UiA University of Agder

Master

The Application of the Tsetlin Machine in Checkers

MARTIN BRÅTEN

SUPERVISOR
Ole-Christoffer Granmo

University of Agder, 2020
Faculty of Engineering and Science
Department of ICT

UiA
University of Agder
Master's thesis

Faculty of Engineering and Science
Department of ICT

Abstract

The Tsetlin Machine has shown promising results in the domain of board games such as Axis & Allies, showing that the Tsetlin Machine may be a contender to more well-known machine learning algorithms. This research aimed to find out how suitable the Tsetlin Machine is for the domain of Checkers, and how to best utilize it for the purpose of playing Checkers. In order to research this, the following questions had to be answered: *Which Tsetlin Machine Configuration is best suited for predicting the results of Checkers games? To what degree can the Tsetlin Machine predict the results of Checkers games? How well can a Checkers player using the Tsetlin Machine compete against Kingsrow?*

These questions were answered by creating a functional Checkers player with a Tsetlin Machine predictor, for both to be tested. This was done by investigating how best to create datasets from a collection of data, which OCA 2.0 was chosen for, testing various configurations of the multiclass Tsetlin Machine with these datasets for finding well-fitted hyper-parameters and to find the most suited Tsetlin Machine configuration for predicting the results of Checkers boards. By self-play, further improvements were made.

The proposed solution was a Checkers player using a tree search with the depth of three and a minimal amount of manually set rules, making it easier to evaluate the proposed Tsetlin Machine predictor. The solution was shown to be the multiclass weighted Tsetlin Machine with positive boost using the hyper-parameters: *clauses:* 19000, *treshold:* 40000 and *s:* 9; which was trained on the third K-Fold of the dataset NoDupeCheck StandardEnd, which was minimally altered from the data source. It utilized a separate Tsetlin Machine predictor for each color, achieving accuracies of 72.21% and 72.12% respectively, and proved to be more accurate in this domain than numerous other machine learning algorithms. The Checkers player itself was not able to compete with Kingsrow, but beat a Checkers player making moves at random and was shown to have the skill comparable with that of a beginner-level human Checkers player.

The code for the proposed solution can be accessed through the following URL: `https://github.com/Fiskesuppen/TsetlinMachine-Checkers`

KEY WORDS: Machine learning, Multiclass Tsetlin Machine, American Checkers, English Draughts, Kingsrow, OCA 2.0, Tree search .

Table of Contents

Glossary

Python Python is a programming language, and is the programming language mainly used in this research. 12

Self-play The act of playing against yourself. In this paper, it refers to a Checkers player playing against itself.. xiii, 5, 30, 45, 67, 68, 72

List of Figures

List of Tables

Part I

Research Overview

Chapter 1

Introduction

This chapter explains the goal of this research, and what method is going to be used for reaching this goal. Related research questions and hypotheses are also listed, as well as a short overview of this paper.

1.1 Motivation

The Tsetlin Machine is an emerging machine learning algorithm which has proven to be suitable for playing games such as Axis & Allies, achieving better predictions of the datasets in games than Logistic Regression; as well as Naive Bayes in Axis & Allies in data sparse environments [3]. This show that the Tsetlin Machine has potential which should be investigated further and in more contexts, such as the game Checkers. This research investigate how well the Tsetlin Machine may perform in this environment compared to other machine learning implementations as well as how to best utilize the data available.

1.2 Thesis definition

The goal of this research is to find out how suitable the Tsetlin Machine is for predicting the results of Checkers games, how suitable the Tsetlin Machine is for playing the game both against human opponents and existing machine Checkers players as well as how to make the Tsetlin Machine play Checkers the best it possibly can. In order to investigate this, the following research questions were made as well as ways to measure the results. Hypotheses corresponding to the research questions, which were mostly based on earlier experiences with the Tsetlin Machine, is listed further below.

1.2.1 Research Questions

Research question 1: Which Tsetlin Machine configuration is best suited for predicting the results of Checkers games?

-Measured by thorough testing of multiple configurations of the Tsetlin Machine against a small collection of unseen test data.

Research question 2: To what degree can the Tsetlin Machine predict the results of Checkers games?

-Measured by testing the capabilities of the Tsetlin Machine by making it predict unseen test data, and evaluating its accuracy for each of the three possible outcomes of a Checkers match.

Research question 3: How well can a Checkers player using the Tsetlin Machine compete against Kingsrow?

-Measured by having the Tsetlin Machine play Checkers against Kingsrow, then compare the outcomes after numerous matches.

1.2.2 Hypotheses

Hypothesis 1: The Weighted Tsetlin Machine will be the most suitable configuration for predicting the results of Checkers games as this configuration has proven its strength in other board games previously [3]. Although, the Weighted Convolutional Tsetlin Machine might prove itself a strong contender.

Hypothesis 2: The Tsetlin Machine will be able to predict the results of Checkers games correctly most of the time, and will also be a better Checkers player than a human non-professional Checkers player.

Hypothesis 3: The Checkers player using the Tsetlin Machine will not have a positive win ratio over Kingsrow, which is well-refined as well as being likely to have access to an enormous dataset. Although, if Kingsrow's dataset is available and rather small; the Tsetlin Machine may have a chance to be able to compete with Kingsrow.

1.3 Contributions

This paper investigates the Tsetlin Machine's potential as a Checkers player and how best to utilize this machine learning algorithm in this domain. The findings of this paper should be suitable for being adapted for domains similar to Checkers. The contributions of this paper is summarized as follows:

- The most suitable Tsetlin Machine configuration for this domain is explained and demonstrated, as well as how this configuration was found.
- How best to transform the available data into dataset compositions allowing the Tsetlin Machine to classify Checkers boards as accurately as possible.
- A method of "adaptive epochs" allowing the training of the Tsetlin Machine to stop when no general increase in accuracy is demonstrated, which simplify the parameter testing of the Tsetlin Machine allowing researchers to set a maximum amount of training epochs limiting the need for retraining with an increased number of epochs and prevents the Tsetlin Machine from over-training.

1.4 Method

The research process started by studying how other machine Checkers players function and finding a source of data. It further progressed by finding the way of best utilizing the data found and systematically test various Tsetlin Machine configurations for these and finding well-fitted hyper-parameters. Then the best way of training a Tsetlin Machine predictor was investigated. The next step was to find a suitable way of utilizing this Tsetlin Machine predictor in a Checkers player. After this; more testing was performed, enabled by the use of self-play which was used to find ways to improve the Checkers player. After this, testing was commenced in order to evaluate the proposed solution. The Tsetlin Machine predictor was compared to other machine learning algorithms and the Checkers player was compared to Checkers players of varying skill levels; ranging from a Checkers player picking moves at random to Kingsrow.

Even though this research propose a Checkers player, it had a heavy focus on researching the capability of the Tsetlin Machine itself. Therefore; the tree search of the Checkers player was kept very simple. If a more advanced tree search such as Monte Carlo Tree Search was to be used; it would be more difficult, albeit not impossible, to research the use of the Tsetlin Machine itself in this domain [6]. This is also the reason for the Checkers player not including more man-made rules than what was proposed.

1.5 Thesis outline

This research propose the following in the domain of Checkers: a Tsetlin Machine predictor with well-fitted hyper-parameters, using the optimal configuration and well-fitted hyper-parameters, and a Checkers player which incorporate this predictor. The structure of this paper, along with its content, is listed below:

- Research overview
  - Background
    - Relevant background is presented and explained. This includes various methods used as well a the game of Checkers.
  - State-of-the-art
    - Existing solutions and tools are presented. This includes Kingsrow and other machine Checkers players.
- Contributions
  - Proposed Solutions
    - All that which together is the proposed solution is presented; both what these are, how they function and how these were created. This includes the dataset compositions, the Tsetlin Machine predictors and the Checkers player itself.
- Experiments and Results
  - Tests and Findings of the Tsetlin Machine
    - Comparisons and tests of the various Tsetlin Machines are presented. These tests and comparisons were performed both for finding the solution, as well as validating it.
  - Tests and Findings of the Checkers Player
    - Comparisons and tests of the Checkers player are presented; both tests of various versions of itself, but also against other Checkers players.
  - Alternative Machine Learning Algorithms
    - The proposed Tsetlin Machine configuration are compared to other machine learning algorithms for the dataset compositions investigated.
  - Conclusion and Future Work
    - The conclusion of this research is presented, in addition to improvements that could or should have been made to further improve both the solution and the reliability of the presented results.

