. The logical $\land$ operation will be performed upon evaluation of the output, making a clause to output either 0 or 1. Upon evaluation, the Tsetlin Machine sums up the clauses

$$C(input) = \sum Positive - \sum Negative.$$

In Figure 2.2, the Tsetlin Machines relation to clauses, features and literals are shown. For each outcome a Tsetlin Machine is created containing both non-negated clauses and negated clauses, a clause will have a feature for each input variable, and two Tsetlin Automatas agreeing on wether to include or exclude the feature(variable):

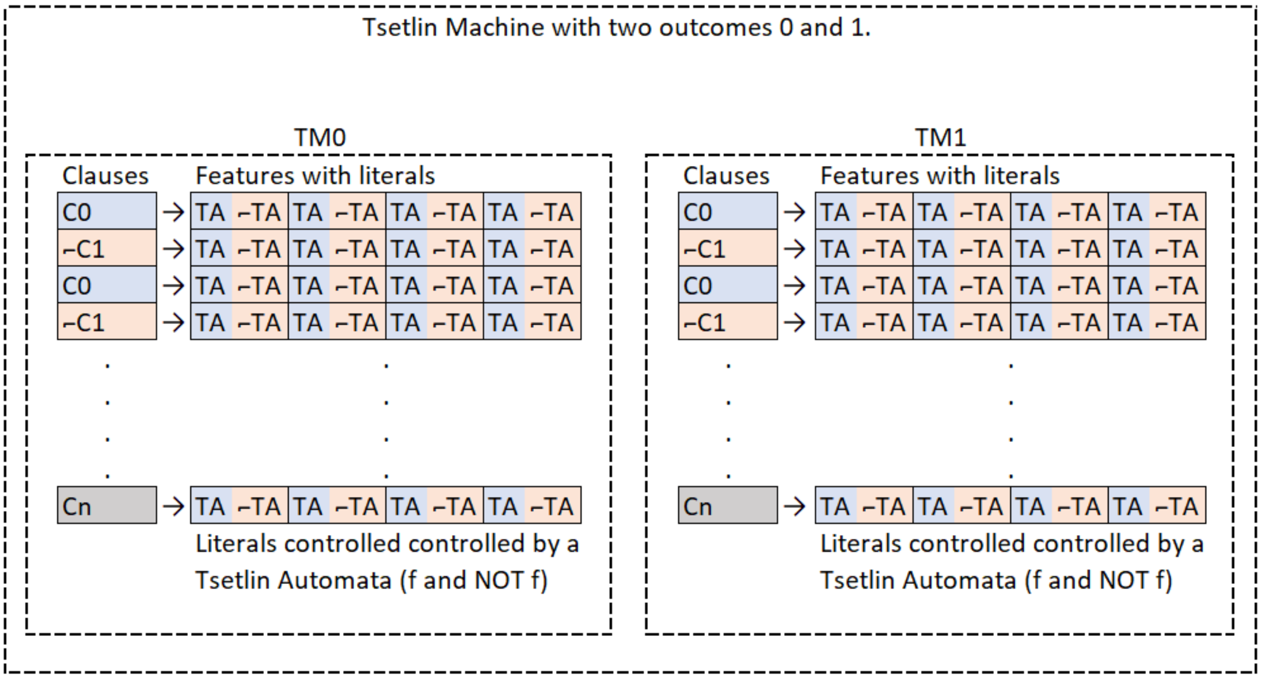
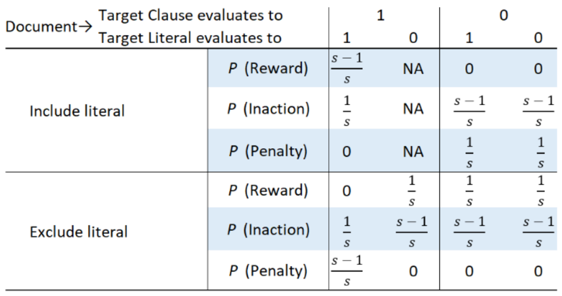

Tsetlin Machine with two outcomes 0 and 1.
TM0
TM1
Clauses
Features with literals
C0
¬C1
Cn
TA ¬TA
Literals controlled controlled by a Tsetlin Automata (f and NOT f)

Figure 2.2: Clauses with features and literals.

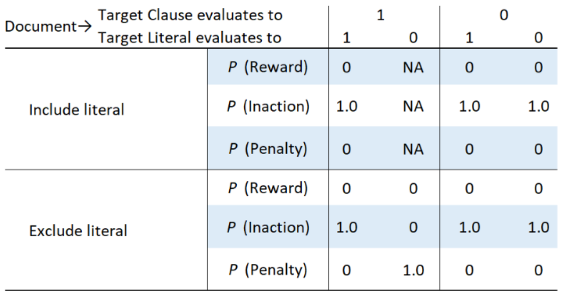
In order to prevent multiple clauses from targeting the same pattern, a upper limit/threshold T is used. The Tsetlin Machine will ask some of the clauses to change if $|C(input)| > T$. This way the Tsetlin Machine tries to force some clauses to choose another pattern.

A value s determine the frequency of feedback based on how the Machine and the clauses evaluate. The feedback consist of two types, Type I and Type II. Their role is to combat false output, and to force a Tsetlin Machines clauses into another direction should it evaluate falsely. They are triggered based on the s value, with a higher value increasing the chance of change. Type I handles false negative output and Type II handles false positive output. The reference tables for the two types can be seen in Figure 2.3 and Figure 2.4.



Document→	Target Clause evaluates to / Target Literal evaluates to	1		0	
		1	0	1	0
Include literal	P (Reward)	$\frac{s-1}{s}$	NA	0	0
	P (Inaction)	$\frac{1}{s}$	NA	$\frac{s-1}{s}$	$\frac{s-1}{s}$
	P (Penalty)	0	NA	$\frac{1}{s}$	$\frac{1}{s}$
Exclude literal	P (Reward)	0	$\frac{1}{s}$	$\frac{1}{s}$	$\frac{1}{s}$
	P (Inaction)	$\frac{1}{s}$	$\frac{s-1}{s}$	$\frac{s-1}{s}$	$\frac{s-1}{s}$
	P (Penalty)	$\frac{s-1}{s}$	0	0	0

Figure 2.3: Type I Feedback to combat False negative output, reprinted from [15]



Document→	Target Clause evaluates to / Target Literal evaluates to	1		0	
		1	0	1	0
Include literal	P (Reward)	0	NA	0	0
	P (Inaction)	1.0	NA	1.0	1.0
	P (Penalty)	0	NA	0	0
Exclude literal	P (Reward)	0	0	0	0
	P (Inaction)	1.0	0	1.0	1.0
	P (Penalty)	0	1.0	0	0

Figure 2.4: Type II Feedback to combat False positive output, reprinted from [15]

Building further on the Tsetlin Machine, Prof. Granmo looked into Convolutional Neural Networks (CNN) and the success of such methods in pattern recognition [16]. CNNs have been very popular and numerous different CNN architectures have been published. A general problem with CNNs is their complexity and being non-transparent. CNNs perform very well, but exploring why they perform so well is hard, which limits the steps of improvement. The new Convolutional Tsetlin Machine is a Binary CNN. Binary CNNs is a way of mitigating the intense computations needed for training CNNs by only allowing two possible values. This allows accumulations to replace many of the intense multiplications CNNs normally use. The CTM builds upon the TM and the learning is still bandit based. Being based on fast summation and logical operators the CTM is an extreme Binary CNN.

While in the TM the whole image is generated into a clause, the CTM uses each clause as a convolutional filter. And each clause will have patches that can be a smaller window of the image. Important to note is that this also requires the patch to capture the x and y coordinates of the selected window dimension. So whereas before the TM gave out one clause for one image, a clause would have several patches based on how many different ways the window dimension can fit within the whole image.

The CTM will select one of the patches that made the clause evaluate to 1 and will update the clause based on it. It will do this randomly to increase the diversity in the clauses.

In late November 2019, Prof. Granmo improved upon the Tsetlin Machine with the Weighted Tsetlin Machine (WTM) [25]. The idea here was simple yet very effective, and involved making the voting system more effective. Since a clause can occur several times, the old Tsetlin Machine would repeat it for each occurrence. The new approach involved giving value or "weighting" the clauses. Instead of repeating the clauses w number of times, one clause would get w number of weights and the duplicates would be removed. A weight could also be given less value by assigning $w = 0.5$. This makes it both more compact, faster and reduces memory usage.

Prof. Granmo recently released a new addition to the Tsetlin family called pyTsetlinMachineParallel [8]. This version is a multi-threaded version of the Tsetlin Machine that allows for both the convolutional and weighted settings in addition to running several operation at the same time, the parallel version increase the speed additionally and during testing no substantial differences was detected between the results of running the original machine and the parallel version.

2.2 Go board game

Go is an ancient board game with simple rules, and a game of Go can last for hours [7]. Go rewards patience and balance, and early mistakes or gains can easily be reversed as the game progresses. Go originated in China more than 2500 years ago, and is still played in its original form [1]. The game has a long history in China, Japan and Korea, and naturally has its strongest players in this region, while western countries got its eyes for Go in the early 1900s. The game is usually played on a 19x19 grid, although 9x9 and 13x13 is also common for beginners [3].

Go is a game between two players, named Black and White. It is played on a board, which consists of a plane grid of 19 horizontal and vertical lines. A point on the board where the horizontal and vertical lines meet are called an intersection. Two intersections are adjacent if they are distinct and connected by a horizontal or vertical line with no other intersection between them. The tokens used to play in Go are called stones. Each player is given sufficient stones to play with, usually 181 for Black and 180 for White. If this is insufficient they will be given more stones. An intersection can only have one of the following three states at any time during the game; empty, occupied by a black stone, or occupied by a whites stone. A position consists of the state of each intersection. It requires no information of whose turn it is, nor any information regarding previous moves or states of the game. Two placed stones of the same color, or two empty intersections, are called connected if it is possible to draw a trail from one to the other by only passing through adjacent intersections which have the same state. A liberty of a stone in a given position is an empty intersection adjacent to the stone in question or adjacent to a stone which is connected to the stone in question. [5, 4, 19, 20]

At the start of the game, the board is empty unless the players have agreed to have a handicap placed. Black moves first, and the players alternate afterwards. When it is a players turn, they can either pass, and perform no action, or play. A play comprise of the following steps:
Step 1: Place a stone of their color on an empty intersection.
Step 2: Remove any stones of the opponents color on the board that has no liberty.
(Step 3): Remove any stone of their own color on the board that has no liberty. [5, 4, 19, 20]

This means that on each turn a player moves once, either a play or pass. Step 3 can also be called self-capture and most games states that any move that will result in a self-capture is illegal. To avoid very repetitive moves the board can not change into a previous state. This rule is called the *Ko* rule, and is there to avoid repetitive capture as demonstrated in Figure 2.5 and Figure 2.6. [5, 4, 19, 20]

The game ends when both players have passed their turn consecutively. The position of the board at the time both players pass consecutively is called the final position. When looking at the final position, an empty intersection belongs to a player's territory if all the stones adjacent to it or to an empty intersection connected to it are of that player's color. At the same time when looking at the final position, an intersection belongs to a player's area if it either belongs to that player's territory or it is occupied by a stone of that player's color. If one of the players has a higher score than the other, then the player with the highest score wins. If not, then the game is drawn. [5, 4, 19, 20]

There are two main scoring systems; territory scoring and area scoring. For the territory a player's score is determined by the number of empty intersections that the player has surrounded minus the number of stones their opponent has captured. Area score is determined by the number of stones that player has on the board as well as the empty area surrounded by the stones of that player. [5, 4, 19, 20]

Other rules that are optional are komi and handicap. Komi is a way to offset black's advantage by having the first move. It's a fixed amount of points, which are agreed upon before the match, added to white's score at the end of the game. Common values used for komi are 5.5, 6.5, 7.5. If the game also uses handicap, the value is often set to 0.5. Handicap is when the weaker player is allowed to take black and the first few moves for white are forced as pass moves. The number of moves are usually the difference between the player's ranks. [5, 4, 19, 20]

This project uses the area scoring as its scoring system. This is due to it using a Go library which has area score implemented, but this swill be explained in more detail later in chapter 5

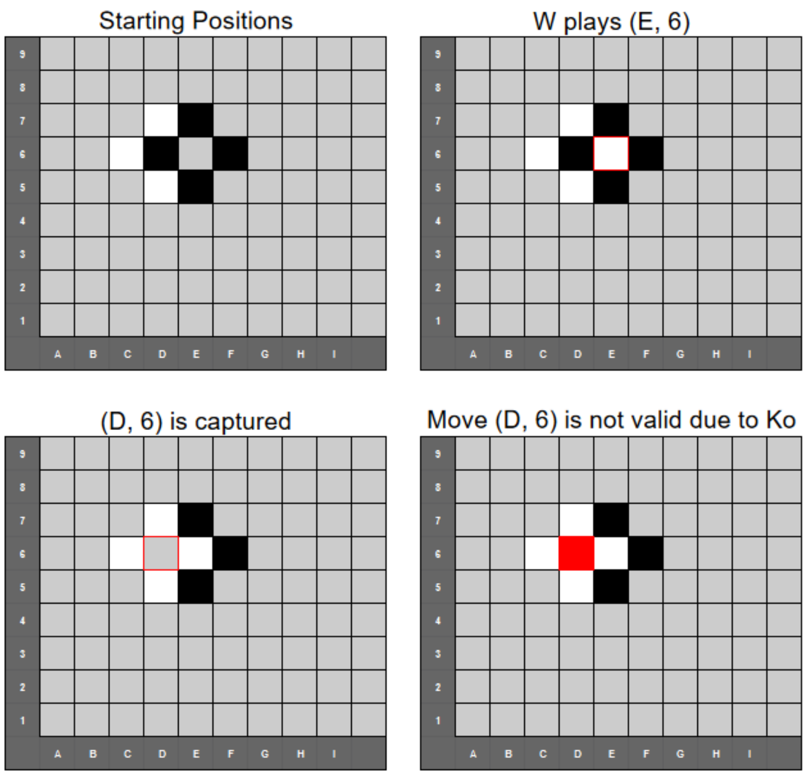


Figure 2.5: Illustration of *Ko* rule.

Figure 2.5 illustrates the *Ko* rule. To avoid the *Ko* rule player black can play a different move in between the capture of (D, 6) by white and recapture by black as seen in Figure 2.6.

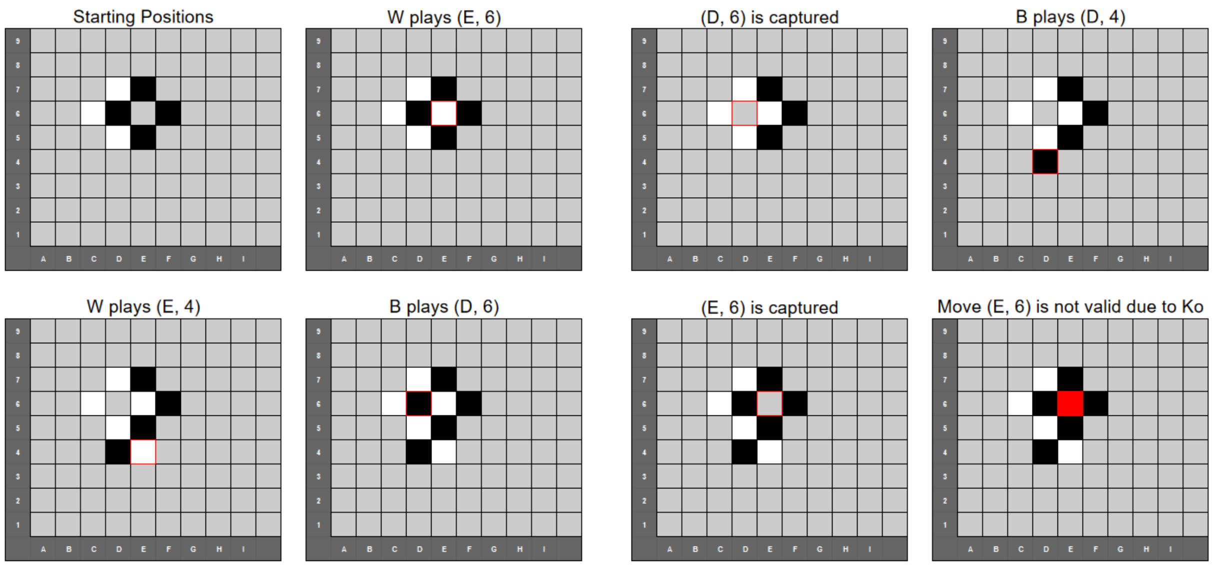


Figure 2.6: Illustration of recapture by not breaking *Ko* rule.

Figure 2.7 depicts an example game of Go on a 4x4 board. For convenience when creating the figure, the squares are used as the intersections. In the figure below you can see an example of a capture when Black does the move D2 and White's stones at D3 and D4 gets captured. By following the area scoring the game ends up with a score of 16 for Black, and a score of 0 for White. While if you follow the territory score then Black has a score of 15, where 10 is territory and 5 is captures. White has 0 on this scoring system as well.

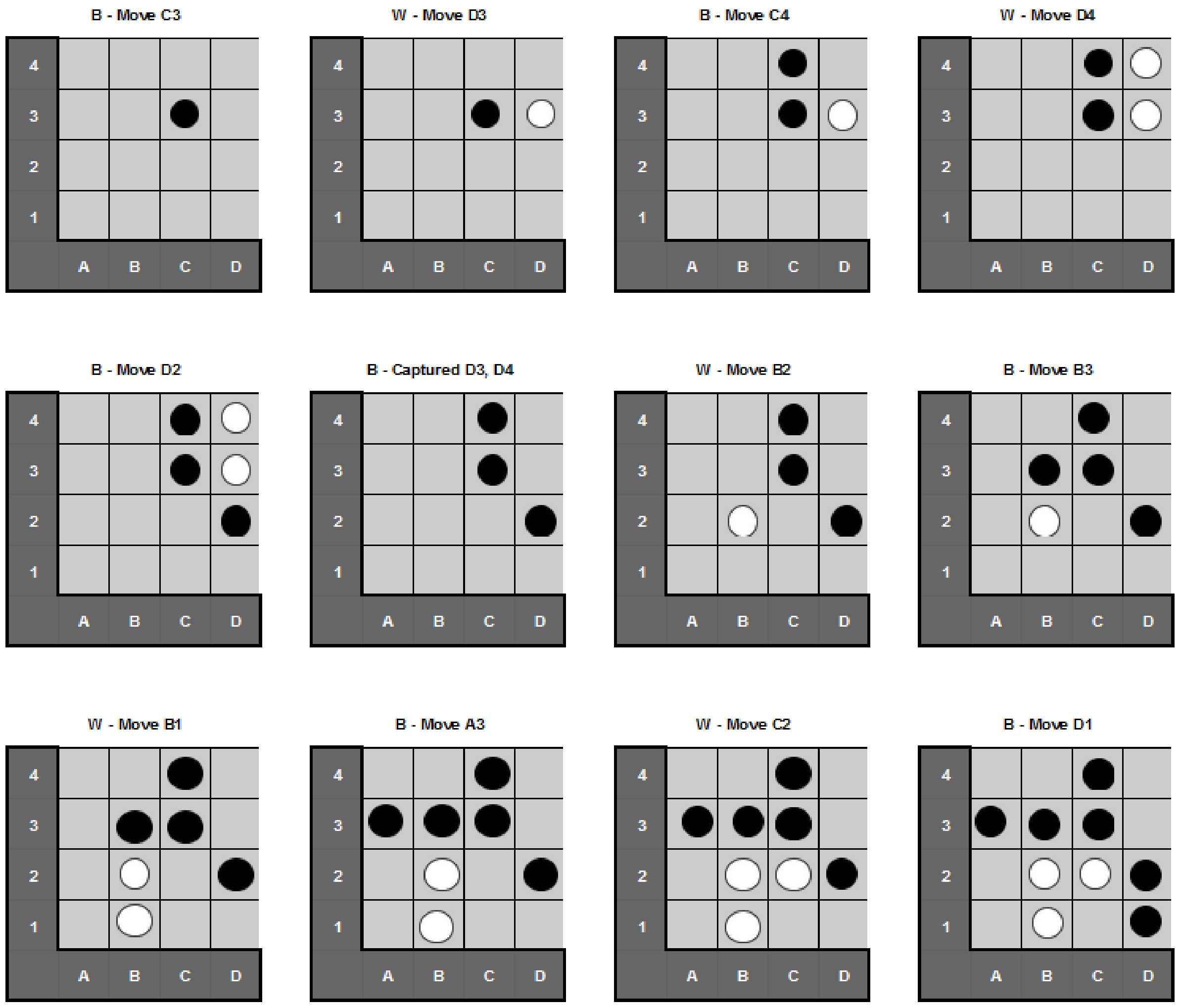


Figure 2.7: Illustration of an example game of Go.