Chapter 2

Background

There is quite a bit of background knowledge that was utilized during this research. Most notably is the game itself; Checkers, as well as the Tsetlin Machine and commonly used methods of creating machine players. Checkers has many variants and variant rules, but American Checkers/English Draughts were chosen as this is the variant that seemingly is most played in the world of machine players

2.1 Checkers

Checkers refers not to a specific game, but a family of games [7]. The rules and board size differs among the families of Checkers and their sub-variants. There are three families of checkers: International Draughts/American Pool Checkers, Spanish Draughts and the smallest; English Draughts/American Checkers. In the latter family, there are three variants: English Draughts, Italian Draughts and Gothic Checkers. English Draughts, often called American Checkers, was the first Checkers variant to be arranged a world championship for; starting in 1840. English Draughts, is therefore one of the most widespread Checkers variant and do have some strong machine players showing promising results [7][8][9].

Checkers is a board game with two players, where the goal is to eliminate all the opponent's pieces in order to win [7]. If any player has no pieces left, but can no longer move any piece; the game results in a draw. The size of a Checkers board is typically either eight by eight, ten by ten or twelve by twelve and have somewhere between eight and thirty pieces on each side. The structure of the board and the rules also varies slightly. English Draughts/American Checkers has a board of eight by eight and twelve pieces on each side. The boards look similar to those of Chess, where the board consists of black and white squares which are only diagonally connected to squares with the same color as themselves. Only black squares are playable and the board is not symmetric. On each player's side of the board, their corner on the right-hand side is not playable as seen in Figure 2.1. Also, the black/colored player is the starting player; this is gold in this example.

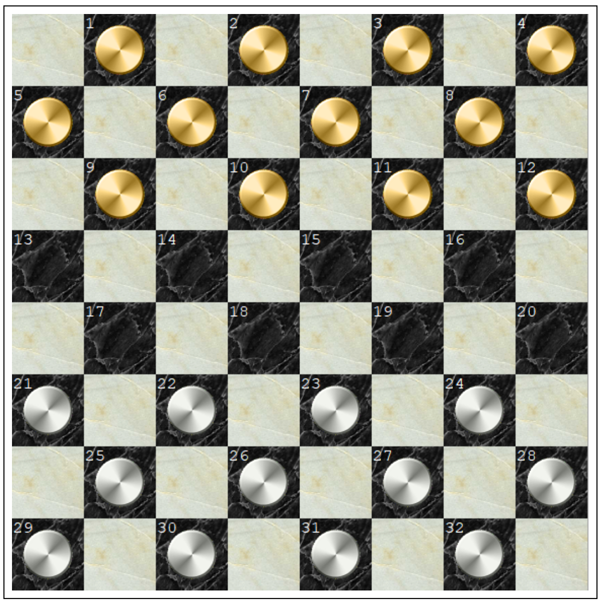


Figure 2.1: The starting board of English Draughts/American Checkers. All pieces shown are pawns. Gold represents the black player. White represents the white player. Picture reprinted CheckerBoard [1].

There are two kind of pieces; pawns and kings. Pawns may only move and capture in such a way that they move away from their starting position and are transformed into kings if they land on a square on the opposite edge of the board in relation to their starting side [7][10]. Figure 2.2 show the direction a black pawn may move, given it started from the top.

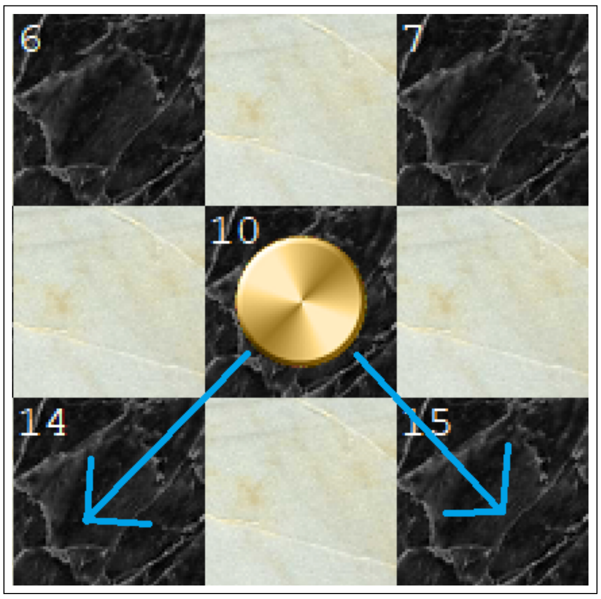


Figure 2.2: The possible non-capture moves a black pawn may perform given no pieces nor borders blocks its path. This pawn started from the top, direction of movement must be away from the starting side of the board for the given player. The directions illustrated also illustrates the directions this black pawn may perform capture moves. Figure adapted from CheckerBoard [1].

Kings may move and capture in any diagonal direction as shown in Figure 2.3. Kings may be differentiated from pawns by placing two pieces on top of each other, turn the piece upside down, or mark it by other means such as marking it with a crown or a star as seen in Figure 2.4.

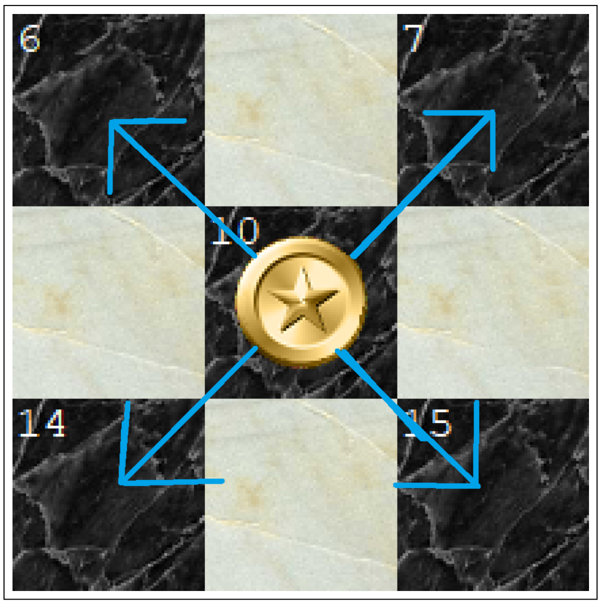


Figure 2.3: The possible non-capture moves a black king may perform given no pieces nor borders blocks its path. Captures may only happen in the directions illustrated. Figure adapted from CheckerBoard [1].

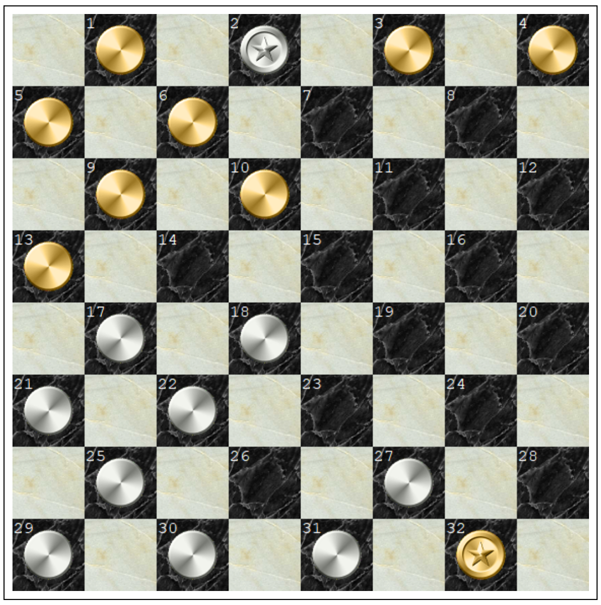


Figure 2.4: A board having two kings, which is different from pawns by having stars on them. Picture reprinted from CheckerBoard GUI [1].

Only one piece may be moved during a single turn, all moves are performed diagonally and a piece may only travel to a free diagonal square right next to it unless a capture move is performed [7][10]. A piece may not jump over allied pieces, but do jump over enemy pieces during a capture move. Capture moves can happen if a piece it normally would be able to move to contains an enemy piece and also has a free square behind it. The enemy piece that were jumped over gets ”captured” and is removed from the game; be it king or pawn. If a capture move can be taken, then a capture move must be taken. If the piece that performed the capture move lands on a square that grants it the opportunity to capture another piece, it must do so.

Some Checkers variants states that the player must capture as many pieces as possible during a turn, but this is not the case in American Checkers/English Draughts. The player simply must perform a capture move if possible, but it must continue its capture moves until it can capture no more. Figure 2.5 shows a piece in the position for a capture move. The white player must move either one of the two pieces shown having a red or blue line. Both possible moves leads a piece to another capture move which must be performed. If multiple captures from that situation were to arise, a choice would have to have been made.

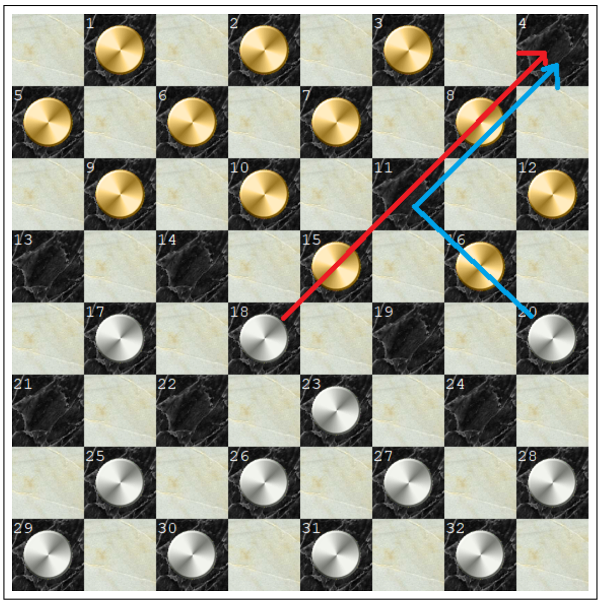


Figure 2.5: A board showing a capture situation for white player. Figure adapted from CheckerBoard [1].

Figure 2.6 shows how a king can capture both downwards and upwards. By sacrificing a piece, a player may force the opponent to put a piece of their own into a bad situation as shown in Figure 2.7 where the black player is capturing a king of the opponent.

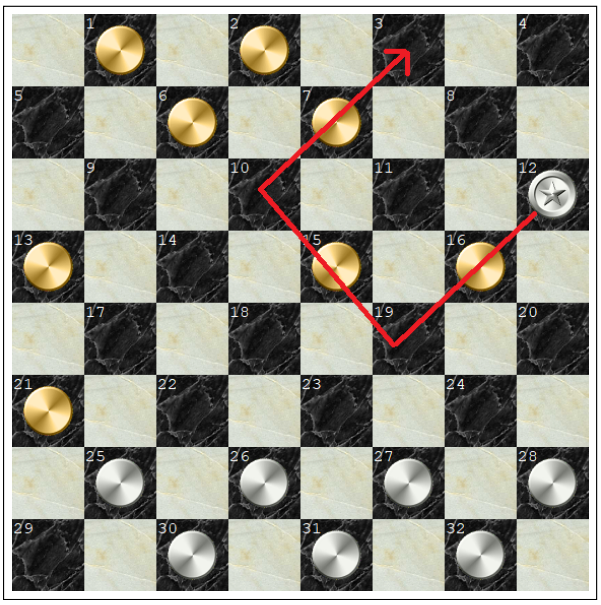


Figure 2.6: A king capturing both up and down, which a pawn can not. Figure adapted from CheckerBoard [1].

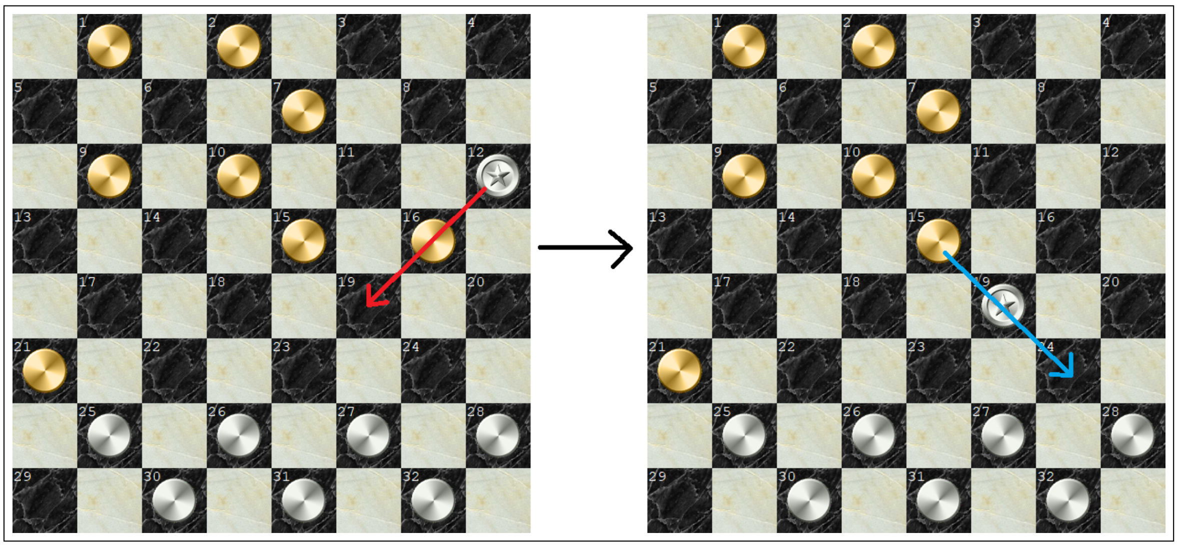


Figure 2.7: A trap to capture a king. Figure adapted from CheckerBoard [1].

A player lose if the player has no pieces left [10]. The game end in a draw if a player is unable to perform any legal move while still having pieces left or neither player can force a win. A depiction of a situation where the white player is unable to perform av move is shown in Figure 2.8. Additionally; players may agree upon a draw. For the rest of this report, English Draughts/American Checkers shall be referred to as Checkers.

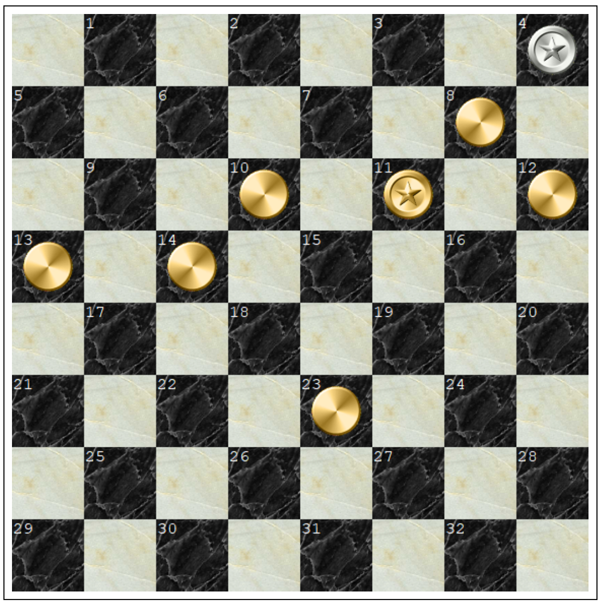


Figure 2.8: White player's last piece is blocked in which makes the game end in a draw. Picture reprinted from CheckerBoard [1].

Checkers has around $5x10^{20}$possible positions, which should be a tough challenge for any machine algorithm to play well [11]. The creators of the Chinook project has concluded Checkers to be a draw.

2.2 Tsetlin Machine

The Tsetlin Machine is the machine learning algorithm of interest for this research. There are two Python libraries of the Tsetlin Machine available: "pyTsetlinMachine" and "pyTsetlinMachineParallel" which features the ability to be run in parallel leading to a faster run speed [12]. The Tsetlin Machine is a machine learning algorithm based on the Tsetlin automaton [3]. The Tsetlin automaton outputs either one of two possible outputs. For simplicity, the possible outputs may be called *positive* and *negative*. It gets rewarded or penalized for the output which in turn affects the probability of what the Tsetlin automaton's next output will be. An illustration of this can be seen in Figure 2.9 where the farther left the automata's state is, the higher the probability for the next action to be *negative* and vice versa with the right side for *positive* output. This particular Tsetlin automaton has six states in total, but the amount of states may be less or more if wanted.

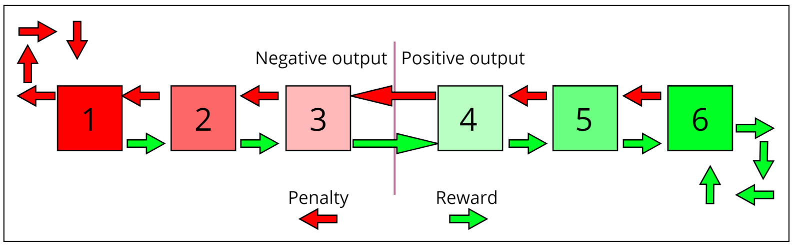


Figure 2.9: The Tsetlin automaton. Illustration adapted from the Powerpoint presentation "Tsetlin Machine Tutorial 4", 2019, by Ole-Christoffer Granmo [2].