2.3 Tree Search

Often in board games there are several ways to achieve victory. The complexity in games can lead to an extremely high number of possible moves and outcomes. To find the optimal solution, all options should be tested, or one can settle for the best possible solution given time and cost required to achieve it.

In Figure 2.8 a tree search example is shown. An empty board tries all possible moves for player B, then the new boards created will again try out all possible moves for player W.

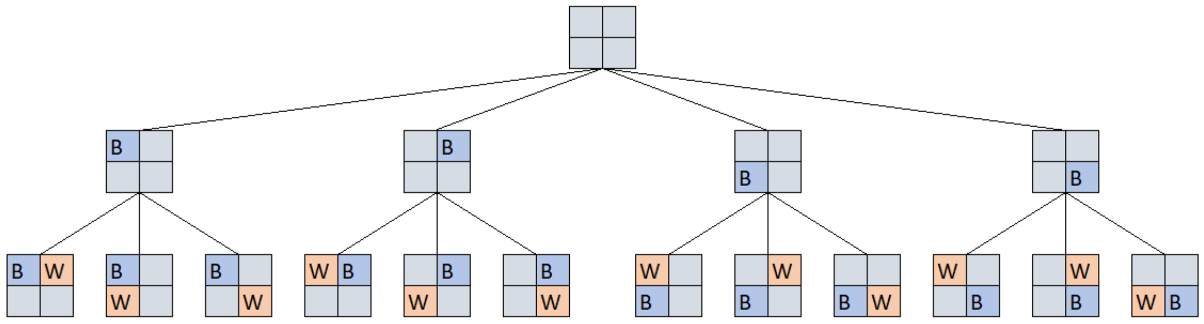
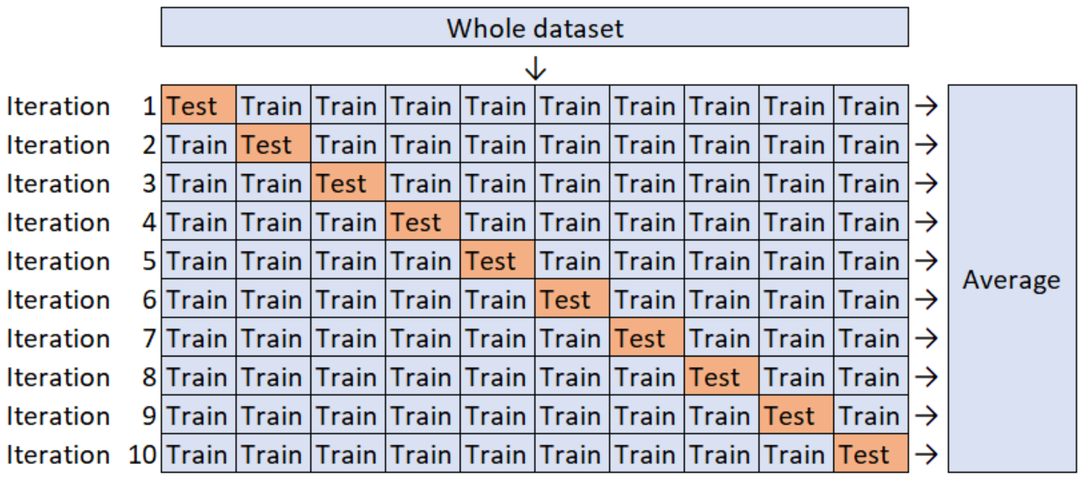


Figure 2.8: Example of a Tree search.

There are many strategies to perform a tree search. Algorithms use different approaches to achieve the desired result. The different algorithms can be grouped into two categories, uninformed search and informed search [17]. Informed search algorithms uses a function to estimate additional information like cost to get there, called heuristic values [9]. This way an informed search algorithm sacrifices precision for speed. Uninformed search does not contain heuristics, all they can do is evaluate if they have reached their goal or not [26]. There are still different methods of performing an uninformed search such as going depth first or breadth first. While informed algorithms reduce the search space through its strategy of selecting the most likely candidates, uninformed search traverse the three methodically requiring time and computer power [6]. If the search space becomes to large, it becomes to complex for uninformed search algorithms.

2.4 K-Fold Cross-Validation

To avoid over-fitting, training and testing should be done with different parts of the dataset. This could however lead to selection bias, meaning that the selection for training and testing is abnormally good by pure chance of random selection [27]. To avoid this, k-Fold Cross-Validation is commonly used within machine learning. The k decides how many times the same test will be run with different configurations of the dataset. A very common number of k is 10. A common way of performing K-Fold Cross-Validation is splitting the dataset into k blocks and divide it into test and training pair where each block is in the test set once and the remaining blocks are in the training set [12]. An illustration of a k-Fold experiment is shown in Figure 2.9.



Iteration 1	Test	Train	Train	Train	Train	Train	Train	Train	Train	Train
Iteration 2	Train	Test	Train	Train	Train	Train	Train	Train	Train	Train
Iteration 3	Train	Train	Test	Train	Train	Train	Train	Train	Train	Train
Iteration 4	Train	Train	Train	Test	Train	Train	Train	Train	Train	Train
Iteration 5	Train	Train	Train	Train	Test	Train	Train	Train	Train	Train
Iteration 6	Train	Train	Train	Train	Train	Test	Train	Train	Train	Train
Iteration 7	Train	Train	Train	Train	Train	Train	Test	Train	Train	Train
Iteration 8	Train	Train	Train	Train	Train	Train	Train	Test	Train	Train
Iteration 9	Train	Train	Train	Train	Train	Train	Train	Train	Test	Train
Iteration 10	Train	Train	Train	Train	Train	Train	Train	Train	Train	Test

Figure 2.9: Illustration of k-Fold in practice.

Here the whole dataset is split up in blocks, where each block alternates being the test block while the rest are used for training. Each result is then added and the average becomes the unbiased result.

Chapter 3

State-of-the-art

3.1 AlphaGo

DeepMind is an organisation which was started by a team of scientists and engineers in 2010 that with the aim to enhance the field of AI by using a interdisciplinary approach, by combining knowledge from several different fields [2]. DeepMind have succeeded in producing methods, programs and results some say was years ahead of its time. In 2014, DeepMind was bought by Google and have a special status under the Alphabet structure which allows them to pursue research that might not be profitable for years [28].

AlphaGo was introduced by Google DeepMind in early 2016 as a new approach towards mastering the game of Go. Their approach consist of a combination of learning from games played by expert human Go players and using self-play to optimize the outcome [29]. DeepMind first train a supervised learning network (SL), which is then further improved by a reinforcement learning network (RL). The SL network was trained using moves from expert human Go players. The SL network is further improved by the RL network that uses self play to train the network more in the direction of winning games. This basically moves it from a predictive accuracy direction towards a more long term view on winning the game. For each iteration, the RL network played against a random previous version of itself. A regression trained value network was trained from 30 million distinct self played games from the RL network and was further used to evaluate winning positions. Figure 3.1 illustrates how the asynchronous policy and value MCTS (APV-MCTS) traverse the tree.

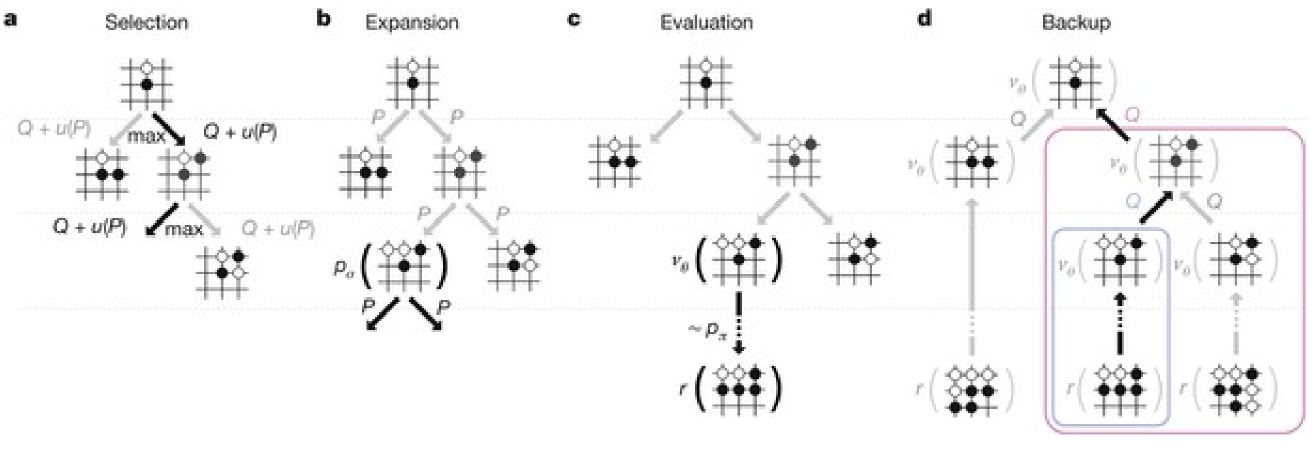


Figure 3.1: AlphaGo's APV-MCTS reprinted from [29].

APV-MCTS differs from normal MCTS by not doing the backpropagation and instead evaluate using the value network and the the outcome of the simulation. AlphaGo was evaluated by holding an internal tournament where AlphaGo was played against several of the presumed best Go programs at the time. It also managed to win a tournament against human champions, which had never happened before. Notably Fen Hui a previous three-time European Go champion said he had never seen humans play some of the moves AlphaGo did, but after some time he was able to grasp how they connected to the rest of the board [22]. Some critics have downplayed the AlphaGo and rather pointed towards the limits of its machine intuition [13]. To avoid loosing on time an effectiveness algorithm tries to make sure that the approximately the same amount of time is made on each move. While a human player would use longer time on deciding moments in the game, AlphaGo might use to little time, evidently when it lost a game against grandmaster Lee.

3.2 AlphaGo Zero

AlphaGo Zero was introduced by DeepMind in 2017 and unlike the original AlphaGo, AlphaGo Zero starts without any previous knowledge [30]. Starting with completely random moves it is trained entirely from self-play. Figure 3.2 shows how AlphaGo Zero use self-play with tree search.

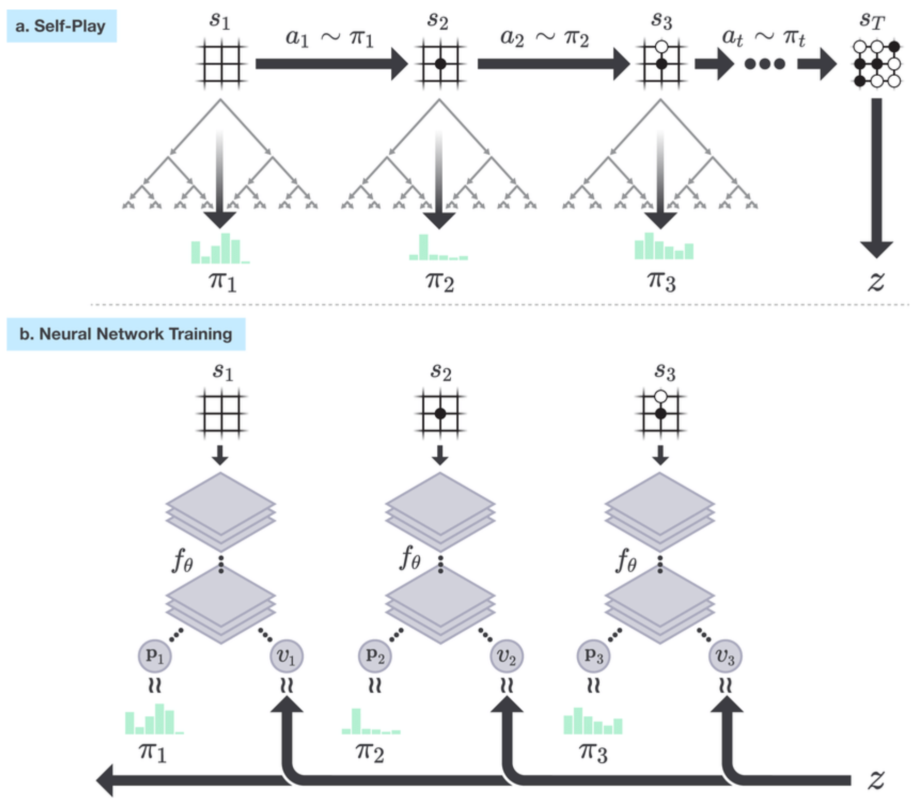


Figure 3.2: Illustration of how AlphaGo Zero use self-play reprinted from [30].

AlphaGo Zero was evaluated against the AlphaGo version that defeated the Go champion Lee Sedol 4-1, and a second machine using the same architecture as AlphaGo Zero but trained based on expert human players moves (Supervised learning). Figure 3.3 shows that although supervised learning achieved better prediction of expert human moves, reinforcement learning quickly outperformed in evaluation rating.

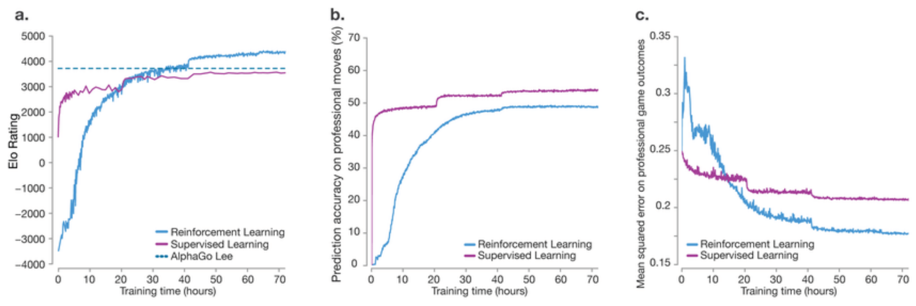


Figure 3.3: AlphaGo Zero performance reprinted from [30].

3.3 OpenSpiel

In December 2019, the DeepMind team released OpenSpiel: A Framework for Reinforcement Learning in Games. OpenSpiel is an open framework with multiple environment and algorithms and a large number of games implemented[21]. OpenSpiel also allow for the user to add new algorithms and new games to the collection. With a number of tools for visualization and evaluation OpenSpiel presents an easy entry into the world of AI research. With the framework in place, new algorithms can be added and tested again different games, and new game can be added and tested against existing algorithms as well as new ones.

3.4 Monte Carlo Tree Search

The Monte Carlo Tree Search or MCTS for short is a family of algorithms used to get the value of an action through simulation and using this value to adjust the policy towards what the best next move will be [11]. MCTS will build a partial game tree, and as it goes on it will become more accurate as it will take into account previous results from exploring the game tree.

MCTS will iteratively build a search tree until a constraint is reached. This constraint could be based on time, memory or number of iterations. Upon reaching the constraint the best performing root action is returned. Common criteria for selecting best performing root action could be based on maximum value output, most visited node, a combination of value and most visited or nodes based on safety.
Figure 3.4 shows the four steps applied in a search iteration.

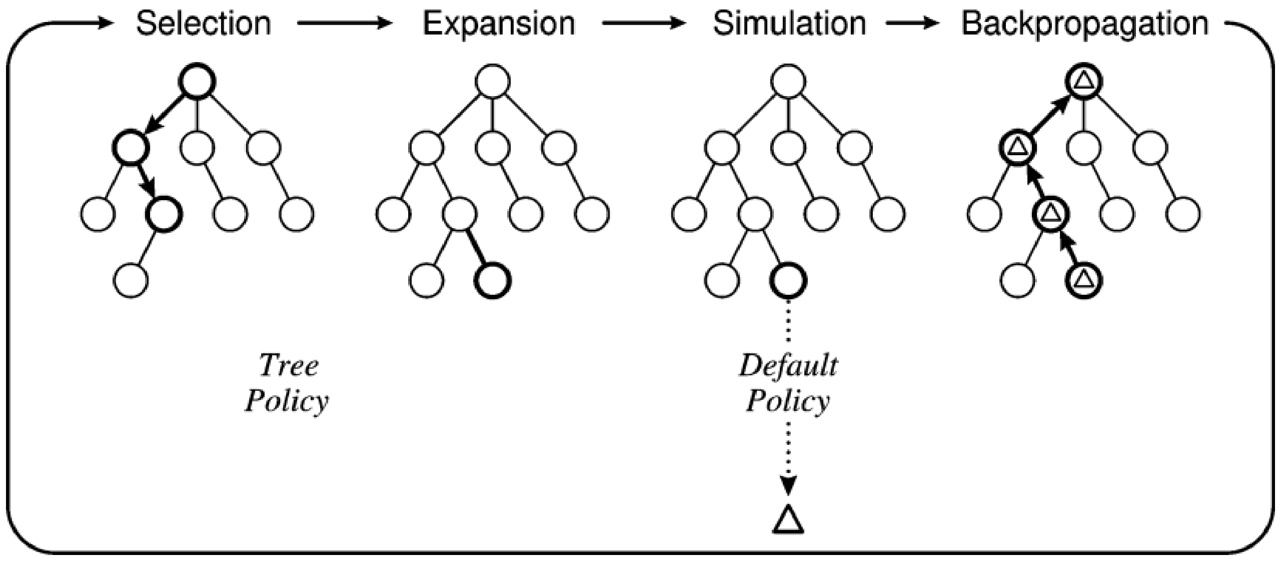


Figure 3.4: The four steps during an iteration in Monte Carlo Tree Search reprinted from [11].

1. Selection: Starting from the root node, nodes are recursively selected until a node that is not fully expanded is found.
2. Expansion: The node is expanded, giving the tree a new leaf node.
3. Simulation: A simulation is run from this node.
4. Backpropagation: The value of the simulation run on this node is backpropagated up the nodes and updates the statistics of the nodes.

The most popular algorithm is the Upper Confidence Bounds for Trees or UCT for short.

$$\text{UCT} = \bar{X}_j + 2C_p\sqrt{\tfrac{2\ln n}{n_j}}$$

With C_p as a non-zero constant, exploitation based on reward are encouraged by the reward term $\bar{X}_j$. The $\sqrt{(2\ln n)/(n_j)}$ term is there to encourage exploration of less frequented nodes. n is the visitation counter for the current node, j is the selected child node, and n_j is the visitation counter for the child node. The algorithm gives each node a chance to be selected by balancing the exploitation and the exploration part. The exploration part is there to ensure that even low rewarding nodes have a probability to be selected. When a child node is visited n_j will increase, giving the node slightly less probability of being selected, also the n counter for the parent will increase and thereby giving the sibling nodes a higher probability of being selected, which should ensure that less frequented nodes will get higher chance of visitation as time goes on. A node will also keep count of how many times it have been been visited and a total payed out reward value. Once a budget on either most visited node, value or a combination as discussed earlier, is reached, the child (of the root) that scores the best will be selected as the best move.

Part II

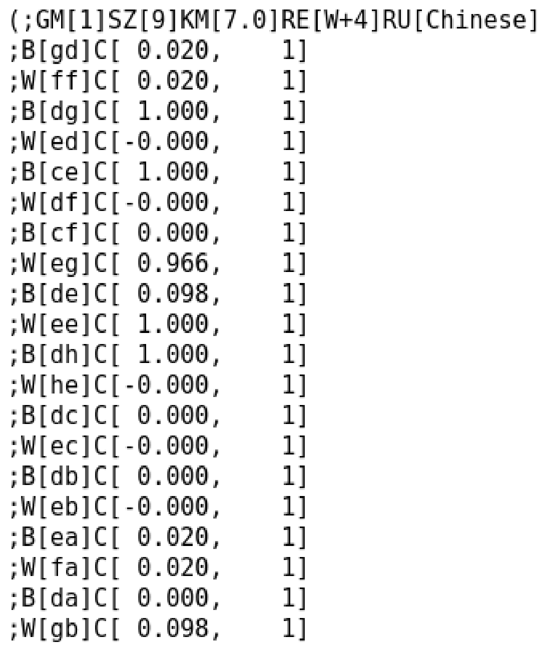
Contributions

Chapter 4

Data Structure

4.1 Dataset

The dataset used in this paper was collected from a site with a collection of self played games using policy network probability [18]. The dataset 9x9_10k_r104_144x20k also called 9x9Aya was chosen due to 9x9 being less complex, while the number of games in the dataset was quite extensive and would provide good data for further research. The 9x9Aya dataset contained 2,340,000 played games of variable length. Each game is contained in a separate *.sgf* file with all the moves played, and the score and information on which player won the game. An example of how the file looks is shown in Figure 4.1.



```
(;GM[1]SZ[9]KM[7.0]RE[W+4]RU[Chinese]
;B[gd]C[ 0.020,    1]
;W[ff]C[ 0.020,    1]
;B[dg]C[ 1.000,    1]
;W[ed]C[-0.000,    1]
;B[ce]C[ 1.000,    1]
;W[df]C[-0.000,    1]
;B[cf]C[ 0.000,    1]
;W[eg]C[ 0.966,    1]
;B[de]C[ 0.098,    1]
;W[ee]C[ 1.000,    1]
;B[dh]C[ 1.000,    1]
;W[he]C[-0.000,    1]
;B[dc]C[ 0.000,    1]
;W[ec]C[-0.000,    1]
;B[db]C[ 0.000,    1]
;W[eb]C[-0.000,    1]
;B[ea]C[ 0.020,    1]
;W[fa]C[ 0.020,    1]
;B[da]C[ 0.000,    1]
;W[gb]C[ 0.098,    1]
```

Figure 4.1: Shows the content of a game file.

GM[ ] means what type of game is in question, SZ[ ] is the size of the board, KM[ ] is the amount of komi assigned in the game, RE[ ] is the results from the game where it's represented by a color and the amount they win by, and RU[ ] means what rule-set is being used for that game. W/B[ ] is a move for either Black or White with the coordinates of the move. The number next to it is from the policy network probability used to create the dataset. [10]

Each file contains the amount of moves done for that game. The minimum moves done in the dataset is 68, while the maximum moves done is 189. The distribution of the dataset per moves done can be seen in Figure 4.2. You can see from the figure that most of the moves is located around the 80 to 108 moves area.

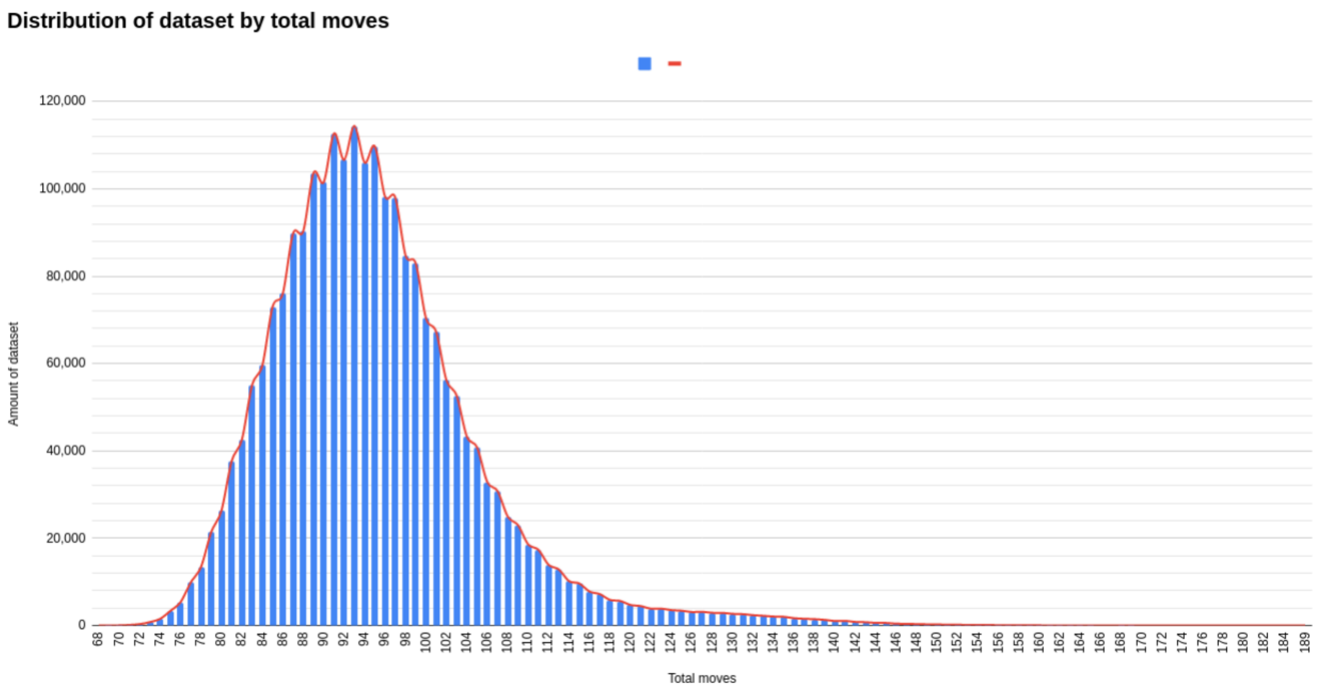


Figure 4.2: An illustration showing the distribution of the dataset by the total moves in the game.

From the player with black stones point of view, the Table 4.1 show the distribution of win, loss and draw in the dataset.

Status	Amount	Percentage
Win	1,072,674	45.84 %
Loss	1,090,240	46.59 %
Draw	177,086	7.57 %
Total	2,340,000	

Table 4.1: Dataset statistics for end game results when all moves have been completed.

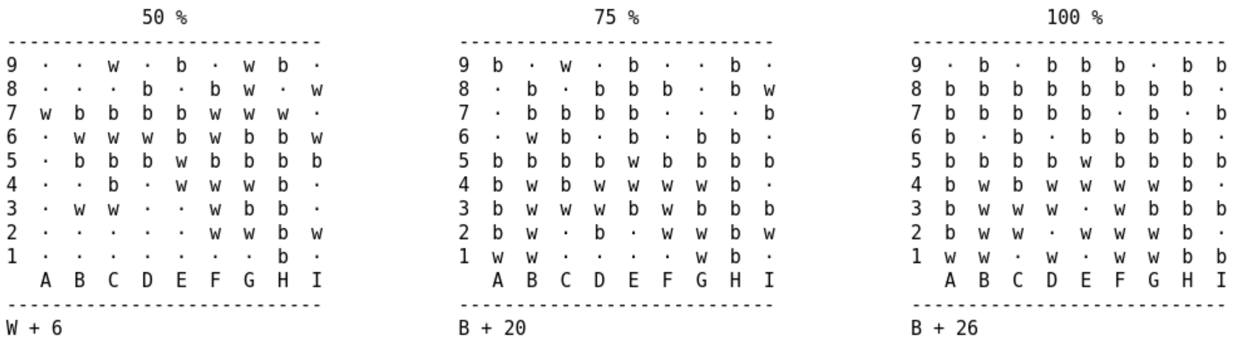
4.2 Datasets for various game states

Both playing a game to the end and stopping before the end would prove interesting, although before the end the player currently in the best position might not be the one ending up as a the winner. In Table 4.2 the datasets generated for moves completed is shown. As can be seen there was relatively few duplicates in the dataset, so no measure to remove the duplicates was taken. When creating the datasets with a certain amount of moves completed, only the boards that had said amount of moves or more were added to the datasets. This means that if the dataset contains board positions after 80 moves, only boards that have 80 moves or more were added. So if a board had 79 moves, it was excluded. Some datasets were also made with a game-threshold. This means that the board game positions were after for example 80 moves, but only the boards that have for example 90 or more moves were included.

Moves completed	**Dataset size**	**Duplicates**	**Win %**	**Loss %**	**Draw %**
All	2,340,000	1	45.84 %	46.59 %	7.57 %
100	614,871	0	55.07 %	41.20 %	3.73 %
95	1,088,475	1	52.84 %	41.91 %	5.25 %
90	1,629,693	6	50.10 %	43.52 %	6.38 %
90_100T	614,871	2	55.07 %	41.20 %	3.73 %
85	2,062,356	11	47.95 %	44.97 %	7.09 %
80	2,283,679	17	46.36 %	46.20 %	7.54 %
80_90T	1,629,693	16	50.1 %	43.52 %	6.38 %

Table 4.2: Datasets with their amount of moves played and duplicates within each dataset.

Figure 4.3 shows the board at various stages of game play with the current score at the bottom, and illustrates how the leader changes as the game progress.



```
            50 %
---------------------------
9  ·  ·  w  ·  b  ·  w  b  ·
8  ·  ·  ·  b  ·  b  w  ·  w
7  w  b  b  b  b  w  w  w  ·
6  ·  w  w  w  b  w  b  b  w
5  ·  b  b  b  w  b  b  b  b
4  ·  ·  b  ·  w  w  w  b  ·
3  ·  w  w  ·  ·  w  b  b  ·
2  ·  ·  ·  ·  ·  w  w  b  w
1  ·  ·  ·  ·  ·  ·  ·  b  ·
   A  B  C  D  E  F  G  H  I
---------------------------
W + 6
```

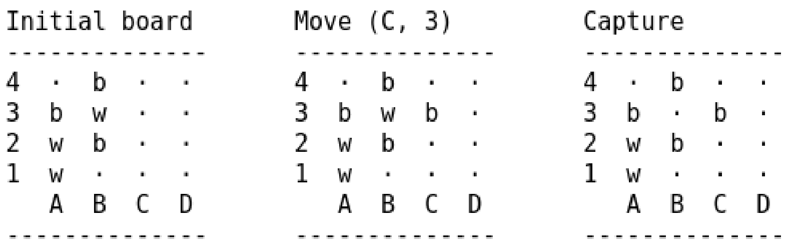
```
            75 %
---------------------------
9  b  ·  w  ·  b  ·  ·  b  ·
8  ·  b  ·  b  b  b  ·  b  w
7  ·  b  b  b  b  ·  ·  ·  b
6  ·  w  b  ·  b  ·  b  b  ·
5  b  b  b  b  w  b  b  b  b
4  b  w  b  w  w  w  w  b  ·
3  b  w  w  w  b  w  b  b  b
2  b  w  ·  b  ·  w  w  b  w
1  w  w  ·  ·  ·  ·  w  b  ·
   A  B  C  D  E  F  G  H  I
---------------------------
B + 20
```

```
            100 %
---------------------------
9  ·  b  ·  b  b  b  ·  b  b
8  b  b  b  b  b  b  b  b  ·
7  b  b  b  b  b  ·  b  ·  b
6  b  ·  b  ·  b  b  b  b  ·
5  b  b  b  b  w  b  b  b  b
4  b  w  b  w  w  w  w  b  ·
3  b  w  w  w  ·  w  b  b  b
2  b  w  w  ·  w  w  w  b  ·
1  w  w  ·  w  ·  w  w  b  b
   A  B  C  D  E  F  G  H  I
---------------------------
B + 26
```

Figure 4.3: A game board is shown in various period of play, 50% moves played, 75% and 100%.

4.3 Data conversion

In order to generate a dataset to predict win, loss or draw, the moves in the various game files had to be played out according to the moves documented, and the rules of the game. Figure 4.4 demonstrates the board before a move, after a move and the board after the white stone was removed from the board according to the rules.



```
Initial board       Move (C, 3)         Capture
-------------       -------------       -------------
4  ·  b  ·  ·       4  ·  b  ·  ·       4  ·  b  ·  ·
3  b  w  ·  ·       3  b  w  b  ·       3  b  ·  b  ·
2  w  b  ·  ·       2  w  b  ·  ·       2  w  b  ·  ·
1  w  ·  ·  ·       1  w  ·  ·  ·       1  w  ·  ·  ·
   A  B  C  D          A  B  C  D          A  B  C  D
-------------       -------------       -------------
```

Figure 4.4: An illustration showing a move on the board leading to the capture of the opponents stone.

After performing the moves in the game files, the resulting board needed to be prepared for input into the Tsetlin Machine. The Tsetlin Machine uses bit format, and due to the 9x9 dimension, each color was given a range of 81 bits each, totaling 162 bits + 1 bit to declare the winner. Figure 4.5 illustrates how a 3x3 dimension board would have been translated into the bit format. Since the player with the black stones start the game, the black positions is added first in the final string, followed by the white and then by the final result bit.

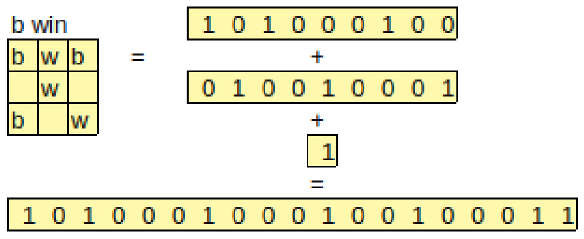


Figure 4.5: Generation of dataset from played moves using 1 for black win, 0 for white win and 2 for draw.

4.4 Implementation of 10-Fold

The total bit dataset was then divided into pure win, loss and draw datasets. Each dataset was then randomized to avoid potential patterns in the descending datasets. The win, loss and draw datasets are then divided into 10 blocks ensuring that the same percentage of each type is represented in each block and each game only occurs once. The blocks are then used to create a 10-fold dataset. To ensure that all experiments have the same train and test sets, 10 training and 10 test datasets are created. Figure 4.6 show how using the blocks to create training and test datasets ensures that no test set occurs in their partner training set, meaning that each k-Fold will first be trained on 90% of the dataset, and then tested on the remaining unknown 10%.

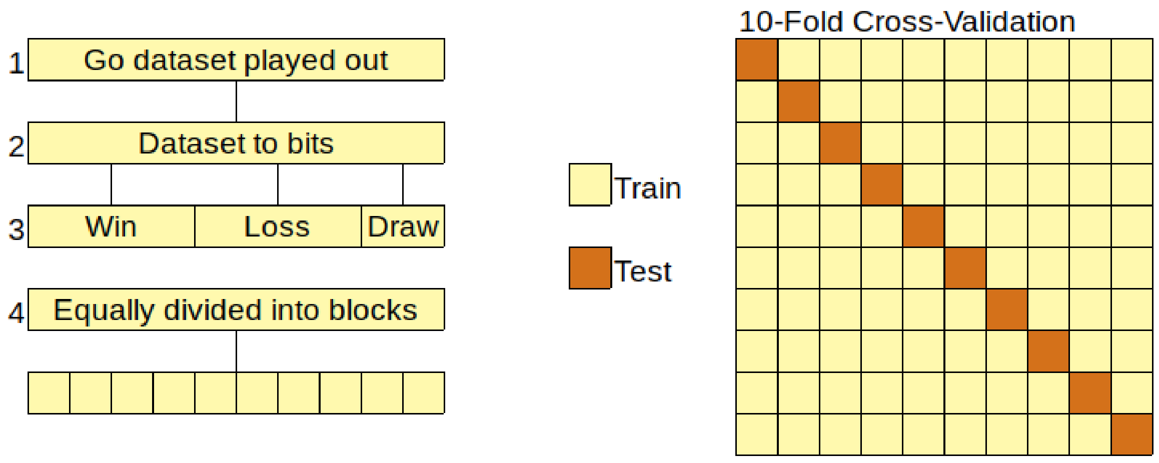


Figure 4.6: k-Fold Cross-Validation performed on dataset.

Chapter 5

Proposed Solutions

In this chapter the proposed solution for the application used in the analysis is shown and explained. First is a conceptual model of the pipeline used as a solution. Second is how the data conversion was done using the gomill library. Thirdly, it goes into detail on the solution for the tree search and self play. Lastly, there is an overview of the solution used for clauses and weights.

5.1 Overview

Figure 5.1 show the overview of the solution used. i) The original dataset consist of a list of moves and to get a board to be used with the Tsetlin Machine, then each move needs to be played out. ii) The game play have produced a board of the finished game. iii) All the games are converted into bits and put into a list to conform with the input requirements of the Tsetlin Machine. iv) k-Fold training is performed on the dataset to avoid biased results. v) For each k-Fold a trained Tsetlin Machine is created and saved, allowing for using the machines in future research. vi) Evaluating the performance of the Tsetlin Machines through accuracy and epochs.

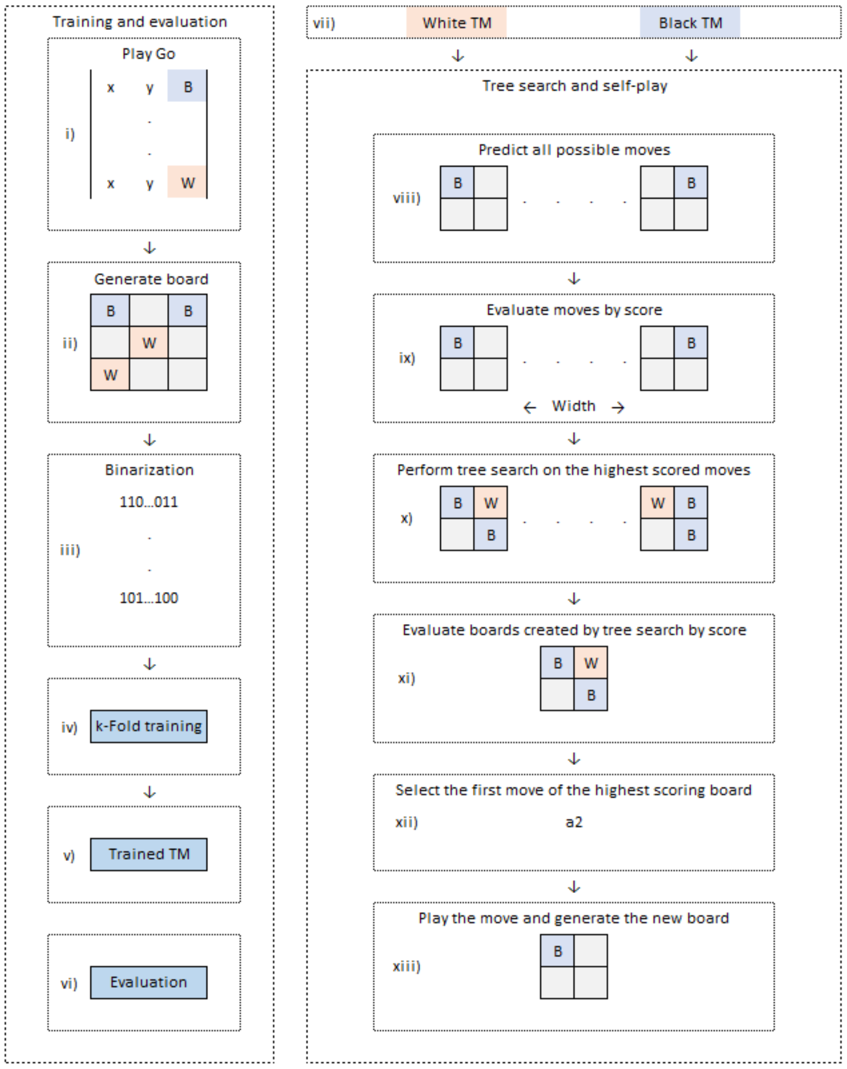
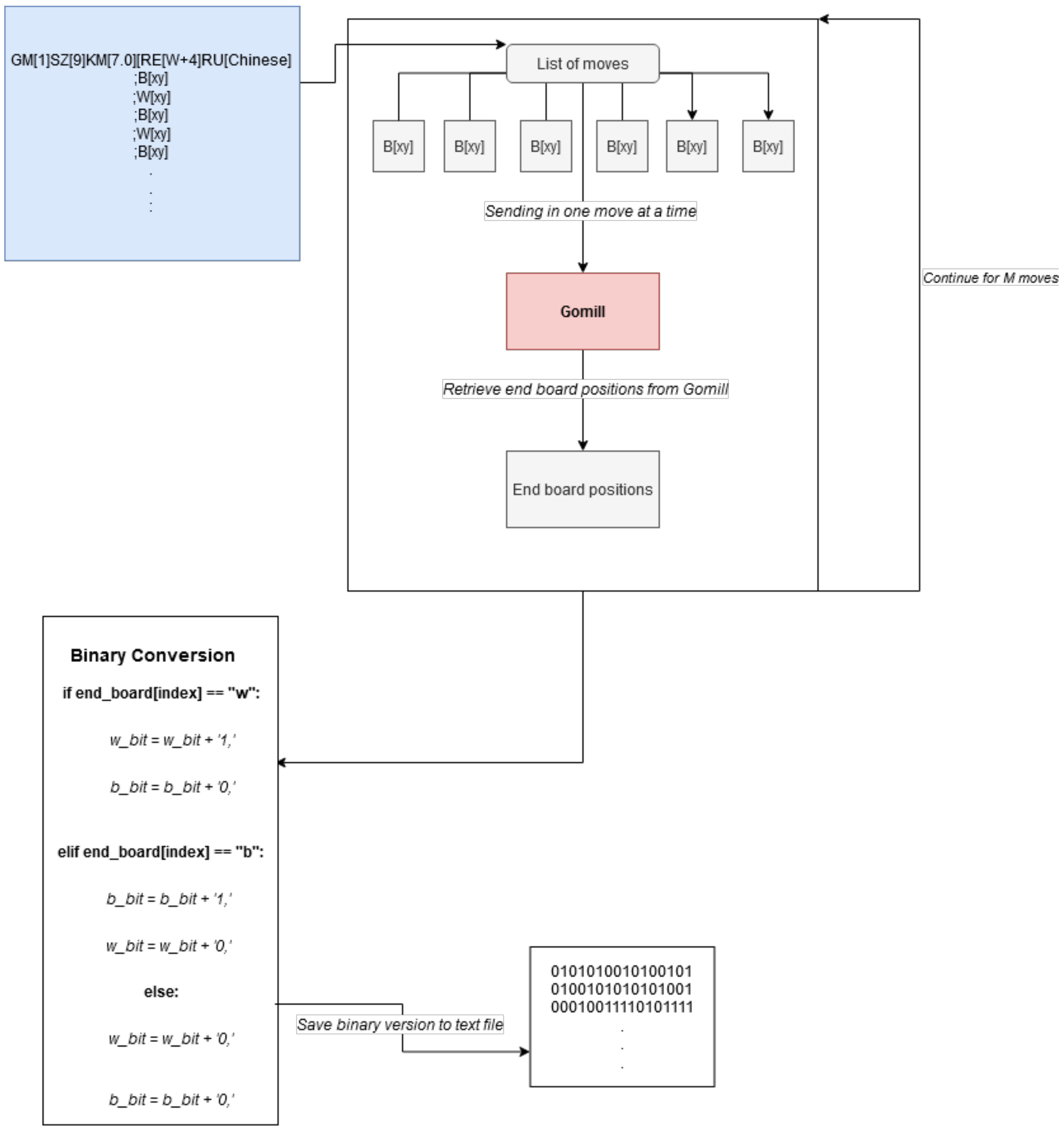