These Tsetlin automata are then put into sets called clauses, and then organized into a Tsetlin Machine as seen in Figure 2.10 and Figure 2.11 [3]. The amount of Tsetlin automata for each clause is equal to the double of the amount of input the clause is allowed to use, as there are two Tsetlin automata tied to each binary input; one which has an opinion on whether its binary input should be positive (1) or not, and one which has an opinion on whether its binary input should be negative (0) or not. The amount of data that a clause may use is affected by the hyper-parameter s, which in turn affects the amount of Tsetlin automata for each clause. Additionally, the amount of clauses is equal for each possible class to predict. Clauses form conjunctive clauses which consists of two clauses; one which votes for whether it thinks the input data corresponds to the class the clause belongs to, and the other votes for whether it thinks the input data does not correspond to this class. For simplicity, these clauses can be called positive and negative clauses. Figure 2.10 feature a conjunctive clause. They predict the class of data where each datapoint consists of two bits. Together; they are called a conjunctive clause.

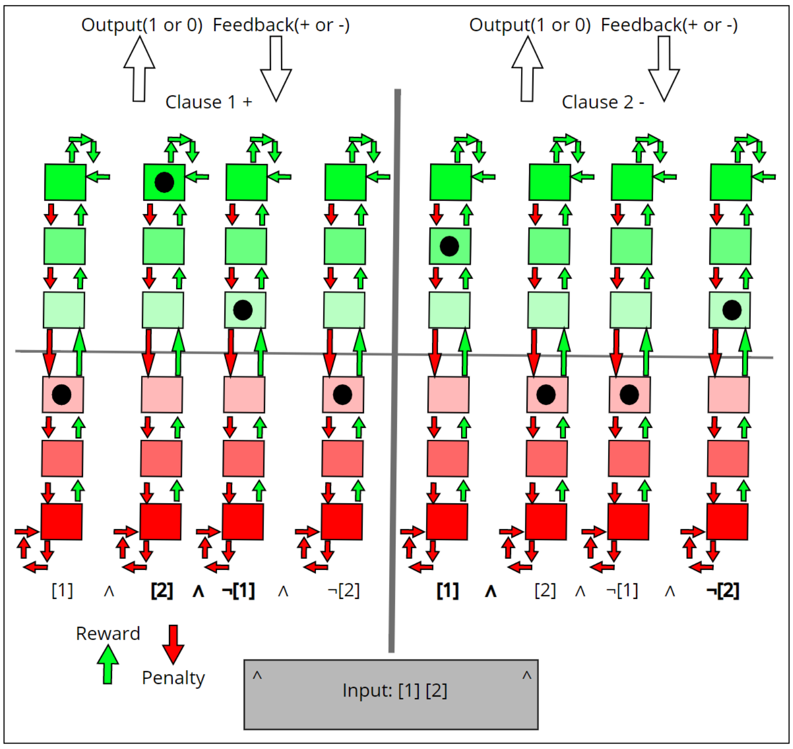


Figure 2.10: An illustration of both a positive and a negative clause, consisting of Tsetlin automata. Together; these are called a conjunctive clause. Bold text beneath each Tsetlin automaton of a Clause indicates the rule the Clause is most likely to evaluate the input for. Illustration adapted from Ole-Christoffer Granmo's Tsetlin Machine paper from 2018 [3].

The clauses in Figure 2.10 has learned a pattern in their input. Most likely: if the input is [0,1], the positive clause would output *1* and the negative clause would output *0*; together they tell the Tsetlin Machine that the class they are tied to is the class this input represents. This is just the most likely scenario as each Tsetlin automaton has a probability to output either *1* or *0* even if they are on a state far towards one end.

Figure 2.11 show these conjunctive clauses put into a Tsetlin Machine. If the input would be larger than two bits, each clause would contain more automatas in order to handle the data. This Tsetlin Machine feature eight conjunctive clauses, four for each of the two classes in this example (1 and 0). That makes eight clauses per output. The amount of clauses is not set, and may be expanded. The treshold function outputs *1* if a certain amount of conjunctive clauses also outputs *1*.

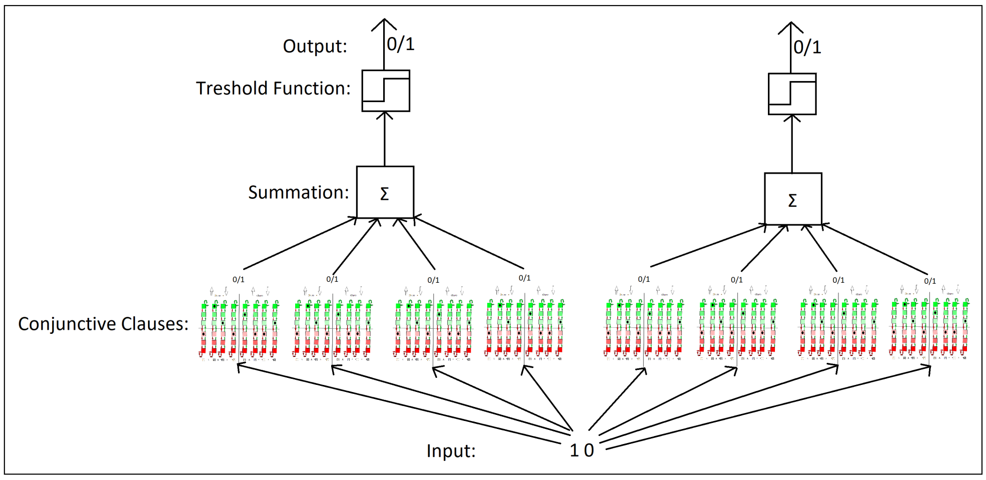


Figure 2.11: An illustration of a Tsetlin Machine, consisting of conjunctive clauses. Illustration adapted from Ole-Christoffer Granmo's Tsetlin Machine paper from 2018 [3].

This structure can further be transformed into the Multiclass Tsetlin Machine as seen in Figure 2.12, which is the variant of Tsetlin Machine used in this research as this Tsetlin Machine supports multiple classifications rather than either *1* or *0* [3]. The multiclass Tsetlin Machine shown below support three classes, but it could be expanded to support more as well. As with the illustration below, the amount of clauses may also be increased.

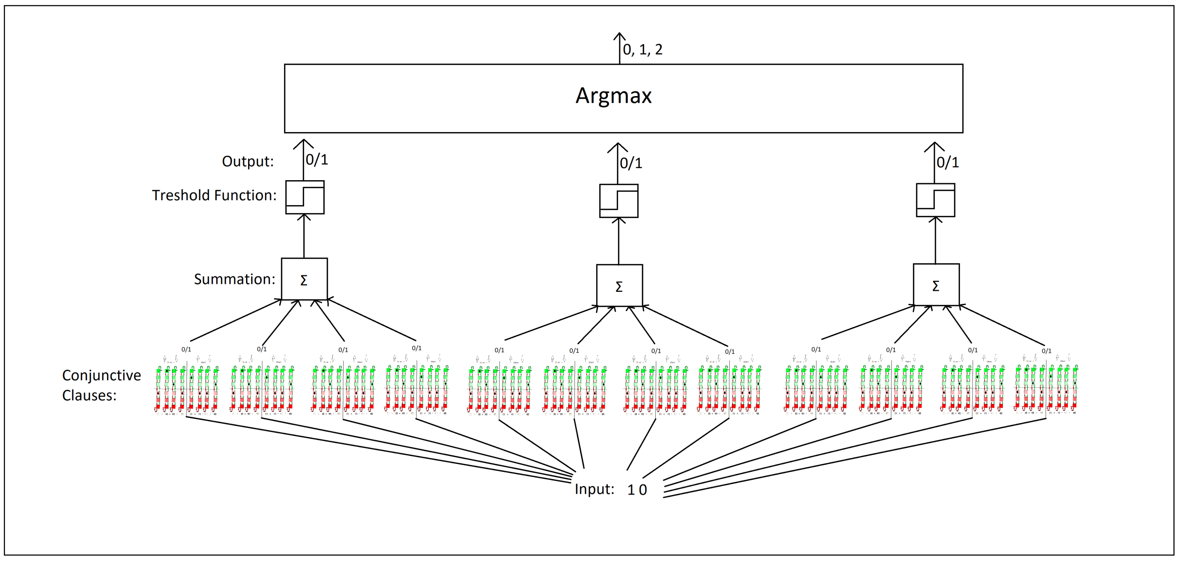


Figure 2.12: An illustration of the Multiclass Tsetlin Machine with three classes. Illustration adapted from Ole-Christoffer Granmo's Tsetlin Machine paper from 2018 [3].

The Tsetlin Machine takes three variables, called hyper-parameters: *clauses*, *treshold* and *s* [3]. The *clauses* is not the total amount of clauses as described in the beginning of this chapter, but the amount of clauses per class [13]. *Treshold* is shown in the illustration above, it determines whether the signal should be a *1* or a *0* depending of how high the number input to it is. Tweaking this input controls how many 1's must be input to it for it to output a *1*. *s* is used to affect the amount of literals to be included in each clause. A literal is a pair of Tsetlin automata; which react to the same bit. The lower the *s*, the lower the amount of Tsetlin automata in each clause, and vice versa. A high *s* allows the clauses to learn rules involving more data.