Training and evaluation
Play Go
i)
x y B
x y W
Generate board
ii)
Binarization
iii)
110...011
101...100
iv)
k-Fold training
v)
Trained TM
vi)
Evaluation
vii)
White TM
Black TM
Tree search and self-play
Predict all possible moves
viii)
Evaluate moves by score
ix)
Width
Perform tree search on the highest scored moves
x)
Evaluate boards created by tree search by score
xi)
Select the first move of the highest scoring board
xii)
a2
Play the move and generate the new board
xiii)

Figure 5.1: Pipeline overview of the solution.

vii) Selected white and black Tsetlin Machines are loaded into the self-play algorithm. viii) All open positions are evaluated with scores. ix) Based on the selected width of the tree search, the highest scoring boards are selected. x) A tree search is performed on the selected boards. xi) The highest ranked boards from the tree search is evaluated by score. xii) From the highest scoring board the first move is selected. xiii) The best move for the current player are played out on the board and the board will pass to the next player, which will repeat the same process as the previous player using his own Tsetlin Machine.

5.2 Data Conversion with Gomill

Figure 5.2 is an overview over the data conversion using the gomill library. [31] The original dataset is gathered from a .sgf file and all the moves are gathered and stored in a list. The gomill library is first used by creating an instance of a go board game by using *gomill.boards.Board(board_size)*. [32] The moves from the list is then sent one by one into the go board instance. From this board one can also retrieve the end board position. When it is done with going through all the moves defined in the settings it will retrieve the end board positions and send it to a binary conversion function. It will then convert the white stones into binary and the black stones into binary as shown in Figure 4.5. When it is done with converting all the intersections it will save the binary data as a line in a .text file.


GM[1]SZ[9]KM[7.0][RE[W+4]RU[Chinese]
;B[xy]
;W[xy]
;B[xy]
;W[xy]
;B[xy]
List of moves
B[xy]
B[xy]
B[xy]
B[xy]
B[xy]
B[xy]
Sending in one move at a time
Gomill
Retrieve end board positions from Gomill
End board positions
Continue for M moves
Save binary version to text file

```
Binary Conversion
if end_board[index] == "w":
    w_bit = w_bit + '1,'
    b_bit = b_bit + '0,'
elif end_board[index] == "b":
    b_bit = b_bit + '1,'
    w_bit = w_bit + '0,'
else:
    w_bit = w_bit + '0,'
    b_bit = b_bit + '0,'
```

```
0101010010100101
0100101010101001
0001001111010111
.
.
.
```

Figure 5.2: An illustration showing the overview of data conversion using the gomill library.

5.3 Tree Search and Self play

Figure 5.3 illustrates how the black players first move is decided. i) Starting with a blank board, the program will make duplicate boards for all possible positions. Now there are four boards that are tested with the Tsetlin Machine. ii) For each possible outcome, white win, black win and draw, a Tsetlin Machine uses its clauses to vote on the desired outcome of the game. Positive clauses will if a matching pattern is found vote yes, and negated clauses will vote yes if they find a pattern not matching the desired outcome. iii) All positive and negative votes are summed up and the sum of all positive votes are subtracted the negative votes sum. iv) Based on the predicted outcome the boards are divided into outcome lists. Since this is the black players turn, the boards that vote for black victory is selected and ordered by score.

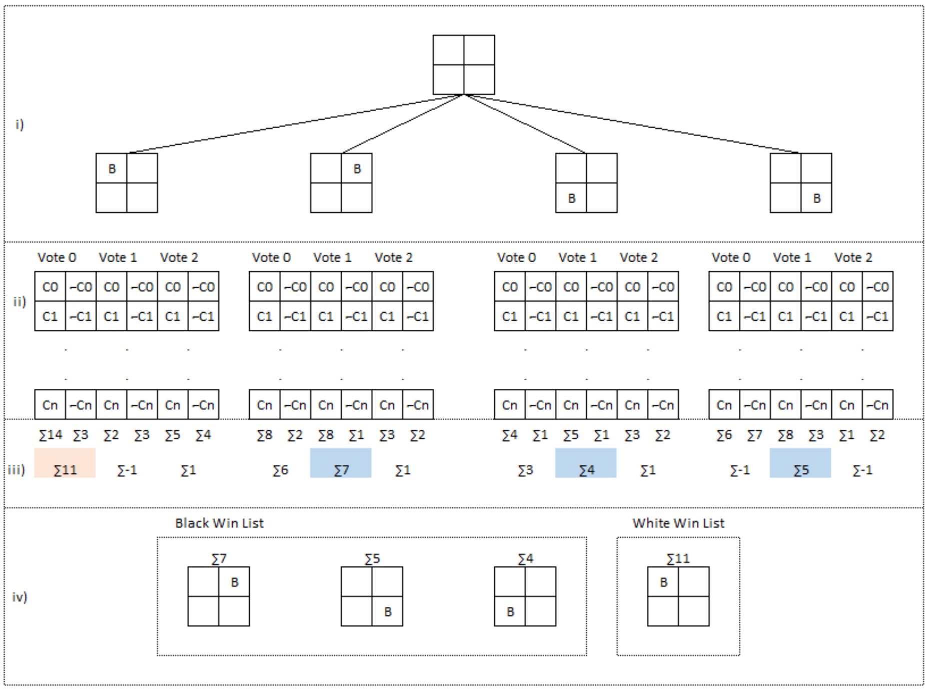


Figure 5.3: Tree search of width 2 and depth 3. #1

Figure 5.4 show the top 2 boards putting a white piece in all open positions. v) Since black TM is here trying to predict what the white player will do, the top boards is selected from the boards where white votes have the highest score. The boards with white victory will be ordered by score and the top 2 will be selected.

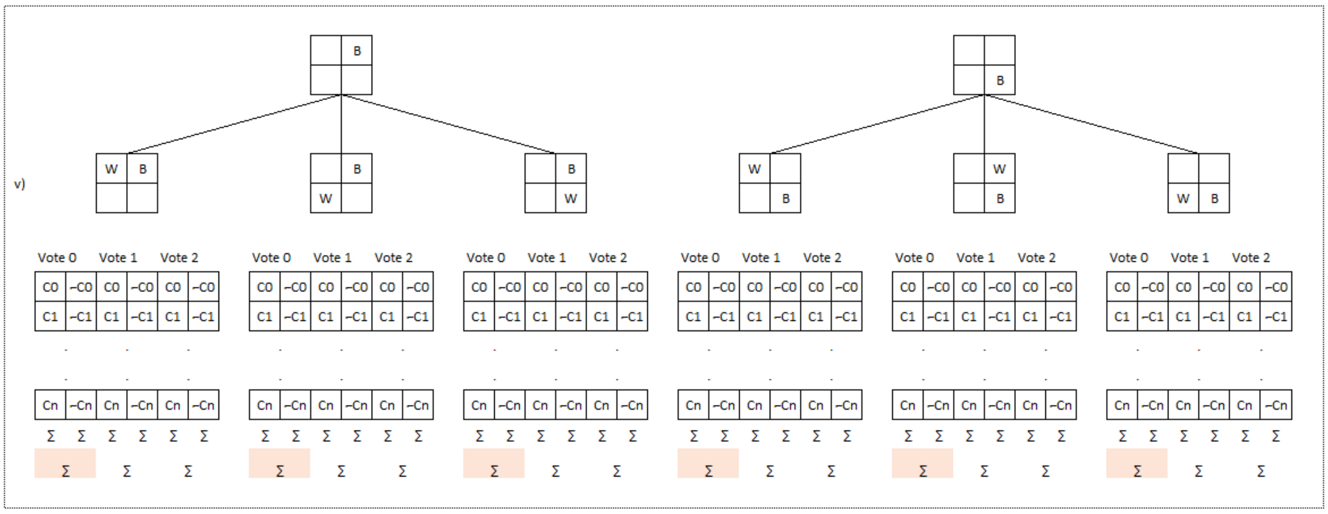


Figure 5.4: Tree search of width 2 and depth 3. #2

In Figure 5.5 again all open positions are given a black piece and each new board will give a prediction along with the score of the prediction. vi) Now all boards are sorted by highest board score with an outcome favorable to black. The board with the highest score will backtrack and find the first move in the 3 move depth tree search. vii) This will be presented as the new board and the turn will be given to white player with the updated board.

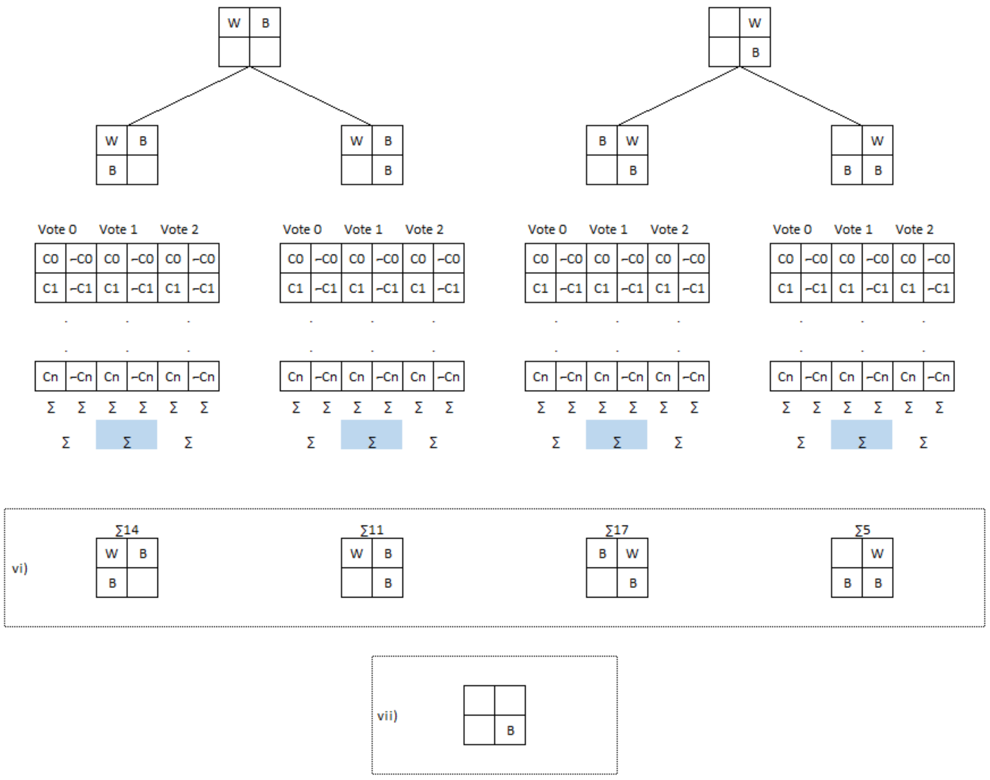


Figure 5.5: Tree search of width 2 and depth 3. #3

The white player will perform the same type of tree search to decide what move to make. If the white player and the black player have different Tsetlin Machines, each player will use their own machines to predict what the other player will do in their tree search.

The recursive function used in tree search and self-play is shown in Algorithm 1. Since this is an recursive function, the end table is a global table in order to collect boards from all the various recursive functions that are started. The board table consist of a number of tables containing detailed information like moves performed, score for each move and much more which makes it possible to backtrack and find the first move in the best board when the tree search is finished.

Algorithm 1 Recursive

```
Global End Table: $eT$
Input Board: $bwt$ Player: $p$ Size of board: $s$ Depth: $m$ Search Width: $w$ Initial Result: $iR$
Output Board: $bwtable$
 1: procedure RecursiveSearch($bwt, p, s, m, w, iR$)
 2:     if $m = 0$ then $eT.append(bwt)$                ▷ Save boards at the final stage
 3:     if $m = 0$ then return $bwt$                     ▷ After depth is reached return tree search
 4:     $m- = 1$                                         ▷ Count down the depth to make sure children will not go to deep
 5:     $newBoards \leftarrow$ FindEmpty(bwt,p,s,w,iR)   ▷ Play all possible moves, predict and evaluate,
 6:                                                      ▷ get back $w$ top boards
 7:     for $i \leftarrow$newBoards do                   ▷ Start a recursive function checking next move.
 8:         if $player = "B"$ then $nplayer \leftarrow$"W"   ▷ Change next player
 9:         if $player = "W"$ then $nplayer \leftarrow$"B"   ▷ Change next player
10:         $i.append($RecursiveSearch$(i, nplayer, s, m, w, iR)$   ▷ Start a new recursive search for each board
11:     end for
12:     $bwt.append(newBoards)$                          ▷ Adding boards from children to parent
13:     return $bwt$                                     ▷ Return tree search to parent recursive function
14: end procedure
```

A count down will ensure that the children will receive one less depth, ensuring that the tree search will eventually stop after reaching the desired depth *d.* All possible moves are explored in the FindEmpty function, and generates new boards. Each board with a new move is evaluated by the Tsetlin Machine, and based on this score the best boards will be sent back to *newBoards.* The amount of boards sent back is decided by the search width *w.* Within the recursive function the next player is altered and each board starts a new recursive function with altered player and one less depth. The input board gets the new boards appended, and with this being a recursive function the first board will at the end contain all the other boards in a tree.

After the parent node is finished and have returned the results to the main function, the main function will choose the best move, alter the original board, change the player and start a recursive function for the next player with the new board.

5.4 Clauses and weights

In Figure 5.6 an illustration of how clauses and weights are gathered from the Tsetlin Machine. i) The illustrated TM have two outcomes 0 or 1, meaning two classes, in essence one TM for each outcome. ii) Class 0 have a number of clauses, in a 2x2 board, each clause will consist of sixteen digits. The first eight digits are positive black and white digits, and the next eight are negative black and white digits.They are grouped into positive black, positive white, negative black and negative white. iii) In order to get the first symbol in the pattern, the first digit in each group is grouped together giving us 1000. The second symbol in the pattern would be the second digit in each group. iv) Each combination of numbers make up an unique symbol. And let us transcribe the bits into a readable symbol. v) 1000 is transcribed into +B.

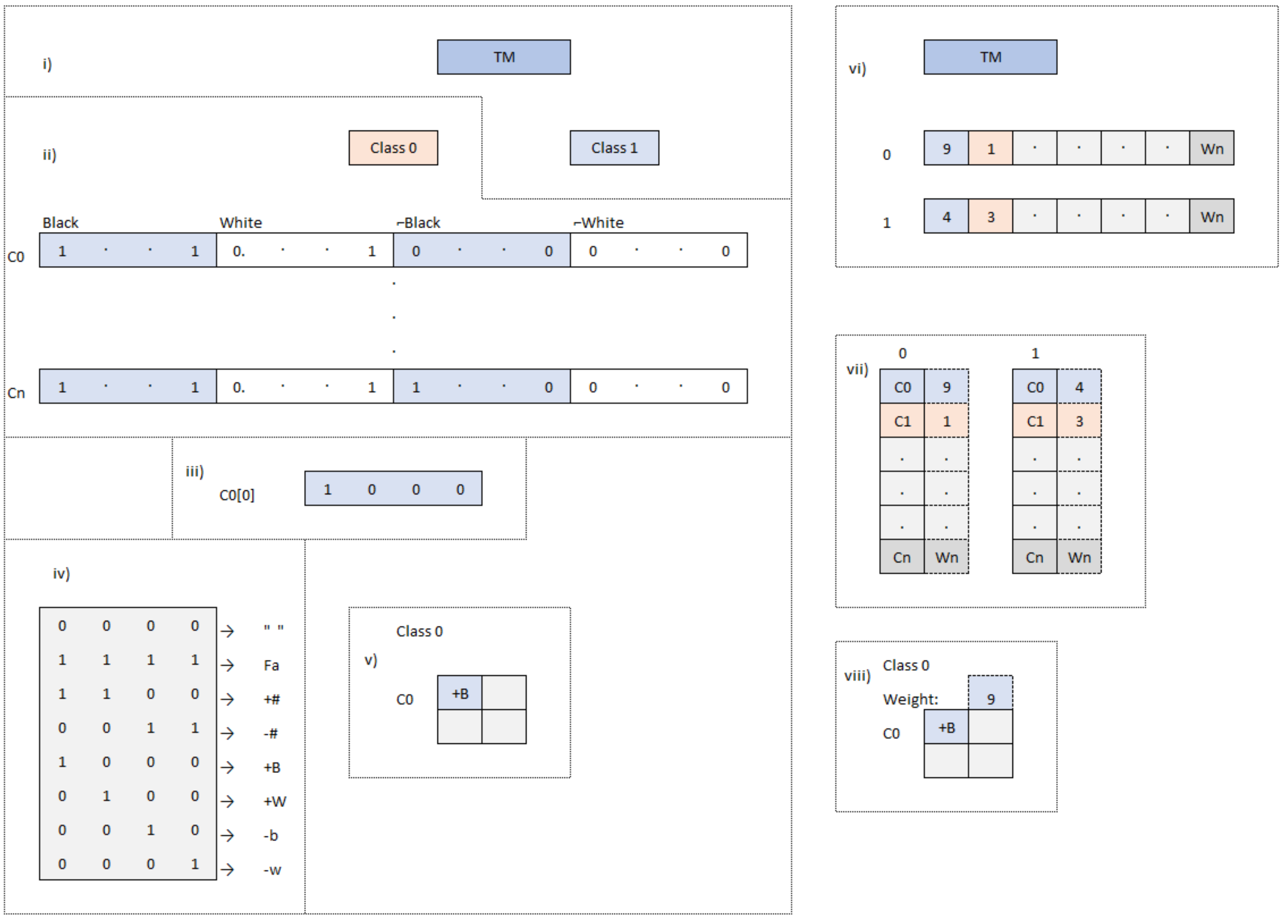


Figure 5.6: Illustration showing how clauses and weights are collected from the Tsetlin Machine.

vi) Gathering the weights from the TM using getState function, provides us with all weights for the various clauses in each class. vii) The weights are in the same order as the clauses making assigning the clauses to the weights easy. viii) The clause have now been assigned a weight, and its true voting power can be counted.

Part III

Experiments, Results and Discussion

Chapter 6

Finding optimal Tsetlin settings

6.1 Introduction

Finding the optimal settings with the Tsetlin Machine can be problematic because of the many variables. An approach to this would be to divide the problem into smaller sub-problems. Testing different variables under different circumstances should yield results. To avoid biased results, 10-fold Cross-Validation was used on all experiments in this chapter.

6.2 Clauses with Tsetlin Machine (TM)

In Figure 6.1 a graph of testing different amounts of clauses under the same threshold and s conditions are shown.

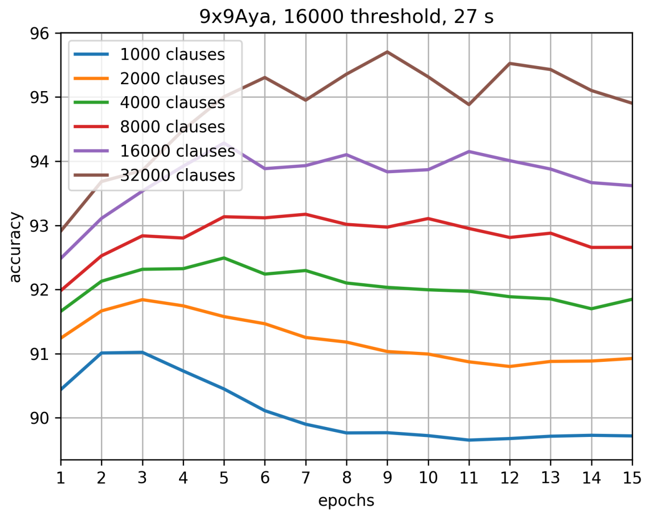


Figure 6.1: Testing 9x9 dataset using TM with different clauses and 16,000 threshold.

Table 6.1 compares a selection of epochs from each experiment, showing the accuracy and standard deviation of the epochs. A higher amount of clauses generally give better accuracy. 32,000 clauses perform better by a good margin. Going much higher in clauses was decided against since each increase in clauses takes a high toll on the TM speed.

Clauses	Epoch 1	Epoch 5	Epoch 9	Epoch 13	Epoch 15
1000	90.44%±0.24%	90.45%±0.12%	89.77%±0.22%	89.71%±0.35%	89.72%±0.16%
2000	91.24%±0.26%	91.58%±0.08%	91.03%±0.09%	90.88%±0.18%	90.93%±0.05%
4000	91.66%±0.15%	92.49%±0.23%	92.03%±0.15%	91.85%±0.15%	91.85%±0.12%
8000	91.98%±0.16%	93.13%±0.31%	92.97%±0.28%	92.88%±0.34%	92.66%±0.43%
16000	92.48%±0.13%	94.28%±0.40%	93.84%±0.60%	93.88%±0.39%	93.62%±0.41%
32000	92.91%±0.32%	95.00%±0.61%	95.70%±0.52%	95.43%±0.53%	94.90%±0.49%

Table 6.1: Testing 9x9 dataset using TM with different clauses and 16,000 threshold.

6.3 Thresholds with Tsetlin Machine (TM)

Moving forward with 32,000 clauses, a number of different thresholds was tested. From Figure 6.2 and Table 6.2 one can see that 8,000 thresholds perform better over 15 epochs. However, 16,000 thresholds had the highest peak, at epoch 9. Also, from the graph it would appear as 64,000, 128,000 and 256,000 is still climbing, although less rapidly. It is possible that over another 15 epochs the higher thresholds could surpass the lower ones.

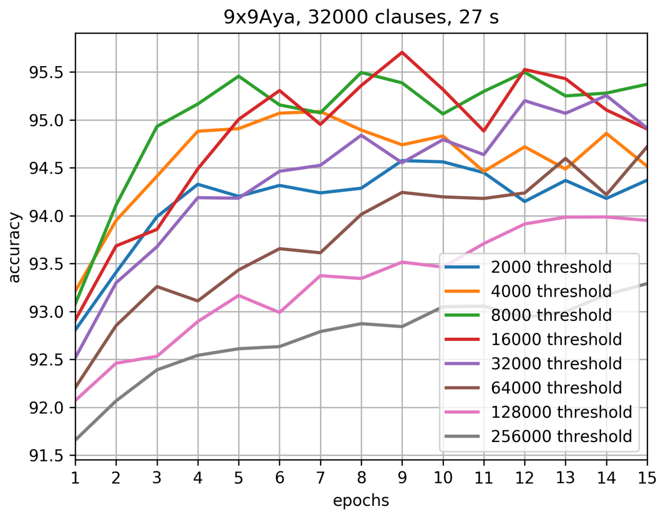


Figure 6.2: Testing 9x9 dataset using TM with 32,000 clauses and different thresholds.

Threshold	Epoch 1	Epoch 5	Epoch 9	Epoch 13	Epoch 15
2000	92.81%±0.25%	94.20%±0.52%	94.57%±0.28%	94.37%±0.52%	94.37%±0.66%
4000	93.21%±0.39%	94.91%±0.69%	94.74%±0.56%	94.49%±0.57%	94.51%±0.51%
8000	93.08%±0.27%	95.46%±0.49%	95.39%±0.51%	95.25%±0.76%	95.37%±0.68%
16000	92.91%±0.32%	95.00%±0.61%	95.70%±0.52%	95.43%±0.53%	94.90%±0.49%
32000	92.52%±0.24%	94.18%±0.53%	94.55%±0.65%	95.07%±0.69%	94.91%±0.56%
64000	92.20%±0.21%	93.43%±0.58%	94.24%±0.49%	94.60%±0.77%	94.72%±0.85%
128000	92.07%±0.12%	93.17%±0.57%	93.51%±0.52%	93.98%±0.84%	93.95%±0.60%
256000	91.66%±0.23%	92.61%±0.17%	92.84%±0.22%	93.00%±0.28%	93.29%±0.40%

Table 6.2: Testing 9x9 dataset using TM with 32,000 clauses and different thresholds.

6.4 Hyperparameter s with Tsetlin Machine (TM)

Figure 6.3 shows the graph of the testing of the s hyperparameter. The lowest at 10 clearly stood out as a poor choice as it flattened right away and does not seem to explore much. Having tested with s at 27.0 up until now, a few test subjects performing slightly better was interesting.

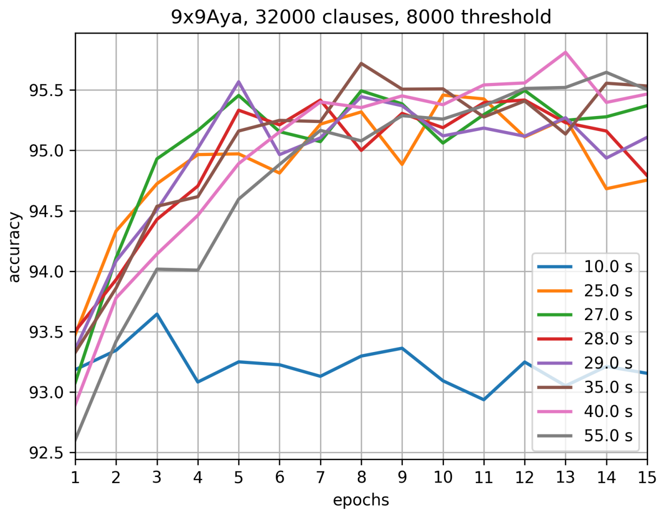


Figure 6.3: Testing 9x9 dataset using TM with 32,000 clauses 8,000 thresholds with different s values.

In Table 6.3 the mean and the standard deviation are shown from the experiment. 40.0 s have the highest k-Fold peak, although it is possible that 55.0 s would have a higher peak if more epochs had been added 40.0 s was chosen as the currently known best setting for s.

S	Epoch 1	Epoch 5	Epoch 9	Epoch 13	Epoch 15
10.0	93.18%±0.48%	93.25%±0.69%	93.36%±0.49%	93.05%±0.47%	93.15%±0.62%
25.0	93.48%±0.69%	94.97%±0.70%	94.88%±0.52%	95.26%±0.72%	94.75%±0.71%
27.0	93.08%±0.27%	95.46%±0.49%	95.39%±0.51%	95.25%±0.76%	95.37%±0.68%
28.0	93.50%±0.30%	95.33%±0.61%	95.31%±0.75%	95.23%±0.52%	94.79%±0.72%
29.0	93.35%±0.47%	95.57%±0.55%	95.37%±0.45%	95.27%±0.58%	95.11%±0.35%
35.0	93.33%±0.50%	95.16%±0.68%	95.51%±0.59%	95.14%±0.81%	95.53%±0.48%
40.0	92.89%±0.39%	94.89%±0.63%	95.45%±0.57%	95.81%±0.45%	95.47%±0.38%
55.0	92.60%±0.37%	94.60%±0.35%	95.29%±0.55%	95.52%±0.47%	95.50%±0.35%

Table 6.3: Testing 9x9 dataset using TM with 32,000 clauses 8,000 thresholds with different s values.

6.5 Window size with Convolutional Tsetlin Machine(CTM)

Figure 6.4 show the graph of the Convolutional Tsetlin Machine using the most optimal settings found from the experiments with the Tsetlin Machine. Testing different windows sizes is here used to find out if the CTM can achieve additional success due to its ability to use windows and thereby finding smaller patterns.

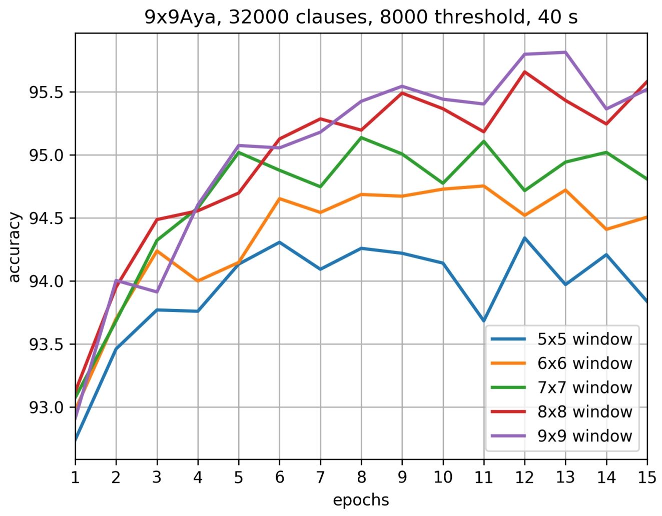


Figure 6.4: Testing 9x9 dataset using CTM with 32,000 clauses, 8,000 thresholds, 40s under different window sizes.

Table 6.4 list the accuracy and standard deviation of the experiment. Smaller windows having less accuracy than the larger windows, indicate that there is little to gain by experimenting with the CTM further.

Window	Epoch 1	Epoch 5	Epoch 10	Epoch 15	Last 10 Epoch
5x5	92.74%±0.27%	94.13%±0.48%	94.14%±0.60%	93.84%±0.71%	94.11%
6x6	92.98%±0.19%	94.15%±0.48%	94.73%±0.73%	94.51%±0.48%	94.62%
7x7	93.07%±0.35%	95.02%±0.29%	94.78%±0.47%	94.81%±0.60%	94.91%
8x8	93.12%±0.37%	94.70%±0.63%	95.37%±0.38%	95.58%±0.54%	95.36%
9x9	92.91%±0.36%	95.08%±0.66%	95.44%±0.67%	95.52%±0.72%	95.46%

Table 6.4: Testing 9x9 dataset using CTM with 32,000 clauses, 8,000 thresholds, 40s under different window sizes.

6.6 Discussion - Optimal Settings

To find the optimal settings a systematical approach was used. Starting with presumably good settings, each setting was explored further. Starting with testing clauses, it was found that higher clauses seemed to improve the Tsetlin Machine, it is possible that there is a correlation with how big the dataset is and how many clauses is the max. The machine gets slower the more clauses it has however and it was decided to use 32,000 clauses and explore further with the other settings. A number of thresholds was tested but 8,000 threshold seemed to be slightly better and was selected for further tests. From Figure 6.2 a slightly higher peak was seen with 16,000 threshold at epoch 9, however 8,000 threshold as it seemed slightly more stable. Moving on to the s parameter, this is the parameter that decides how much exploring the Tsetlin Machine will do. Too low values leads to the machine not exploring enough, too high and it becomes unstable. Quite a few s values showed great promise, however 40.0 was chosen as it had a very steady climb along with the highest peak.

The settings found with the Tsetlin Machine was also tested with a Convolutional Tsetlin Machine using different windows. There had been hopes that smaller windows would be able to find universal patterns and therefore that a smaller window than 9x9 would produce the best accuracy. However 9x9 was found to be overall better, with the CTM being slightly slower, this made the CTM unsuitable for this project, and the non-convolutional TM was chosen for the rest of the research. It is possibly that the nature of the game makes it less suitable for the window driven CTM as the game focus on taking control of the board which usually means edge to edge connection to establish the biggest border. Small windows might not give sufficient information to build the clauses.

Chapter 7

Classification with Tsetlin Machine at a given position

7.1 Introduction

In section 4.1 Figure 4.2 showed the distribution of boards based on the amount of moves performed in the board. The distribution showed that approximately half of all the boards was finished at 92-94 moves. After 80 moves there are 26,305 boards that are finished, which is equal to 1.15% of the total amount, 2,283,679, of boards in the dataset. A dataset with 90 moves have almost a 6% chance of being solved. Using 80 moves would lead to less chance of a concluded board, but since the purpose is to predict the next moves, a board when one side already have won goes against the purpose. By playing 80 moves of a board that have a move-threshold of 90 will give a dataset with suitable boards that have not yet been concluded. A move-threshold is not the same as the hyper-parameter called threshold mentioned earlier in the report. Using this move-threshold means that only the games that have at least Y moves in the board will be used, with X moves being played.

7.2 Accuracy distribution over different completed moves

In this experiment the goal was to look at the accuracy for different datasets. The datasets have different moves completed and some also have a move-threshold. All the tests were done with the same hyper-parameters over the same amount of epochs. The parameters were chosen from what gave best results in the previous experiments in chapter 6. 10-fold Cross-Validation was used on all the results.

In Figure 7.1 and Table 7.1 different datasets of various completion and size are tested with the optimal settings.

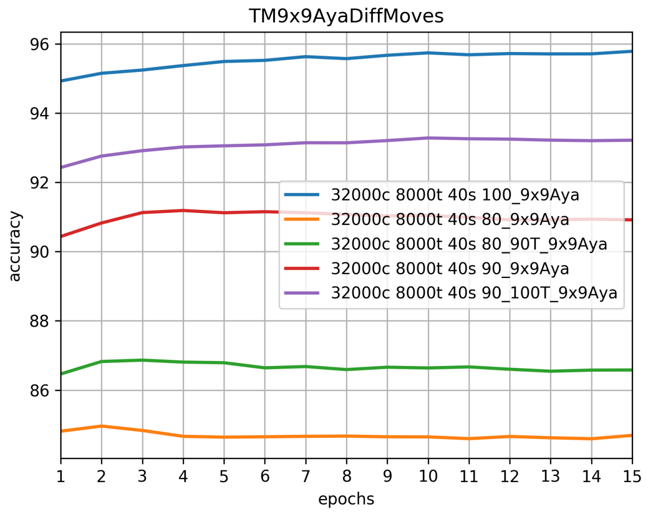


Figure 7.1: Testing 9x9 dataset using TM with different moves completed and with move-threshold.

Moves Completed	Epoch 1	Epoch 5	Epoch 10	Epoch 15	Last 10 Epoch
100_9x9Aya	94.93%±0.13%	95.49%±0.18%	95.74%±0.08%	95.78%±0.1%	95.67%
80_9x9Aya	84.81%±0.18%	84.64%±0.15%	84.65%±0.12%	84.69%±0.1%	84.64%
80_90T_9x9Aya	86.47%±0.1%	86.79%±0.1%	86.64%±0.12%	86.58%±0.19%	86.62%
90_9x9Aya	90.43%±0.21%	91.12%±0.1%	91.05%±0.07%	90.92%±0.1%	91.01%
90_100T_9x9Aya	92.43%±0.11%	93.05%±0.1%	93.28%±0.08%	93.22%±0.07%	93.2%

Table 7.1: Testing 9x9 dataset using TM with different moves completed and with move-threshold.

"100_9x9Aya" stands for 100 moves completed with the 9x9Aya original dataset, while "80_90T_9x9Aya" means 80 moves completed with a 90 move-threshold on the 9x9Aya original dataset.

From the results you are able to see that the higher the amount of moves completed on the board, the higher the accuracy. This is most likely due to the fact that when the amount of moves completed is increased results in more boards being finished. This is why there are also datasets with move-threshold. For instance if you look at 80 moves completed with 90 threshold. The accuracy is lower than the one that only uses 90 moves, but it is still higher than the one just using 80 moves. The datasets with 100 moves does also contain less boards than the one with 90, and same with 90 and 80. Due to the difference in the size of the datasets the various versions could achieve higher accuracy if the hyper-parameters were optimized separately.

7.3 Clauses impact on accuracy at a given position

From testing with lower clauses on a dataset using move-threshold there were small changes to the accuracy. Experimenting with very low clauses gives surprisingly high performance, which could be due to the similarities the same amount of moves brings to pattern recognition. When gathering all the results, there was used 10-fold Cross-Validation.

In Figure 7.2 and Table 7.2, you can see that the results from the last 10 epochs does not vary too much with the increase of clauses. This means that you can still get high accuracy from using smaller amount of clauses, which again will drastically reduce the time required to train the Tsetlin Machine. Due to the close range of accuracy, the versions running with lower clauses can be used with the Tree Search in the later experiments since it gives high enough accuracy, but it is much quicker to run than the ones with high amount of clauses. As when running Tree Search, the time it takes when searching is also a factor to think about.

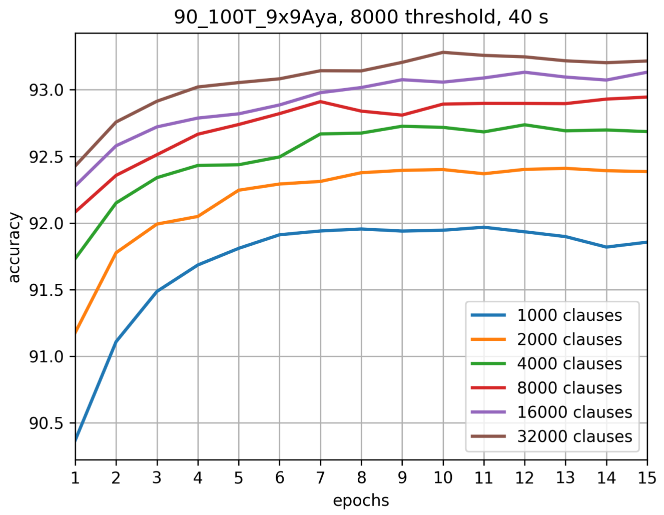


Figure 7.2: Testing 9x9 dataset using TM with 90 moves and 100 move threshold with different amount of clauses.