For this research, there were three outcomes: loss (0), win (1) and draw (2). Specifying the *clauses* hyper-parameter to be 100, would lead to there being 100 clauses per outcome [3]. 50 of the clauses per outcome find rules indicating that its outcome is not the correct outcome, while the 50 other clauses per outcome find rules indicating that this is the correct outcome. When the final prediction is to be performed by the Tsetlin Machine, there is a vote of a sort where the negative and positive clauses of each possible outcome votes against each other in order to come to an agreement of whether their respective clause is the most probable to be the case for this specific situation, which were Checkers boards in this research. The possible outcome with the most amount of positive agreement results in the final prediction of the board.

An option that is available for the Tsetlin Machine is **boost_true_positive_feedback**, later referred to as **positive boost** [13]. This may be an interesting option to investigate as Ole-Christoffer Granmo mention in his paper from 2018 regarding the Tsetlin Machine that in certain datasets; by boosting the rewarding of positive clauses when they produce true positive outcomes; that is, when the clauses predicts positively on the correct outcome, the accuracy of the predictions may improve [3].

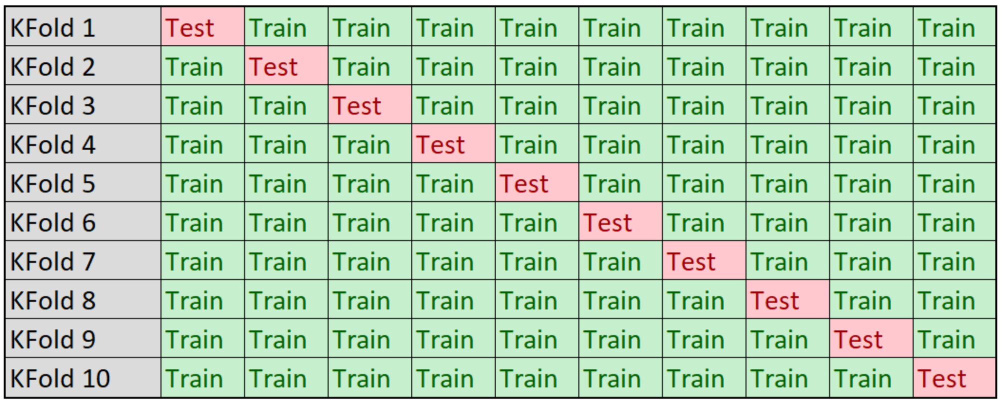
There is a variant of the Tsetlin Machine that may be called the **Weighted Tsetlin Machine** [13][14]. The Weighted Tsetlin Machine introduces weights for each clause, dictating how much value the vote of each clause has. With this weight attribute, it is possible to state that some clauses/rules are more valuable than others, and may improve the accuracy of the Tsetlin Machine. Tsetlin Machines utilizing weights may also achieve as high of an accuracy as non-weighted Tsetlin Machines, while having a less amount of *clauses*; which would make the Tsetlin Machine train faster.

Another variation of the Tsetlin Machine is the **Convolutional Tsetlin Machine** [15]. This Tsetlin Machine looks for patterns during training by investigating the data in chunks. This could lead to a higher prediction accuracy when trying to predict the result of a Checkers board. In addition to the hyper-parameters: clauses, treshold and s, the Convolutional Tsetlin Machine also incorporates the parameters: *shape_x*, *shape_y*, *shape_z*, *frame_x* and *frame_y*. The shape parameters define the shape of the board. During this research, most of the datasets that were used had a binary representation of Checkers boards where the board were represented by four series of bits, representing black pawns, black kings, white pawns and white kings respectively. The playable Checkers board is four by eight. The shape parameters were therefore *shape_x* = 4, *shape_y* = 8 and *shape_z* = 4. The parameters *frame_x* and *frame_y* define a square or rectangle which is the area to be investigated at a time. This area would move around the board while training instead of making the Tsetlin Machine train at complete boards at a time. The **Convolutional Tsetlin Machine** is most suited for situations that may be transformed into board-like structures, such as board games and pictures.

It is worth noting that these versions of the Tsetlin Machine can be combined as wanted. That is; The **Tsetlin Machine** may be **weighted** and/or have **positive boost** enabled. The same goes for the **Convolutional Tsetlin Machine**. Variations of these were investigated in this research and the Tsetlin Machine variations tested were based on the Multiclass Parallel Tsetlin Machine. Multiclass for the ability to predict more than two outcomes and parallel for speed.

2.3 K-Fold Cross Validation

The idea of k-fold cross validation is to validate the performance of machine learning algorithms more accurately than by training and testing the algorithm by using a single train/test set [16]. K-fold cross validation is to split the dataset into multiple parts, then train and test the machine learning algorithm with every combination of train/test data. Averaging the results for all of these combinations will give a better measurement of machine learning algorithms' accuracy by reducing the impact of lucky splits of train/test sets. An illustration of how the k-folds are split up and restructured for this project's implementation can be seen in Figure 2.13. In this research; every K-Fold contains the whole dataset, but each of them vary by the split of training and test data.



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

Figure 2.13: This project's implementation of K-Fold cross validation, 10 splits; 10 datasets based on the same data

2.4 Tree Search

Tree search typically refer to tree traversals, which is the act of traversing search trees [4][5]. Tree search is utilized in this research, but as a method of exploring possible moves in a Checkers match, not to explore existing trees. There are multiple ways of performing a tree search/tree traversal, such as in-order, pre-order, post-order, breadth first and more. The search used in this research is breadth-first. Each type of tree traversal typically needs an order of which node to start from within the rules of the selected tree traversal type. This may for example be Left-To-Right, where the search starts from the left and moves to the right, or Right-To-Left where the search starts from the right and moves to the left. In order to illustrate tree search in a way that is as relevant for this project as possible, Figure 2.14 illustrates a search tree created after exploring possible moves in Checkers. It was created using Left-To-Right Breadth-First traversal. The top node represents the current board for which the player in question must make a move for. The first row of nodes represents the moves the player may perform, while the second row of nodes represents moves the opponent may perform if the respective parent node's move was to be performed. Lastly, the nodes of the third row represents the moves the player in question may perform in response to the move of their respective parent's node. Scores are set on leaf nodes and are eventually averaged and set for their parent node resulting in the first row showing both possible moves to perform as well as the value of picking each particular move enabling a machine Checkers player to have a preference of what move to pick. If the search starts from the left-most node, the search is called left-to-right traversal.

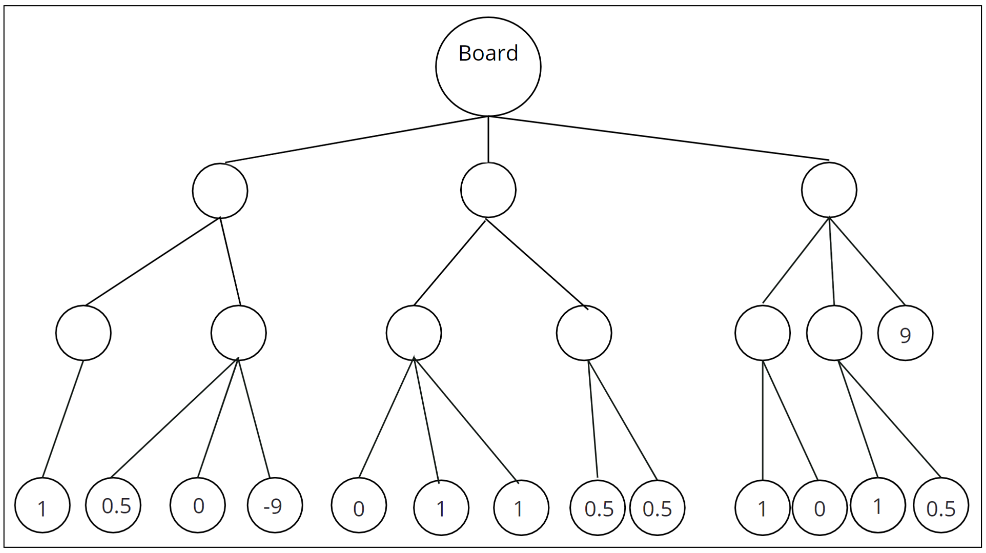


Figure 2.14: A visual representation of a tree search performed in the game of Checkers, reaching depth three.

Pre-order is a tree traversal method that aims to search deep down the layers until reaching a leaf, then searching the siblings and searching deep down for each of them until reaching another leaf node [4][5]. When all sibling's leaf grandchildren are found, the search moves one layer up and continues the search for the next leaf nodes. Left-To-Right Pre-order tree traversal is illustrated in Figure 2.15 where each number represents the order of which the node is found.

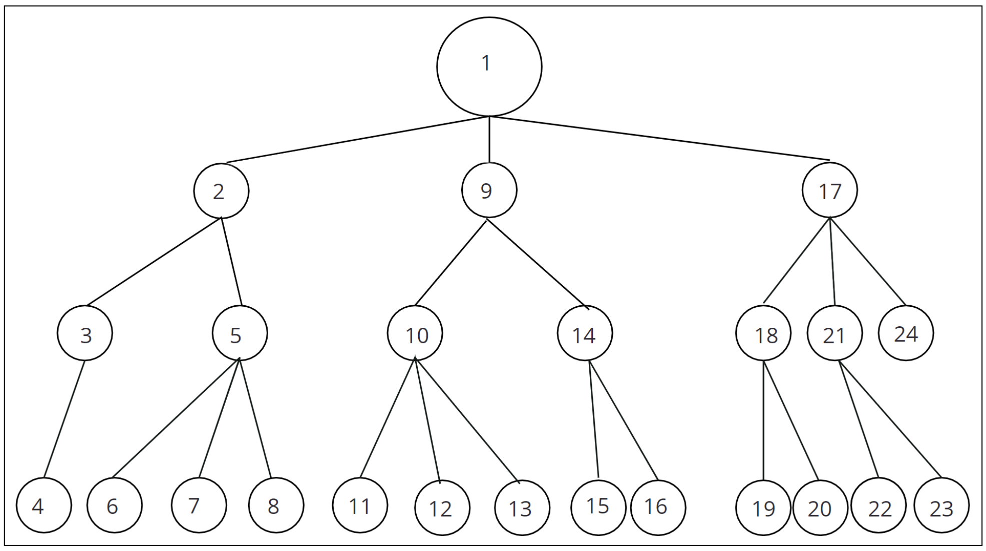


Figure 2.15: A visual representation of a Left-To-Right Pre-order tree traversal, adaptation of slide number three of [4].

Breadth-first is a tree traversal method investigating each of a node's children before moving to each of their children [5]. In this way, a full depth-level is found before moving deeper. Left-To-Right Breadth-first tree traversal is illustrated in Figure 2.16

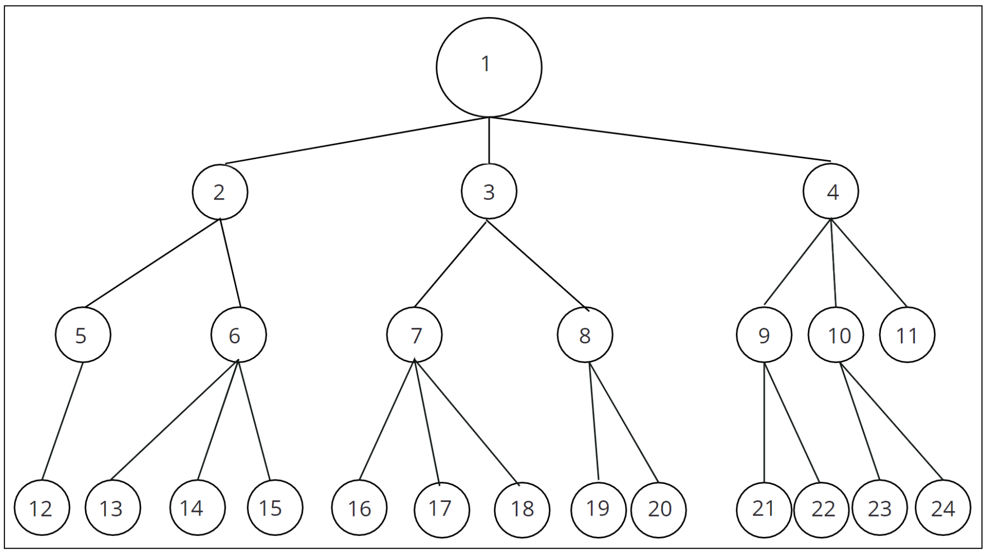


Figure 2.16: A visual representation of a Left-To-Right Breadth-first tree traversal [5].

2.5 Precision, Recall and FScore

The accuracy of a machine learning algorithm's predictions is typically a score shown in percent, representing how well the algorithm is able to classify data. There exists a more in-depth method of measuring how well a machine learning algorithm is able to predict a specific outcome/class, which is by the calculation of the scores *precision*, *recall* and *FScore* [17][18]. This research use the three possible outcomes of a Checkers match: loss, win and draw and these are therefore the three classes the machine learning algorithm should be able to predict well for this research. These three scores may be calculated for each of the possible classes. "win" is the class in focus for the examples when explaining what the scores are, but may be interchanged with any other class if need be.

The three scores are as follows:

- **Precision:** The percentage of predicted wins that are actual wins.
- **Recall:** The percentage of wins in the tested data that was predicted to be wins.
- **FScore:** A way of summarizing of both *precision* and *recall* into a single score by calculating their harmonic mean.

These scores together may be called *stats* in the rest of this paper for simplicity's sake. There exists multiple ways of calculating the FScore, but for this research; the following formula was used [18]:

$$FScore = 2 * (\frac{precision * recall}{precision + recall})$$

Example
The test data contains **500** data points. **239** of these are *wins*, the rest; **261**, are *losses.* The machine learning algorithm predicts **340** *wins*, and **231** of these are *actual wins.* Given this data and these predictions, the stats are calculated as follows [18]:

- $Precision = 100 * (\frac{correctly-predicted-wins}{predicted-wins}) = 100 * (\frac{231}{340}) = 67.94\%$
- $Recall = 100 * (\frac{correctly-predicted-wins}{total-wins-in-test-data}) = 100 * (\frac{231}{239}) = 96.65\%$
- $FScore = 2 * (\frac{precision*recall}{precision+recall}) = 2 * (\frac{67.94*96.65}{67.94+96.65}) = 79.79\%$

Precision tells how precise the predictions are [17]. If the precision is high, it is highly likely that a predicted win is actually a win. With a precision score of **67.94%**, it would not be wise to entirely rely on the prediction; but it would be unwise to guess against a win prediction as the precision is of a level way above chance as there are three classes in the domain of Checkers. Recall tells how many of the classes in question was predicted correctly. If the recall is high such as **96.65%**, it is highly likely that a predicted loss or draw is not a win. As FScore incorporate both precision and recall, it may be used as a way of collectively evaluating the prediction performance [18].

Chapter 3

State-of-the-art

This chapter gives an overview of previous work in the world of machine learning in English draughts/American Checkers.

3.1 Chinook

The Chinook project started in 1989 with the goal of building a program capable of challenging the world checkers champion [11][19]. The Chinook project earned the right to play in the World Championship a single year after conception. Marion Tinsley was the 8-time Checkers World Champion, he was highly respected as a Checkers player by the creators of Chinook; Jonathan Schaeffer et al., and had a single recorded loss in Checkers between 1950 and 1991 [11][20][21]. Naturally, it was important for the Chinook project to beat Marion Tinsley. In 1992 Marion Tinsley narrowly won against Chinook in the title match. A rematch were held in 1994, but Marion Tinsley unfortunately had to withdraw due to illness. Eight months later, Marion Tinsley passed away. In 1996; having completed its eight-piece dataset, Chinook proved itself to be much stronger than all human players, but whether it was able to beat Tinsley had to go unanswered for a while [9]. The project was suspended in 1997, but were continued from 2001. The Chinook project expanded the database to include all possible boards containing up to ten pieces present; 39 trillion positions. In 2007, it was announced that Checkers was weakly solved by Chinook [11].

For reference, the three degrees of solving a game according to the creators of Chinook is ultraweakly solved, weakly solved and strongly solved [11]. The definitions are extracted from the paper by the name "Checkers is solved" by Jonathan Schaeffer et al. but the definitions are widely acknowledged [22][23].

Ultraweakly solved: "For the lowest level, ultraweakly solved, the perfect-play result, but not a strategy for achieving that value, is known.".
Weakly solved: "For weakly solved games, both the result and a strategy for achieving it from the start of the game are known."
Strongly solved: "Strongly solved games have the result computed for all possible positions that can arise in the game."