Clauses	Epoch 1	Epoch 5	Epoch 10	Epoch 15	Last 10 Epoch
1000	90.37%±0.12%	91.81%±0.1%	91.95%±0.12%	91.86%±0.1%	91.92%
2000	91.18%±0.11%	92.25%±0.09%	92.4%±0.12%	92.39%±0.09%	92.37%
4000	91.73%±0.06%	92.44%±0.12%	92.72%±0.07%	92.69%±0.11%	92.68%
8000	92.08%±0.09%	92.74%±0.09%	92.89%±0.11%	92.95%±0.09%	92.88%
16000	92.28%±0.08%	92.82%±0.2%	93.06%±0.1%	93.13%±0.11%	93.05%
32000	92.43%±0.11%	93.05%±0.1%	93.28%±0.08%	93.22%±0.07%	93.2%

Table 7.2: Testing 9x9 dataset using TM with 90 moves and 100 move threshold with different amount of clauses.

Chapter 8

Tree Search

8.1 Tree Search on partly played games

To avoid full boards and games that have already ended, a dataset where we only performed 90 moves on boards that contained 100 moves or more was used. The set was played with 1000 clauses, 8000 threshold and an s at 80.0. Figure 8.1 and Table 8.1 show the training results of the machine used in this experiment.

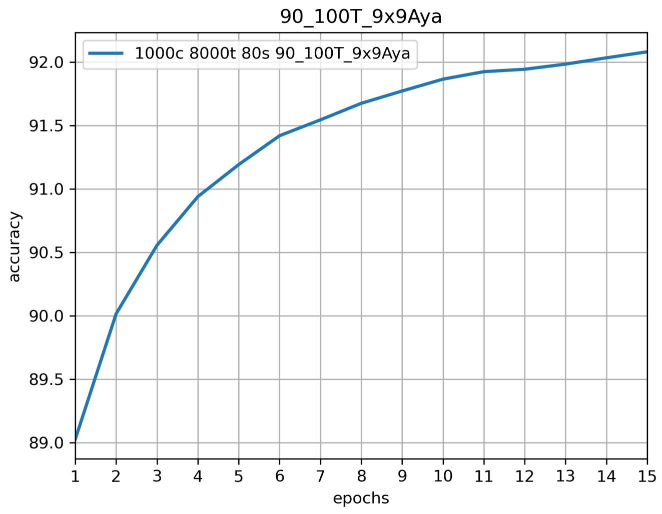


Figure 8.1: Data set with 90 moves performed from game with 100 moves or more.

Settings	Epoch 1	Epoch 5	Epoch 10	Epoch 15	Last 10 Epoch
1000/8000/80	89.03%±0.13%	91.19%±0.12%	91.86%±0.1%	92.08%±0.09%	91.82%

Table 8.1: Data set with 90 moves performed from game with 100 moves or more.

Tree search expand and increases the computations needed greatly for each level of depth as can be seen in Figure 8.2. The computations will include predictions and based on width and depth the calculations will increase significantly.

Computations	Width				
Depth	1	2	3	4	5
1	1	2	3	4	5
2	2	6	12	20	30
3	3	14	39	84	155
4	4	30	120	340	780
5	5	62	363	1364	3905
6	6	126	1092	5460	19530
7	7	254	3279	21844	97655
8	8	510	9840	87380	488280
9	9	1022	29523	349524	2441405

Figure 8.2: Computations needed based on width and depth.

In addition to the number shown all empty moves are predicted on a new board, and causes some extra calculations depending on how many available moves are on the board.

The initial board at 90 moves is tested both with prediction from the Tsetlin Machine and the actual current board area score. A tree search depth at 7 and a width of 3 was selected for this experiment, meaning black will perform 4 moves and white will perform 3 moves. All possible moves are performed and the new boards are evaluated by the Tsetlin Machine. The boards are divided into lists based on their outcome and the tree search select from the most desirable outcome list for the current player making the move. From this list the 3 boards with highest prediction score is selected as this indicates high agreement among the clauses. A tree search on the 3 selected boards is then performed. In Figure 8.3 the end result is checked with the prediction from the Tsetlin Machine.

A very high accuracy rate can be seen for black win while white win start with less accuracy at start but slowly climbs on the even numbered moves which is when white makes a move, both sides improve as more moves are performed.

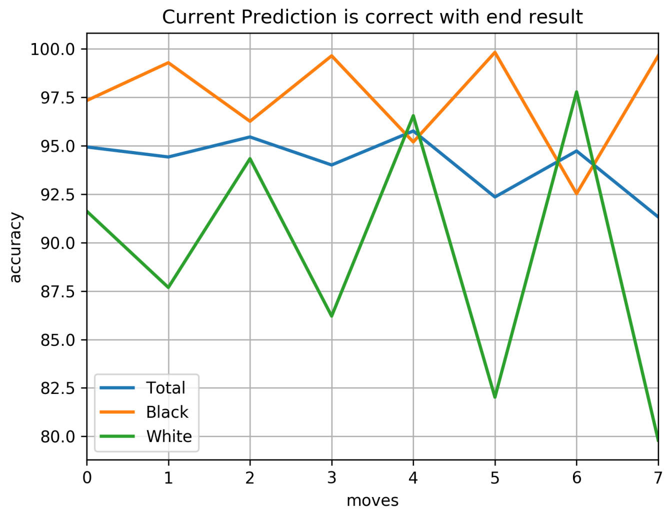


Figure 8.3: Current Prediction is correct with end result

In this experiment draw was omitted due to being very unlikely as it has to be exactly zero, while anything above or below zero can be classified as black or white win. Of 1,000 games 32 draw was ignored, leaving a total of 968, with 406 white wins and 562 black wins. In Table 8.2 the number of correct predictions for black and white player is shown. On odd numbers black player makes a move and on even numbers the white player makes a move. Looking at odd numbers and even numbers an increase in the accuracy for the player making a move can be seen.

Outcome	Initial 0	B Move 1	W Move 2	B Move 3	W Move 4	B Move 5	W Move 6	B Move 7
Correct B	547	558	541	560	535	561	520	560
Correct W	372	356	383	350	392	333	397	324

Table 8.2: Current Prediction is correct with end result

In Figure 8.4 and Figure 8.5 the end result is checked with the actual board area score both with and without the komi rule that starts white with 7 extra points.

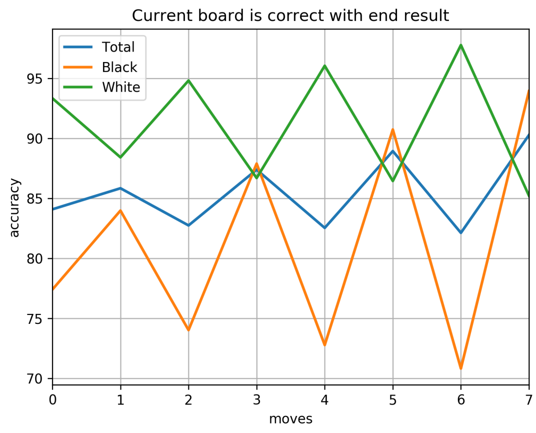


Figure 8.4: Current board is correct with end result

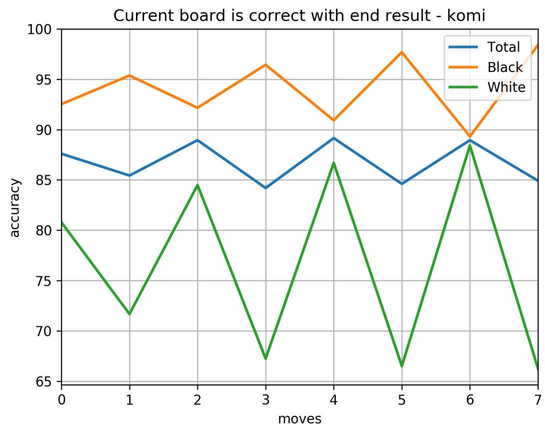


Figure 8.5: Current board is correct with end result - komi

Table 8.3 show the number of correct predictions on odd numbers for the board when the actual board is evaluated by Go rules. Table 8.4 show the same evaluation when we also deduct the komi extra points. We can clearly see that the Tsetlin Machine have been trained based on white having a score advantage.

Outcome	Initial 0	B Move 1	W Move 2	B Move 3	W Move 4	B Move 5	W Move 6	B Move 7
Correct B	435	472	416	494	409	510	398	528
Correct W	379	359	385	352	390	351	397	346

Table 8.3: Current board is correct with end result

Outcome	Initial 0	B Move 1	W Move 2	B Move 3	W Move 4	B Move 5	W Move 6	B Move 7
Correct B	520	536	518	542	511	549	502	553
Correct W	328	191	343	273	352	270	359	269

Table 8.4: Current board is correct with end result - komi

Measuring the precision shows how many of the selected elements are relevant, the formula

$$\texttt{Precision} = \frac{\texttt{True Positive}}{\texttt{True Positive} + \texttt{False Positive}}$$

will give an understanding of how accurate the positive votes are. While recall using the formula

$$\texttt{Recall} = \frac{\texttt{True Positive}}{\texttt{True Positive} + \texttt{False Negative}}$$

will show how good the machine is to find all the relevant elements. The f-measure is used to measure the performance and is the harmonic mean of precision and recall.

$$\texttt{F-measure} = 2 * \frac{\texttt{Precision} * \texttt{Recall}}{\texttt{Precision} + \texttt{Recall}}$$

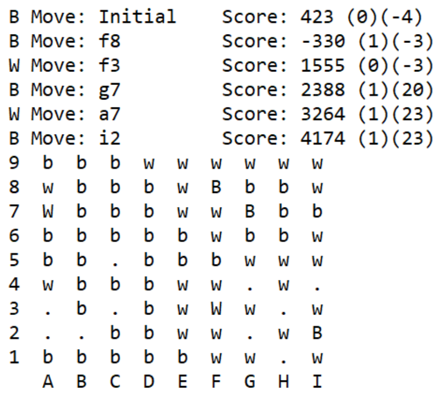
In Table 8.5 the precision, recall and f-measure for each depth in the tree search have been calculated.

Precision	Initial 0	B Move 1	W Move 2	B Move 3	W Move 4	B Move 5	W Move 6	B Move 7
White	96.12 %	98.89 %	94.80 %	99.43 %	93.56 %	99.70 %	90.43 %	99.39 %
Black	94.15 %	91.78 %	95.92 %	90.91 %	97.45 %	88.49 %	98.30 %	87.23 %
Total	95.14 %	95.33 %	95.36 %	95.17 %	95.50 %	94.09 %	94.37 %	93.31 %
Recall	**Initial 0**	**B Move 1**	**W Move 2**	**B Move 3**	**W Move 4**	**B Move 5**	**W Move 6**	**B Move 7**
White	91.63 %	87.68 %	94.33 %	86.21 %	96.55 %	82.02 %	97.78 %	79.80 %
Black	97.33 %	99.29 %	96.26 %	99.64 %	95.20 %	99.82 %	92.53 %	99.64 %
Total	94.48 %	93.49 %	95.30 %	92.93 %	95.87 %	90.92 %	95.15 %	89.72 %
F-measure	**Initial 0**	**B Move 1**	**W Move 2**	**B Move 3**	**W Move 4**	**B Move 5**	**W Move 6**	**B Move 7**
White	93.82 %	92.95 %	94.57 %	92.35 %	95.03 %	90.00 %	93.96 %	88.52 %
Black	95.71 %	95.38 %	96.09 %	95.08 %	96.31 %	93.81 %	95.33 %	93.02 %
Total	94.77 %	94.17 %	95.33 %	93.71 %	95.67 %	91.91 %	94.64 %	90.77 %

Table 8.5: Precision, Recall and F-measure for each depth in the tree search

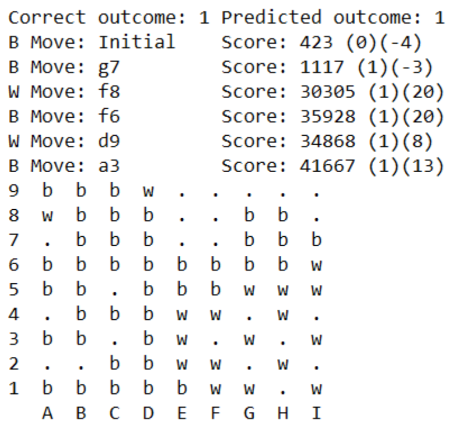
Interestingly the precision of the white player and the black player increases when the other player have performed a move. And decreases when the players own move have been performed. This is most likely because the machines detect that the other player have gained a small advantage and the positive votes are less likely to happen. If we compare the odd moves of where black make a move and even moves where white makes a move, we can see that white have a slightly higher precision.

The Tsetlin Machine have been trained on boards that have been altered based on area score, meaning the pieces will disappear if captured. In Figure 8.6 an example of a board that is not altered through gomill before getting a prediction is shown. Gomill is in this board only used to feed back area score. In Figure 8.7 the same board is used. Here we get a board as well as area score back from the gomill library. The prediction is substantially stronger when the board has been altered according to the Go rules, although Figure 8.6 does show that even without the alteration of the board the Tsetlin Machine can still predict the right result, albeit not as confident as can be seen by the lower score.



```
B Move: Initial    Score: 423 (0)(-4)
B Move: f8         Score: -330 (1)(-3)
W Move: f3         Score: 1555 (0)(-3)
B Move: g7         Score: 2388 (1)(20)
W Move: a7         Score: 3264 (1)(23)
B Move: i2         Score: 4174 (1)(23)
9  b  b  b  w  w  w  w  w  w
8  w  b  b  b  w  B  b  b  w
7  W  b  b  b  w  w  B  b  b
6  b  b  b  b  b  w  b  b  w
5  b  b  .  b  b  b  w  w  w
4  w  b  b  b  w  w  .  w  .
3  .  b  .  b  w  W  w  .  w
2  .  .  b  b  w  w  .  w  B
1  b  b  b  b  b  w  w  .  w
   A  B  C  D  E  F  G  H  I
```

Figure 8.6: Example of an unaltered board

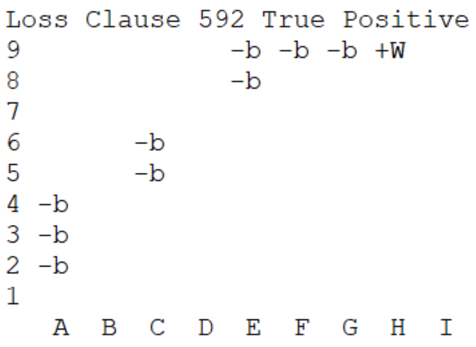


```
Correct outcome: 1 Predicted outcome: 1
B Move: Initial    Score: 423 (0)(-4)
B Move: g7         Score: 1117 (1)(-3)
W Move: f8         Score: 30305 (1)(20)
B Move: f6         Score: 35928 (1)(20)
W Move: d9         Score: 34868 (1)(8)
B Move: a3         Score: 41667 (1)(13)
9  b  b  b  w  .  .  .  .  .
8  w  b  b  b  .  .  b  b  .
7  .  b  b  b  .  .  b  b  b
6  b  b  b  b  b  b  b  b  w
5  b  b  .  b  b  b  w  w  w
4  .  b  b  b  w  w  .  w  .
3  b  b  .  b  w  .  w  .  w
2  .  .  b  b  w  w  .  w  .
1  b  b  b  b  b  w  w  .  w
   A  B  C  D  E  F  G  H  I
```

Figure 8.7: Example of an altered board

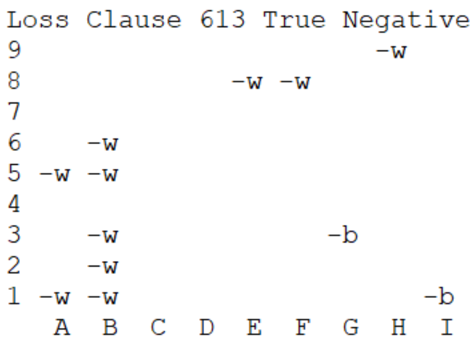
8.1.1 Evaluating clauses in Tree Search on partly played games

A strength of the Tsetlin Machine is the interpretability of the clauses. The clauses can be easily translated to show how the clauses look like and thereby what the clause reacts on and why it vote yes or no to a given position. During the experiment chapter 8 all clauses was counted every time they made a correct prediction and every time they made a wrong prediction. Adding them with their weights we can examine why some clauses vote correctly while others vote incorrect. The clauses generate patterns that they try to find on the board when casting their votes. The clauses are weighted which will give each clause a different weight when they vote. In all clauses shown, the clauses have had their weights added showing their true voting power. In Figure 8.8 the highest weighted clause that vote for a white win is shown, and in Figure 8.9 the highest weighted clause that vote against a white win is shown. Clause 592 will vote for a white win if there are no black pieces in the position noted with ”-b” and it needs a white piece in the position noted with ”+W”. If the negative clause 613 finds that there are no white pieces in the position noted with ”-w” and no black pieces in the position noted with ”-b” it will vote against this being a winning board for white.



```
Loss Clause 592 True Positive
9             -b -b -b +W
8             -b
7
6       -b
5       -b
4 -b
3 -b
2 -b
1
   A  B  C  D  E  F  G  H  I
```

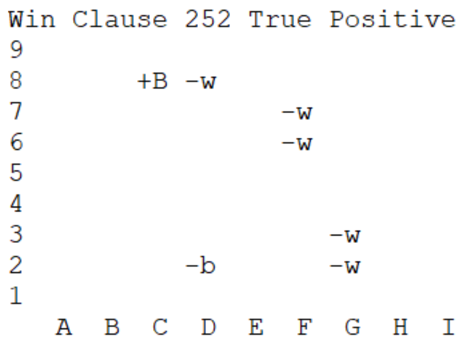
Figure 8.8: TM2 Clause 592TP Highest Weighted



```
Loss Clause 613 True Negative
9                       -w
8              -w -w
7
6    -w
5 -w -w
4
3    -w             -b
2    -w
1 -w -w                   -b
   A  B  C  D  E  F  G  H  I
```

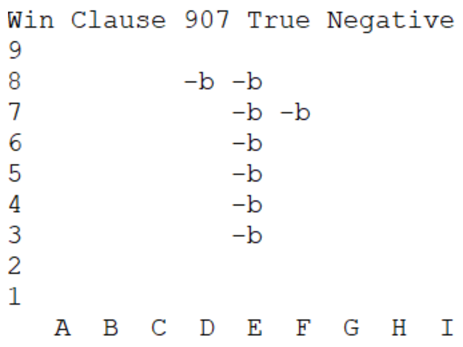
Figure 8.9: TM2 Clause 613TN Highest Weighted

In Figure 8.10 and Figure 8.11 the highest black weighted positive and negative clauses are shown. They will vote for or against a black win.



```
Win Clause 252 True Positive
9
8       +B -w
7                -w
6                -w
5
4
3                   -w
2          -b       -w
1
   A  B  C  D  E  F  G  H  I
```

Figure 8.10: TM2 Clause 252TP Highest Weighted

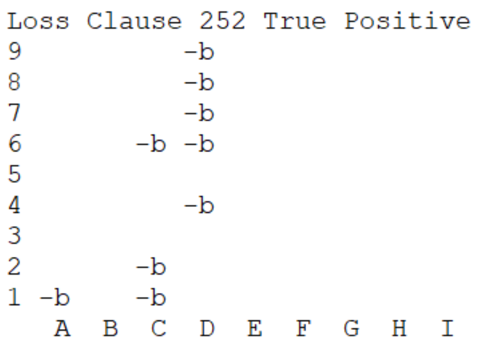


```
Win Clause 907 True Negative
9
8          -b -b
7             -b -b
6             -b
5             -b
4             -b
3             -b
2
1
   A  B  C  D  E  F  G  H  I
```

Figure 8.11: TM2 Clause 907TN Highest Weighted

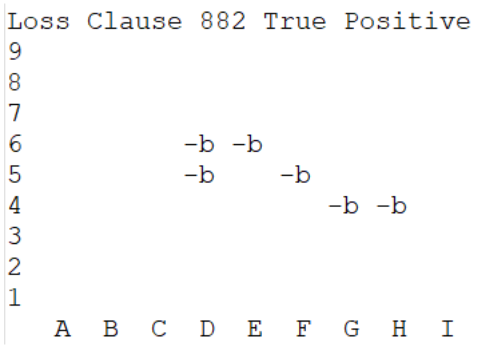
The clauses was given weight during the training of the Tsetlin Machine, and the highest ones was presumably the best during the constant retraining of the machine during the epochs. However when performing the tree search each clause was counted when they voted correct and false. Allowing us to see the true positive, false positive, true negative and false negative clauses. Giving us more insight in how the clauses are built up and why certain clauses perform well while others fail.

In Figure 8.12 and Figure 8.13 the two clauses voting for a white win correctly most often when weights are counted is shown. It is worth noting that the clauses only focuses on where black pieces should not be and not where white pieces should be.



```
Loss Clause 252 True Positive
9          -b
8          -b
7          -b
6       -b -b
5
4          -b
3
2       -b
1 -b    -b
   A  B  C  D  E  F  G  H  I
```

Figure 8.12: TM2 Clause 252TP



```
Loss Clause 882 True Positive
9
8
7
6          -b -b
5          -b    -b
4                   -b -b
3
2
1
   A  B  C  D  E  F  G  H  I
```

Figure 8.13: TM2 Clause 882TP

In Figure 8.14 and Figure 8.15 the two clauses that mistakenly votes for a white win most is shown. Their pattern is more spread than the true positive clauses, and probably are triggered more times while the true positive clauses look for a dangerous pattern when giving their votes.

```
Loss Clause 470 False Positive
9       -b    -b
8
7                      +B
6
5
4 -b
3    -b -b
2
1                      -w -w
   A  B  C  D  E  F  G  H  I
```

Figure 8.14: TM2 Clause 470FP

```
Loss Clause 870 False Positive
9                -b
8       -b                -b
7                      -b
6
5
4             +W
3    -w
2
1
   A  B  C  D  E  F  G  H  I
```

Figure 8.15: TM2 Clause 870FP

The white clauses voting negative correct the most times are shown in Figure 8.16 and Figure 8.17. Both show a clear pattern which they find important for a white win, and lacking this pattern in the board will lead to a vote against a white win.

```
Loss Clause 593 True Negative
9
8          -w -w
7          -w
6                -w
5                -w
4                -w
3                -w
2                   -w
1
   A  B  C  D  E  F  G  H  I
```

Figure 8.16: TM2 Clause 593TN

```
Loss Clause 41 True Negative
9
8
7 -w
6       -w    -w    -w    -w
5 -w    -w -w
4
3
2
1
   A  B  C  D  E  F  G  H  I
```

Figure 8.17: TM2 Clause 41TN

In Figure 8.18 and Figure 8.19 the white negative clauses that voted most times against a white win falsely is shown. Their pattern is weaker than the best true negative clauses and much like with the false positive white win clauses they also look for certain pieces on the board, and a similarity can be seen in the white clauses that vote positive incorrect.

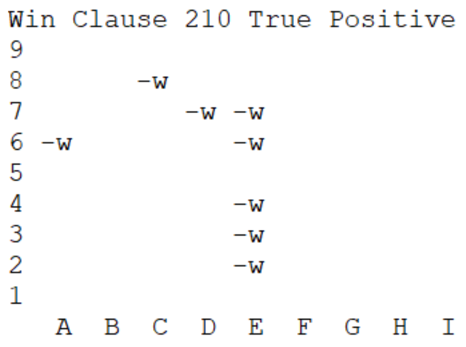
```
Loss Clause 379 False Negative
9                         -w
8                -w
7
6
5
4                      -w
3                         -w
2       +W             -w
1                -w       -w
   A  B  C  D  E  F  G  H  I
```

Figure 8.18: TM2 Clause 379FN

```
Loss Clause 101 False Negative
9
8
7          -w
6
5             -w -w
4                      +B
3
2                         -w
1                         -w
   A  B  C  D  E  F  G  H  I
```

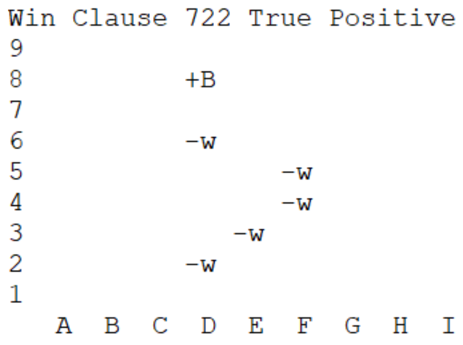
Figure 8.19: TM2 Clause 101FN

Looking closer at the black clauses we can see the clauses that vote correctly for a black win the most in Figure 8.20 and Figure 8.21. Similar to the white true positive clauses, the black clauses look for a pattern played by the other player which it seeks to avoid.



```
Win Clause 210 True Positive
9
8       -w
7          -w -w
6 -w          -w
5
4             -w
3             -w
2             -w
1
   A  B  C  D  E  F  G  H  I
```

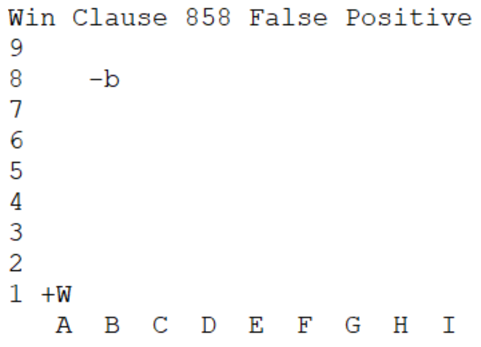
Figure 8.20: TM2 Clause 210TP



```
Win Clause 722 True Positive
9
8          +B
7
6          -w
5                -w
4                -w
3             -w
2          -w
1
   A  B  C  D  E  F  G  H  I
```

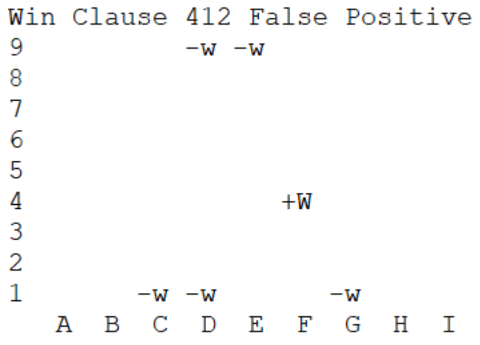
Figure 8.21: TM2 Clause 722TP

The highest false positive clauses are shown in Figure 8.22 and Figure 8.23. They both have a very weak pattern, with clause 858 only having 2 symbols which means it can trigger very often and it is not certain having a white piece in the corner will lead to a black win, although it could be speculated that white placing a piece in the corner is a bad move. Clause 412 have white pieces along the edges which also is a weak pattern that would not automatically lead to a black victory.



```
Win Clause 858 False Positive
9
8    -b
7
6
5
4
3
2
1 +W
   A  B  C  D  E  F  G  H  I
```

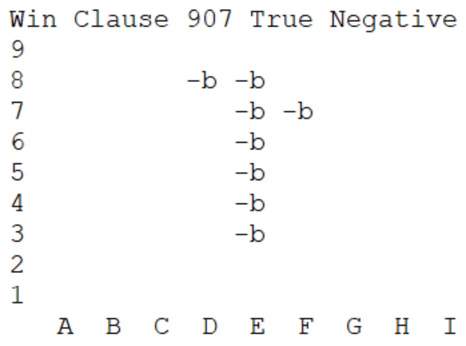
Figure 8.22: TM2 Clause 858FP



```
Win Clause 412 False Positive
9          -w -w
8
7
6
5
4                +W
3
2
1       -w -w       -w
   A  B  C  D  E  F  G  H  I
```

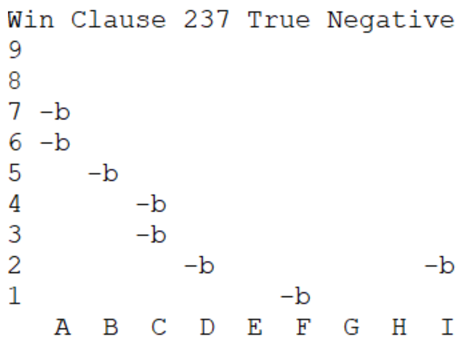
Figure 8.23: TM2 Clause 412FP

The true negative clauses shown in Figure 8.24 and Figure 8.25 show the clauses that correctly voted against a black win most. They both show clear black patterns that would be important for a black win, and when lacking those patterns in the board they vote against. Clause 907 is also the highest weighted clause from the training of the machine.



```
Win Clause 907 True Negative
9
8          -b -b
7             -b -b
6             -b
5             -b
4             -b
3             -b
2
1
   A  B  C  D  E  F  G  H  I
```

Figure 8.24: TM2 Clause 907TN



```
Win Clause 237 True Negative
9
8
7 -b
6 -b
5    -b
4       -b
3       -b
2          -b             -b
1                -b
   A  B  C  D  E  F  G  H  I
```

Figure 8.25: TM2 Clause 237TN

The false negative clauses shown in Figure 8.26 and Figure 8.27 voted against a black win wrongly the most times. The symbol ”-#” means that there should not be anything in this position. The false negative clauses lack a strong pattern and have very spread symbols.

```
Win Clause 505 False Negative
9                         -b
8             -b -b
7 -w    +B       -b
6
5                         -b
4 -w
3                    -b
2
1
   A  B  C  D  E  F  G  H  I
```

Figure 8.26: TM2 Clause 505FN

```
Win Clause 97 False Negative
9       -w
8
7
6
5
4 -b
3                -b
2
1    -b -#          -b
   A  B  C  D  E  F  G  H  I
```

Figure 8.27: TM2 Clause 97FN

Both the positive and negative clauses for each player share a similarity that the correct clauses contain strong and clear patterns. A bit surprisingly the strongest positive clauses looks for patterns the opposing player should not have and the strongest negative clauses would vote against if the board did not contain the players own pieces in a strong pattern. The clauses that voted most often wrong was weaker than the stronger, with less symbols and lacking a clear pattern.

8.2 Self-play with the Tsetlin Machine

Table 8.6 show the trained Tsetlin Machines used for self-play with their hyper-parameters and accuracy. TM1, TM3 and TM4 have been trained using the complete dataset while TM2 is trained using the 90_100T dataset.

Name	Clauses	Threshold	S	Prediction Accuracy
TM1	26000	16000	27.0	95.08
TM2	1000	8000	80.0	91.98
TM3	1000	16000	27.0	89.71
TM4	20000	4000	27.0	94.08

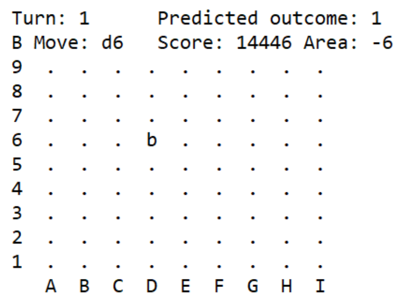
Table 8.6: Trained Tsetlin Machines used for self-play.

Table 8.7 show the results of self-play experiments among the machines and also random play. The TM2 trained with 90_100T did not get more matches due to under performing against random. In general white seems stronger, with the presumably weaker machines being able to win a few more matches when facing a black stronger machine. A high number of clauses have shown to increase accuracy greatly, however threshold did appear to have a maximum. The machine with the highest amount of clauses, TM1 did not perform well against TM4 which had slightly less clauses. It is possible that the combination of high clauses and high threshold can produce weaker results in self-play. TM3 which had a low amount of clauses and a high threshold performed well against TM4 could mean that a low amount of clauses along with a high threshold could produce stronger clauses.

Black Machine	Black Win	White Win	White Machine
TM1	10	0	Random
TM2	5	5	Random
TM3	8	2	Random
Random	0	10	TM3
TM3	1	9	TM4
TM4	6	4	TM3
TM1	0	10	TM4
TM4	8	2	TM1

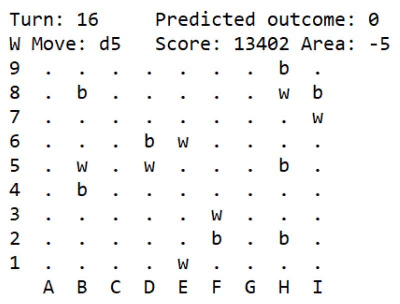
Table 8.7: Results of self-play after 70 moves

Figure 8.28 show the presumed optimal move for black player after a tree search at depth 3 and width 3. The picture show the turn played, the predicted outcome, the color and the move performed, the Tsetlin Machine score of the prediction and the actual current score given by gomill analysis of the board. Figure 8.29 show turn 16, where white TM makes a move it predicts can lead to a win, with a relatively strong score from the Tsetlin Machine.



```
Turn: 1     Predicted outcome: 1
B Move: d6  Score: 14446 Area: -6
9  .  .  .  .  .  .  .  .  .
8  .  .  .  .  .  .  .  .  .
7  .  .  .  .  .  .  .  .  .
6  .  .  .  b  .  .  .  .  .
5  .  .  .  .  .  .  .  .  .
4  .  .  .  .  .  .  .  .  .
3  .  .  .  .  .  .  .  .  .
2  .  .  .  .  .  .  .  .  .
1  .  .  .  .  .  .  .  .  .
   A  B  C  D  E  F  G  H  I
```

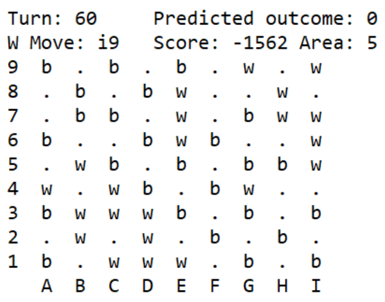
Figure 8.28: TM4 vs TM3 game 7 turn 1



```
Turn: 16    Predicted outcome: 0
W Move: d5  Score: 13402 Area: -5
9  .  .  .  .  .  .  .  b  .
8  .  b  .  .  .  .  .  w  b
7  .  .  .  .  .  .  .  .  w
6  .  .  .  b  w  .  .  .  .
5  .  w  .  w  .  .  .  b  .
4  .  b  .  .  .  .  .  .  .
3  .  .  .  .  .  w  .  .  .
2  .  .  .  .  .  b  .  b  .
1  .  .  .  .  w  .  .  .  .
   A  B  C  D  E  F  G  H  I
```

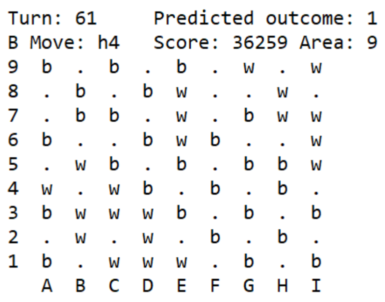
Figure 8.29: TM4 vs TM3 game 7 turn 16

Figure 8.30 show the last turn where the white Tsetlin Machine predicts a win for himself, note that the area score give black player the lead, and the negative score for the white player means that there are more clauses that votes against a win rather than for a win, meaning the Tsetlin Machine have little faith in a future win. In Figure 8.31 black player makes his move, captures a white piece, along with gaining more dominance on the board. The score shows an increase in confidence in a victory.



```
Turn: 60     Predicted outcome: 0
W Move: i9   Score: -1562 Area: 5
9  b  .  b  .  b  .  w  .  w
8  .  b  .  b  w  .  .  w  .
7  .  b  b  .  w  .  b  w  w
6  b  .  .  b  w  b  .  .  w
5  .  w  b  .  b  .  b  b  w
4  w  .  w  b  .  b  w  .  .
3  b  w  w  w  b  .  b  .  b
2  .  w  .  w  .  b  .  b  .
1  b  .  w  w  w  .  b  .  b
   A  B  C  D  E  F  G  H  I
```

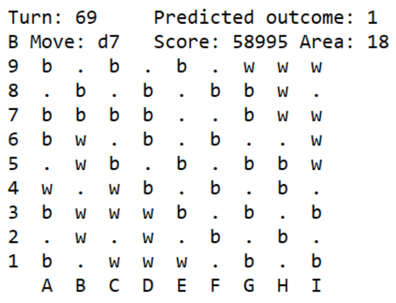
Figure 8.30: TM4 vs TM3 game 7 turn 60



```
Turn: 61     Predicted outcome: 1
B Move: h4   Score: 36259 Area: 9
9  b  .  b  .  b  .  w  .  w
8  .  b  .  b  w  .  .  w  .
7  .  b  b  .  w  .  b  w  w
6  b  .  .  b  w  b  .  .  w
5  .  w  b  .  b  .  b  b  w
4  w  .  w  b  .  b  .  b  .
3  b  w  w  w  b  .  b  .  b
2  .  w  .  w  .  b  .  b  .
1  b  .  w  w  w  .  b  .  b
   A  B  C  D  E  F  G  H  I
```

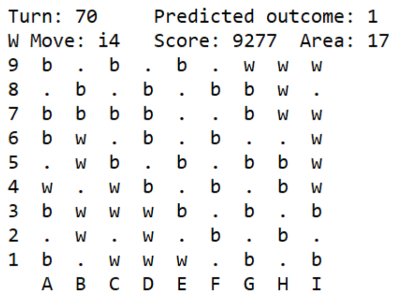
Figure 8.31: TM4 vs TM3 game 7 turn 61

Figure 8.32 shows black player place a piece at d7, which finished an encirclement of 4 white pieces at e6,e7,e8 and f7, thereby removing them from the board and increasing area score along with Tsetlin score. Figure 8.33 shows that now the white Tsetlin Machine is becoming relatively sure on a black win. Out of all the moves the white machines tree search tested no move that would lead to a white win or a draw was found. The move shown at i4 is the one that is least likely to give black a win, it does predict a black win, but it is the smallest score in the black prediction list.



```
Turn: 69     Predicted outcome: 1
B Move: d7   Score: 58995 Area: 18
9  b  .  b  .  b  .  w  w  w
8  .  b  .  b  .  b  b  w  .
7  b  b  b  b  .  .  b  w  w
6  b  w  .  b  .  b  .  .  w
5  .  w  b  .  b  .  b  b  w
4  w  .  w  b  .  b  .  b  .
3  b  w  w  w  b  .  b  .  b
2  .  w  .  w  .  b  .  b  .
1  b  .  w  w  w  .  b  .  b
   A  B  C  D  E  F  G  H  I
```

Figure 8.32: TM4 vs TM3 game 7 turn 69

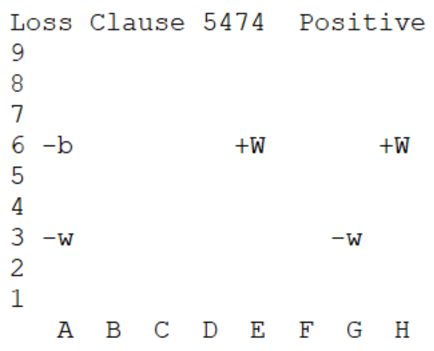


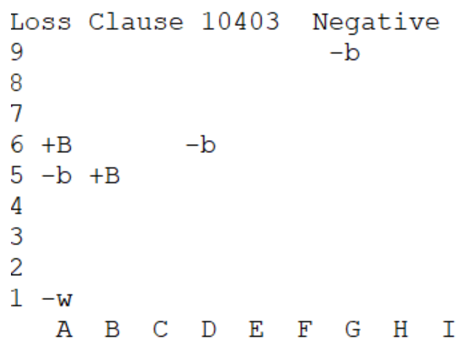
```
Turn: 70     Predicted outcome: 1
W Move: i4   Score: 9277   Area: 17
9  b  .  b  .  b  .  w  w  w
8  .  b  .  b  .  b  b  w  .
7  b  b  b  b  .  .  b  w  w
6  b  w  .  b  .  b  .  .  w
5  .  w  b  .  b  .  b  b  w
4  w  .  w  b  .  b  .  b  w
3  b  w  w  w  b  .  b  .  b
2  .  w  .  w  .  b  .  b  .
1  b  .  w  w  w  .  b  .  b
   A  B  C  D  E  F  G  H  I
```

Figure 8.33: TM4 vs TM3 game 7 turn 70

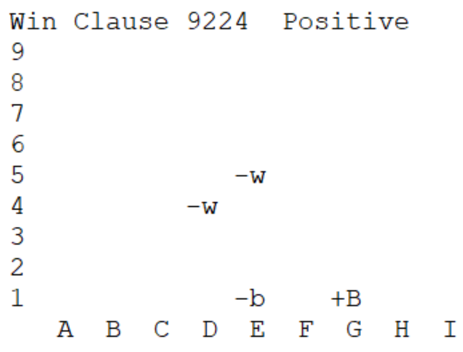
8.2.1 Evaluating clauses in Self-play with the Tsetlin Machine

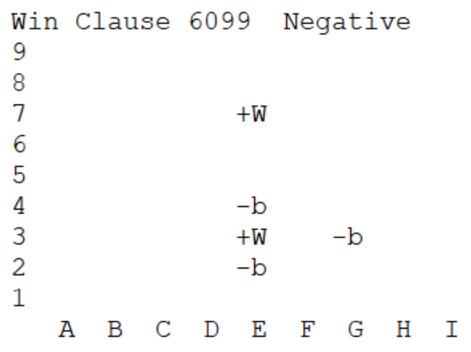
In the self-play match between TM4 and TM3 the clauses was counted when they voted. The clauses was counted in all 10 matches played in the match where TM4 played as black player. The white clauses are from the TM4 and is used to simulate possible white moves when black player is performing tree search. In Figure 8.34 and Figure 8.35 the highest weighted positive and negative clauses for white player is shown.


```
Loss Clause 5474  Positive
9
8
7
6 -b          +W       +W
5
4
3 -w                -w
2
1
   A  B  C  D  E  F  G  H  I
```
Figure 8.34: TM4 Clause 5474P Highest Weighted


```
Loss Clause 10403  Negative
9                   -b
8
7
6 +B       -b
5 -b +B
4
3
2
1 -w
   A  B  C  D  E  F  G  H  I
```
Figure 8.35: TM4 Clause 10403N Highest Weighted

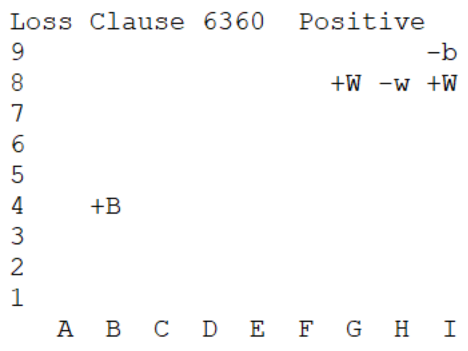
In Figure 8.36 and Figure 8.37 the highest weighted positive and negative clauses are shown for black player.