It is not known entirely how Chinook works, but its proof algorithm works as follows [11]. Chinook utilizes retro-grade analysis together with its database containing information of endgame boards and their win/loss/draw value. The algorithm starts not from the current board, but from the end of the game, finding its way to the current board. All one-piece positions are enumerated which determine their value. Then all two-piece positions are enumerated and analyzed. Analyzing each position eventually directs the algorithm to a one-piece position with a known value or a repeated position resulting in draw. This goes on until it has been performed for all boards with up to ten pieces.

According to a document by the World Checkers & Draughts Federation from 2014, Chinook is the World Man v Machine Champion [24][21].

3.1.1 Chinook dataset

The Chinook project's dataset were available and has "perfect information for all checker positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions" according to a website by the University of Alberta [25]. The dataset contains information about the board and whether it results in a loss, win or at least a draw (draw or win) [26]. The generation of this dataset relied on Chinook itself and a secondary program using the Df-pn algorithm as a way to prove the result after having algorithms checking the legality of the board [11]. How the first boards were proven, which Chinook relied on to make an assessment of new boards, is unknown. This is a very large dataset, as it contains every single possible board with 8 or less pieces as stated. Considering that the well known Chinook Checkers player use this dataset, this dataset were therefore the first dataset that were considered to be used. The data is stored in a database for limiting the storage space it takes as well as for making it faster to access for the software. The database files are split by characteristics of boards; the base data, which is the "end-game", contain data of boards with 2 through 6 pieces. The other files in the database are split by more specific rules; not only are they split by the amount of pieces on the board, but also by how many pieces there are on each side and the number of kings present on the board. Considering Chinook rely on perfect information, being able to look up a specific type of board quickly is key to its success. Unfortunately, the provided driver were not able to be run in order to retrieve data from the database which made it difficult to retrieve the boards from the Chinook dataset. The cost in terms of time to make the driver able to be run was estimated to be too high.

3.2 Kingsrow

Kingsrow is a Checkers engine written by Ed Gilbert [27][8]. Kingsrow formerly used a manually built and tuned evaluation function, which gave a board a score based on a number of factors. Written by Ed Gilbert himself, found on Bob Newell's website bobnewell.net dating back to the 17th of november 2018: "This function computes a numeric score for a game position based on a number of material and positional features. It looks at the number of men and kings of each color, and position attributes including back rank formation, center control, tempo, left-right balance, runaways (men that have an open path to crowning), locks, bridges, tailhooks, king mobility, dog-holes, and several others." [8].

The newest version of Kingsrow use machine learning techniques, logistic regression, and has proven to be a better Checkers player than earlier renditions of Kingsrow, which it has learned through playing against itself [8][28]. The machine learning evaluation view the board through rectangles containing either 8 or 12 pieces. In the rectangles containing 8 pieces, any piece may be present, but in the rectangles containing 12 pieces; no kings are present. Every combination of pieces in these rectangles are given a score by the machine learning algorithm, and the sum of these represents the final score given to the board.

Bob Newell mentions on his site that Ed Gilbert's new version of Kingsrow is "super-strong", although; he does not define this further [8]. In 2002, the Checkers World Championship in Las Vegas was held where Kingsrow was competing against the machine Checkers players Nemesis and Cake [29]. Nemesis won the tournament, Kingsrow took second place and Cake took third and last place. The winner of this tournament was said to earn the right to play against Chinook in the Man-Machine World Championship, but it is tough to find any source indicating Nemesis playing against Chinook in any circumstance; even Nemesis' former home-page is not active anymore: *https://www.nemesis.info/* [30]. A document by the World Checkers & Draughts Federation from 2014 contain a list of all title holders [24][21]. In this writing, Chinook is stated to still hold the title of World Man v Machine Champion, indicating that if Nemesis got to challenge Chinook in a Man-Machine World Championship between it's victory in Las Vegas and 2014, Chinook defended the title.

3.2.1 Kingsrow dataset

Kingsrow use a dataset with boards of 2 through 10 pieces which use 102 GB of storage space with the newest compressed version of the dataset [8][27]. It is unknown how many boards is included in this dataset, but given that an article from 2019 by Bob Newell state that Ed Gilbert created a new version of Kingsrow that use machine learning and create new data by playing; implies that the former dataset was not complete [8]. There are two auxiliary endgame databases that may also be used, but only one at a time in conjunction to the 2 through 10 pieces database [31]. These are the DTW (depth to win) database; allowing Kingsrow to play the absolute shortest path to a win, eventually longest path to a loss, and MTC (depth to conversion) which allows Kingsrow to take the shortest path to the next non-reversible move. This dataset was unfortunately built similarly to the Chinook dataset which was presented in chapter 3.1.1 and the provided driver was also unfortunately not possible to be run. The time predicted to be needed to make it run was too high a cost for this research. However, the Kingsrow player itself was possible to be run through the CheckerBoard GUI by Martin Fierz; making it possible to play against Kingsrow [31].

3.3 Cake

Inspired by the matches between Marion Tinsley and Chinook, Martin Fierz started programming what would later be called the Checkers player Cake in the end of 1996 [28]. Cake was born in 1999 when Martin Fierz decided to split the interface and the Checkers engine (player). The Checkers player was given the name Cake and the interface was given the name CheckerBoard.

Most, if not all, traditional Checkers players used handmade evaluation functions not long ago [8][28]. The evaluation function give a board a score by numerous factors and weights; such as the amount of pieces and their positions, which are used to determine how much the Checkers player should want to make each evaluated board to happen. Creating and fine-tuning evaluation functions was more akin to art than science, according to Martin Fierz in his article about Cake 1.87 from late 2019 [28]. This was of course very time consuming, and optimal tuning was very difficult to achieve. Recently, quite a few Checkers players have gotten machine learning implemented in order to improve the evaluation function, so did Cake. Cake had its evaluation function improved by logistic regression.

While Kingsrow utilized machine learning for learning its own evaluation function and set its weights, Martin Fierz wanted to implement machine learning into Cake without replicating the method used in Kingsrow [28]. Cake kept its hand-crafted evaluation function, but the weights of each of the 379 hand-made evaluation factors was optimized with logistic regression.

Martin Fierz has compared his Cake with Ed Gilbert's Kingsrow numerous times [28]. They did both also compete in the Las Vegas Computer Checkers World Championship in 2002 where Kingsrow took second place; behind Nemesis, and Cake took third and last place [29]. On the 18. of September 2019, one day before Ed Gilbert published a new version of Kingsrow, Martin Fierz published an article about Cake 1.87 which featured a comparison between it and Kingsrow 1.18f; both of which used machine learning [28][31]. Although Cake 1.87 did perform significantly better than the older version of Cake which did not use machine learning, Cake 1.87 was easily beaten by Kingsrow 1.18f.

3.4 Nemesis

Not much is known about Nemesis as the website most sites refer to for more information seems to be a half-finished website to be used for generic advertisement, *https://www.nemesis.info/* [29][32][30]. What is known however; is that Nemesis has been the reigning World Computer Champion since 2002 [21][24][29]. It is also known that it was created by Murray Cash [33]. It is assumed that Murray Cash has stopped developing Nemesis, although; Martin Fierz did mention in a forum by "The American Checker Federation" stating Murray Cash was working on Nemesis 2 in march 2007 [34].

3.5 CheckerBoard

CheckerBoard is a free Checkers interface program for the operating system Windows, created by Martin Fierz [1][27][35]. It allows for both manual play by interaction by the use of the mouse cursor, as well as both Checkers engine versus Checkers engine and human versus Checkers engine. It can be used for playing against both Cake and Kingsrow, with various databases and settings. In this research, it served as a means of playing against Kingsrow.

3.6 Contributions

This research propose the following:

- The multiclass weighted Tsetlin Machine with positive boost trained on a dataset made by assembling final boards found in OCA 2.0 without the removal of duplicate data is the best Tsetlin Machine configuration with the best available and accessible dataset for this domain.
- Comparisons between methods of extracting data from OCA 2.0 and comparisons of various Tsetlin Machine configurations are presented.
- The Tsetlin Machine put in a tree search algorithm of depth tree makes for a Checkers player able to play Checkers more successfully than making moves at random.
- The use of a function of adaptive epochs during hyper-parameter testing can reduce both processing time and time spent by the researcher when finding the optimal hyper-parameters.
- The Tsetlin Machine is generally more proficient in predicting outcomes of boards in the domain of Checkers than various other machine learning algorithms, including Logistic Regression and Decision Tree Classifier.
- The proposed Checkers player may be able to compete with highly tuned and skilled Checkers players in the likes of Cake and Kingsrow by improving both the quality and the quantity of data; as well as further tuning.