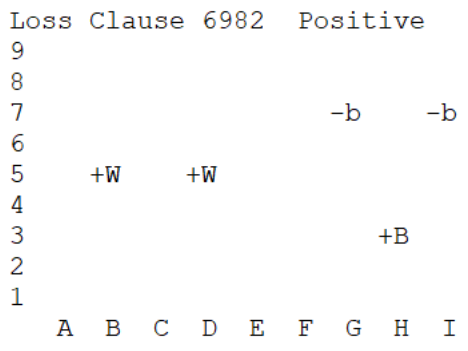

```
Win Clause 9224  Positive
9
8
7
6
5             -w
4          -w
3
2
1             -b    +B
   A  B  C  D  E  F  G  H  I
```
Figure 8.36: TM4 Clause 9224P Highest Weighted


```
Win Clause 6099  Negative
9
8
7             +W
6
5
4             -b
3             +W    -b
2             -b
1
   A  B  C  D  E  F  G  H  I
```
Figure 8.37: TM4 Clause 6099N Highest Weighted

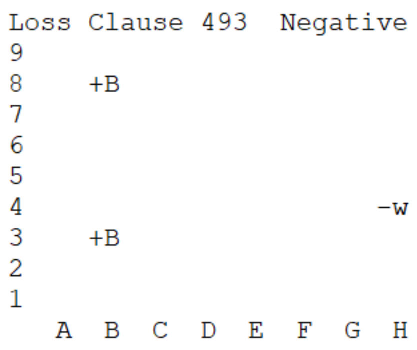
Unlike the clauses in **??**, the highest clauses does not contain a clear pattern. An explanation could be TM4 having 20,000 clauses allow having smaller patterns that together can form a pattern by voting together.

In Figure 8.38 and Figure 8.39 the clauses that voted for a white win the most is shown.


```
Loss Clause 6360  Positive
9                         -b
8                   +W -w +W
7
6
5
4    +B
3
2
1
   A  B  C  D  E  F  G  H  I
```
Figure 8.38: TM4 Clause 6360P1

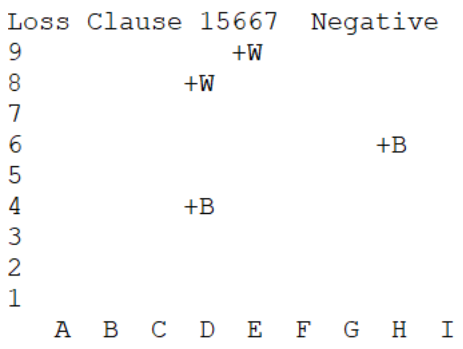

```
Loss Clause 6982  Positive
9
8
7                   -b    -b
6
5    +W    +W
4
3                      +B
2
1
   A  B  C  D  E  F  G  H  I
```
Figure 8.39: TM4 Clause 6982P2

The white clauses that voted against a white win the most times is shown in Figure 8.40 and Figure 8.41.



```
Loss Clause 493  Negative
9
8     +B
7
6
5
4                             -w
3     +B
2
1
   A   B   C   D   E   F   G   H   I
```

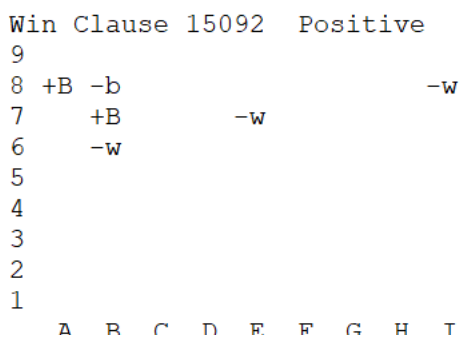
Figure 8.40: TM4 Clause 493N1



```
Loss Clause 15667  Negative
9                 +W
8             +W
7
6                             +B
5
4             +B
3
2
1
   A   B   C   D   E   F   G   H   I
```

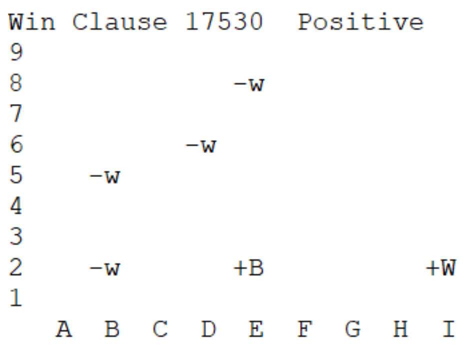
Figure 8.41: TM4 Clause 15667N2

In Figure 8.42 and Figure 8.43 the two highest voting clauses for a black win are shown. Although they do not show a large pattern, a pattern can be seen.



```
Win Clause 15092  Positive
9
8 +B  -b                          -w
7     +B          -w
6     -w
5
4
3
2
1
   A   B   C   D   E   F   G   H   I
```

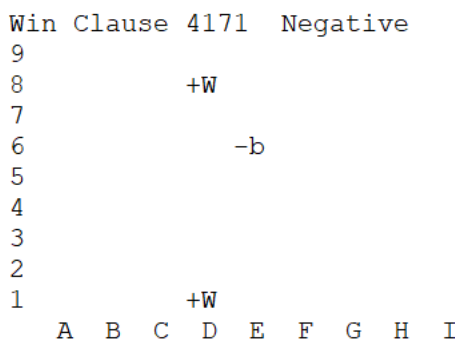
Figure 8.42: TM4 Clause 15092P1



```
Win Clause 17530  Positive
9
8                 -w
7
6             -w
5     -w
4
3
2     -w          +B              +W
1
   A   B   C   D   E   F   G   H   I
```

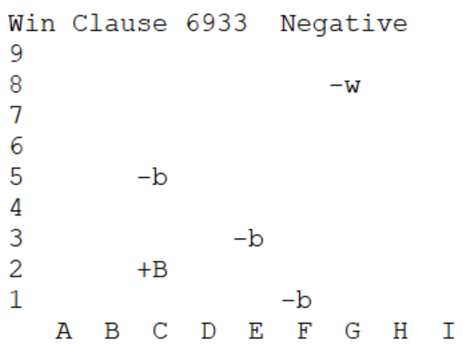
Figure 8.43: TM4 Clause 17530P2

In Figure 8.44 Figure 8.45 the two black clauses that voted most times against a win is shown.



```
Win Clause 4171  Negative
9
8             +W
7
6                 -b
5
4
3
2
1             +W
   A   B   C   D   E   F   G   H   I
```

Figure 8.44: TM4 Clause 4171N2



```
Win Clause 6933  Negative
9
8                         -w
7
6
5         -b
4
3                 -b
2         +B
1                     -b
   A   B   C   D   E   F   G   H   I
```

Figure 8.45: TM4 Clause 6933N1

Unlike in subsection 8.1.1 where the clauses had a larger and more clear pattern, the clauses that was most popular in TM4 had a smaller pattern. This is most likely due to the lower hyper-parameter *s*. When it is higher there is a higher chance for changes within the clauses, so a higher *s* means more literals or symbols.

8.3 Discussion - Tree search on partly played games

As can be seen in Figure 8.3 the Tsetlin Machine managed to improve gradually for every move. However for each time the opponent made a move there was a small dip for the other, which is to be expected since each side will try their best to win. When looking at Figure 8.4 which shows how the board score matches the end result it is still very high, it is also important to remember that the dataset used has performed 90 moves with 10 or more moves yet to be done before the game was finished. Black starting low here is to be expected since white starts with extra points and until winning moves are made white tends to lead midway even if Black have played a more strategic game and wins in the end. Looking at Figure 8.5 where there is no komi, black and white have basically switched places. Without komi white have the disadvantage of starting later and generally suffers from this. When examining the clauses, the highest weighted and most used clauses was focused on avoiding certain patterns from the other player. This could explain why the precision in Table 8.5 was higher for each player when the other player had performed a move. Since the clauses look for pattern to avoid, there is a higher chance of clauses that look for dangerous patterns among the opponents pieces to trigger.

8.4 Discussion - Self-play with the Tsetlin machine

When experimenting with self-play, the trained Tsetlin Machines was tested against an opponent that places pieces randomly. The machine using 90_100T dataset which was used in 8 did not perform well against a random white opponent, and ended with 5 wins and 5 losses. The next attempt was more successful, this was a Tsetlin Machine trained on the full dataset with 26000 clauses 8000 threshold and 27.0 s. Out of 10 games it won all 10 games against a random opponent. Testing a presumably weaker Tsetlin Machine would also be interesting, and TM4 with 20000 clauses, 4000 threshold and the same s value was selected. To be fair each machine got to play both sides against each other. In this case the presumed slightly weaker machine beat the presumed strongest one with more clauses. In section 6.3 an optimal threshold was found to be around 8000, and it is possible that high threshold can lead to certain clauses getting to high weights and leading the machine down the wrong path. TM3 trained on the full dataset with 1000 clauses and 8000 threshold performed slightly better than TM1 with 26000 clauses when playing against TM4 with 20000 clauses.

When evaluating the clauses of TM4, the patterns was different from the clauses found in TM2 in subsection 8.1.1. The patterns found in the clauses in TM4 was smaller than the ones found in TM2. Since TM3 only had 1000 clauses as well and performed better than expected, clauses as the only reason can be excluded. The noticeable difference here is the hyper-parameter s, if it is set to high it leads to higher patterns in the clauses, while a smaller s lead to smaller clauses, and although it is hard to see a clear pattern in the clauses of TM4, several clauses will work together to vote. Smaller patterns will also be triggered more, and can explain why TM2 under performed against an opponent placing pieces randomly. With large patterns, very few clauses will be triggered early in the game and TM2 is practically placing pieces randomly as well.

Part IV

Conclusion and Future works

Chapter 9

Conclusion and Future Work

9.1 Conclusion

The thesis stated that the Tsetlin Machine would be able to predict the outcome of a game by finding patterns in games of Go at various stages of play. The Tsetlin Machine instantly showed high accuracy within classification when testing for optimal settings. All the experiments used k-Fold Cross-Validation to avoid biased results. Multiple settings was analyzed in a very thorough and systematic approach, which lead us to presumed optimal hyper-parameters for this paper which was used further in our experiments. The dataset was large and made the experiments timely but also gave the machine a lot of good data early in the experiments which can be seen by the high accuracy at the first epochs at around 90%. This further climbed up to above 95% upon retraining. The Tsetlin Machine gave consistent high results when testing accuracy from different stages in a game. Thus, the Tsetlin Machine is more than capable to analyze and classify a game like Go.

The second thesis stated that through training the Tsetlin Machine could learn and be able to play the game of Go. Using tree search without an advanced algorithm was decided to ensure that the experiment tested the Tsetlin Machine and not the algorithm. The Tsetlin Machine provided good results on half played games, the precision decreased from 95% to 93% over 7 moves. A decrease in precision indicate that false positives are increasing. It is not a large increase when considering that the dataset was only trained on unfinished games. When evaluating the clauses, strong patterns was found that gave more insight into how the machine works. Examining the machine on this level have given us more insight into how the Tsetlin Machine works.

An important question was also if the machine could learn to play games starting with a empty board. While experimenting with self-play starting on a empty board, white was observed to have the upper hand when playing against black. White TM4 won 9-1 against black TM3, but black TM4 only won 6-4 against white TM3. Examining the clauses of TM4 revealed in this case smaller clauses than those found in TM2. TM2 played 5-5 against a white player making random moves, TM1 with the high number of clauses and higher accuracy was believed to be the stronger machine but it was not strongest in self-play. With the results from self-play, it can be concluded that large patterns is not suited for playing from start. The Tsetlin Machine was able to play complete games of Go using machines trained with a smaller hyper-parameter s, which lead to smaller clauses. We can conclude that the Tsetlin Machine can be trained to play the game of Go, but high accuracy on classification is not a good indication on how strong a Tsetlin Machine will be during self-play. This may indicate that in order to fine tune the assessment of board positions faced during self-play, producing training data directly from self-play may be required. The results and knowledge gained from the experiments in this thesis should provide a benchmark for further research within the field of the Tsetlin Machine and Go.

9.2 Future Work

Classification with Tsetlin Machine at a given position

1. Optimize hyper-parameters individually for each version of the datasets.
2. Do the same experiments, but with Convolutional Tsetlin Machine instead.

By optimizing the hyper-parameters individually one might achieve better results from each of the versions of the datasets. It would also have been interesting to look at how the Convolutional Tsetlin Machine would have performed compared to the normal Tsetlin Machine.

Playing against other AI implementation opponents.

1. Implement already discovered solutions of other algorithms capable of solving Go.
2. Use the other solutions to play against the Tsetlin Machine & Tree Search method.
3. Compare the different solutions.

In this research the focus has been primarily towards the Tsetlin Machine and Tree Search method, but it would be interesting to see how it fares against other algorithms that are already implemented for Go. For instance, by using algorithms from the openspiel library mentioned earlier in the research. To further test the Tsetlin Machine, having it play against other AI implementations could lead to more interesting discoveries in its abilities. It could also lead to more weaknesses and strengths being highlighted.

Combine Tsetlin Machine with Monte Carlo Tree Search

1. Instead of using standard Tree Search, a Monte Carlo Tree Search method would be implemented and combined with the Tsetlin Machine.
2. Compare the Monte Carlo Tree Search method with the normal Tree Search method.
3. Compare the Monte Carlo Tree Search method with other AI solutions for Go.

With a game as complex as Go, the tree search is time consuming. The Monte Carlo Tree Search would make it easier to build a tree of possible moves. Testing if the combination of Monte Carlo Tree Search and Tsetlin Machine would be fruitful and should be a high priority.

Expand the board size

1. When the solution and analyze quality of the 9x9 board is at a high enough point, one can look at increasing the board size to 13x13, and even later 19x19.
2. Gather new datasets for the new board sizes, and process this data to fit with the Tsetlin Machine.
3. Repeat the steps from this research.

A 9x9 board was used, but 13x13 and 19x19 boards are also an interesting aspect. There is no reason to believe that the same results would not be seen, however it would be a completely new dataset and some accuracy alteration should be expected. In general finding patterns in a smaller board should also work on larger boards. This should however be examined. Since the board and dataset size would be different, there would be a need to do another hyper-parameter optimization. The process would then follow the steps established in this research as well as the other already mentioned future works.

Acquire training data from self-play

1. Use the data produced from self-play for training the Tsetlin Machine.

Since the high accuracy on classification did not give a good indication on how strong the Tsetlin Machine will perform during self-play, using the data produced from self-play for training might fine tune the assessment.

References

[1] A Brief History of Go. [Accessed Feb. 2020].

[2] Expanding our knowledge, finding new answers. [Accessed Mar. 2020].

[3] How to play Go. [Accessed Jan. 2020].

[4] Official AGA Rules of Go. [Accessed May. 2020].

[5] Rules of Go. [Accessed May. 2020].

[6] Tree Search. [Accessed May. 2020].

[7] What Is Go? [Accessed Jan. 2020].

[8] cair/pyTsetlinMachineParallel, 11 2019. [Accessed Jan. 2020].

[9] Rafi Akhtar. Made Lapuerta, 09 2019. [Accessed May. 2020].

[10] Arno Hollosi Anders Kierulf, Martin Mueller. Smart Game Format. [Accessed May. 2020].

[11] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M.Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 1*, 3 2012. [Accessed Feb. 2020].

[12] Jason Brownlee. A Gentle Introduction to k-fold Cross-Validation, 05 2018. [Accessed Jan. 2020].

[13] Dae Ryun Chang. AlphaGo and the Limits of Machine Intuition, 03 2016. [Accessed May. 2020].

[14] Ole-Christoffer Granmo. The Tsetlin Machine - A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic. *arXiv*, 4 2018. [Accessed Dec. 2019].

[15] Ole-Christoffer Granmo, Geir Thore Berge, Tor Oddbjørn Tveit, Morten Goodwin, Lei Jiao, and Bernt Viggo Matheussen. Using the Tsetlin Machine to Learn Human-Interpretable Rules for High-Accuracy Text Categorization with Medical Applications. *arXiv*, 9 2018. [Accessed Dec. 2019].

[16] Ole-Christoffer Granmo, Sondre Glimsdal, Lei Jiao, Morten Goodwin, Christian W. Omlin, and Geir Thore Berge. The Convolutional Tsetlin Machine. *arXiv*, 5 2019. [Accessed Dec. 2019].

[17] Prashant Gupta. Search Algorithms in Artificial Intelligence, 11 2017. [Accessed May. 2020].

[18] Hiroshi. Aya and Natsukaze's selfplay games for training value network., 2 2018. [Accessed Jan. 2020].

[19] adapted James Davies, extracted and edited by Fred Hansen. The Chinese Rules of Go. [Accessed May. 2020].

[20] Fred Hansen James Davies, Jonathan Cano. The Japanese Rules of Go. [Accessed May. 2020].

[21] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. *arXiv*, abs/1908.09453, 2019.

[22] Cade Metz. The Sadness and Beauty of Watching Google's AI Play Go, 11 2016. [Accessed May. 2020].

[23] Danielle Muoio. AlphaGo and the Limits of Machine Intuition, 03 2016. [Accessed May. 2020].

[24] Michael Nielsen. Is AlphaGo Really Such a Big Deal?, 03 2016. [Accessed May. 2020].

[25] Adrian Phoulady, Ole-Christoffer Granmo, Saeed Rahimi Gorji, and Hady Ahmady Phoulady. The Weighted Tsetlin Machine: Compressed Representations with Weighted Clauses. *arXiv*, 11 2019. [Accessed Dec. 2019].

[26] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approachs*. Alan Apt, 1995.

[27] Georg Seif. Why and How to do Cross Validation for Machine Learning, 05 2019. [Accessed Feb. 2020].

[28] Sam Shead. Google's Complex Relationship With DeepMind Gets Exposed. [Accessed Mar. 2020].

[29] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 1 2016. [Accessed Feb. 2020].

[30] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas baker, Matthew Lai, Adrian bolton, Yutian chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 10 2017. [Accessed Feb. 2020].

[31] Matthew Woodcraft. gomill 0.8.3. [Accessed May. 2020].

[32] Matthew Woodcraft. gomill 0.8.3. documentation. [Accessed May. 2020].

Appendices

A Source Code

https://github.com/KristofferLM96/TsetlinMachine-GO