Part II

Contributions

Chapter 4

Proposed Solutions

This chapter explains in-depth what the proposed solutions are. This includes an explanation of the chosen data source for this research; OCA 2.0, and why this data source was selected. The most interesting and promising of the created dataset compositions from this data source are presented, as well as the process of creating these dataset compositions. The Checkers board predictor, the chosen Tsetlin Machine configuration with fitting hyper-parameters, is explained as well as showing a short overview of the performance of the trained Tsetlin Machines used in the Checkers player for the chosen dataset composition. Also, the Checkers player itself, and its use of tree search; will be explained.

A simplified overview showing the flow from the data source, all the way to the Checkers player is presented below; all of which is explained in-depth in this Chapter. Figure 4.1 show this overview, with the Checkers player itself excluded. The dataset composition is created, then trained on by the Tsetlin Machine for which to be used by the Checkers player in the Checkers script. The Tsetlin Machine must be trained on a specific K-Fold, even though 10 K-Folds are created. The Checkers player takes the place of *Player 1*, *Player 2* or both. The same flow of training the Tsetlin Machine would be used for evaluating its accuracy as well, however; without saving the Tsetlin Machine and training on a full 10 K-Fold instead of just one.

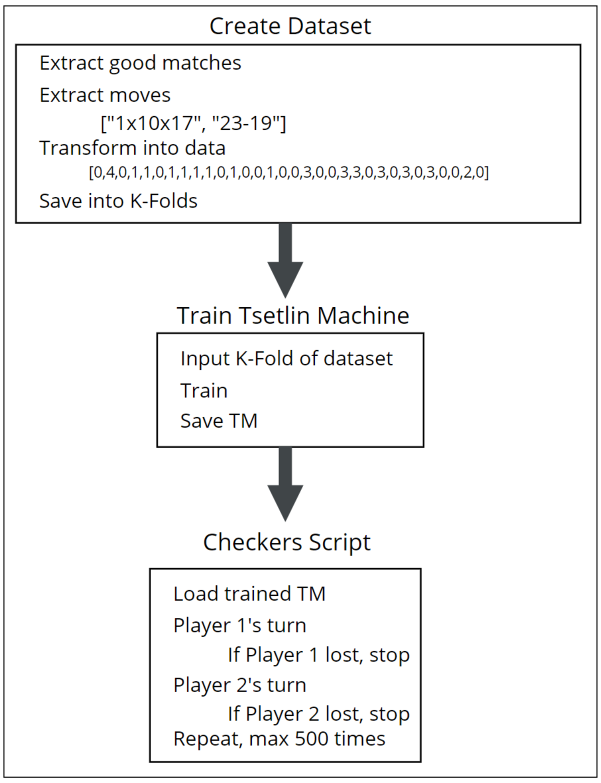


Figure 4.1: Simplified overview of the solution; flow from data all the way to the Checkers script. The Checkers player may take the place of *Player 1*, *Player 2* or both. *TM* is short for Tsetlin Machine.

The flow of the functionality for accepting moves from an external Checkers player and the flow of the Tsetlin Checkers player itself is shown in Table 4.2. Although; not much is going on in the flow for accepting moves from an external Checkers player, it shows that no rules are enforced. This is due to most other Checkers players enforce the rules of checkers and the implementation was meant to be used for research purposes only; which means a researcher would oversee non-self-play games. It would of course be beneficial to enforce these rules, but in the case of a misplaced piece; the researcher could simply input the correct board and continue the game.

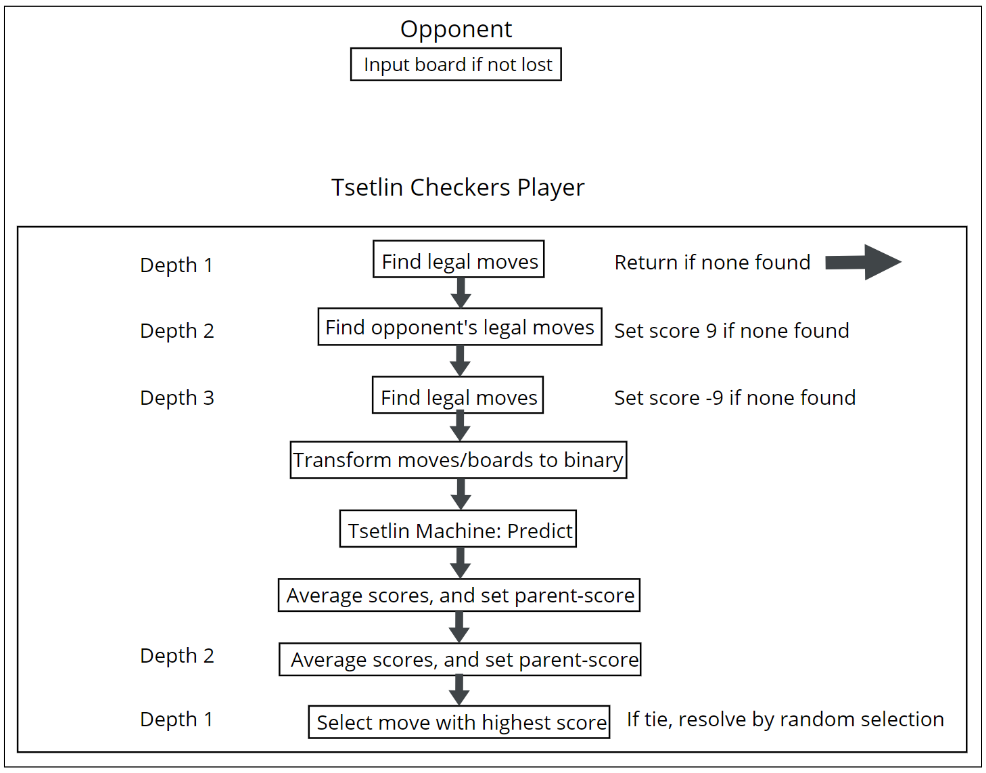

Opponent
Input board if not lost
Tsetlin Checkers Player
Depth 1
Find legal moves
Return if none found
Depth 2
Find opponent's legal moves
Set score 9 if none found
Depth 3
Find legal moves
Set score -9 if none found
Transform moves/boards to binary
Tsetlin Machine: Predict
Average scores, and set parent-score
Depth 2
Average scores, and set parent-score
Depth 1
Select move with highest score
If tie, resolve by random selection

Figure 4.2: Simplified flow of the Checkers player as well as the functionality for accepting moves from an external Checkers player. If the Checkers player find no moves, the Checkers player has lost.

4.1 Data source

Both the dataset of Chinook and of Kingsrow was considered as valuable sources of data, but due to non-functional access code and incomprehensible file formats; it was not possible to extract information from these datasets as mentioned in Chapter 3.1 and 3.2. OCA 2.0 was chosen as the data source for this research. The reasoning for choosing this particular source of data was due to this dataset being the largest data source that was both available and accessible.

4.1.1 Open Checkers Archive 2.0

Open Checkers Archive 2.0, shortened "OCA 2.0" is the dataset that was chosen for this research. OCA 2.0 is not a very large dataset, but the data itself is simple to extract. OCA 2.0 contains logs from real matches in American Checkers/English Draughts. Quite a few people were involved in the creation of this dataset, but the most notable people are Hans l'hoest and Jim Loy [36]. Each datapoint has information of whether the match ended in a loss, win or draw, the name of the two players and what colors they played, the name of the event where the match was held, the date and location of this event and a representation of the board in the form of a list of moves. As it was likely manually compiled, some comments are added for unique situations; for example if one player chose to forfeit or an illegal move was made without this being noticed until after the match. The base dataset contains 22622 datapoints, 22614 of these follows the typical move-pattern of the data file and has no additional comments which might compromise the correctness of the match, 22578 of these are made of legal moves and 21702 of these are unique datapoints; that is, the combination of both the final board and the result is unique for each of these 21702 datapoints.

4.2 Creation of Datasets

The data from OCA 2.0 needed to be processed in order to be read by the Tsetlin machine, as it required data in binary, while the result did not have this limitation. It was assumed that the Checkers boards would be more relevant to train and test on than the moves made. The data that were used were the moves made and the result of the match. These were checked for formatting, checked for rule-breaking, assembled into boards and converted to binary. The OCA 2.0 dataset did contain a noticeable amount of matches where additional notes regarding irregularities such as overlooked rules, forfeits and matches that simply did not follow the rules of English Draughts/American Checkers. Various different compositions of the data in OCA 2.0 were experimented with, some including boards where not all moves were added as an effort to manufacture more data as well as datasets containing metadata of the boards. This section will go in-depth of the handling of data during the creation of the numerous datasets created for this research with the goal of finding the optimal usage of the OCA 2.0 dataset for the best accuracy when fed to the Tsetlin Machine.

Various methods of creating dataset compositions based on the data in OCA 2.0 was attempted in order to train the Tsetlin Machine to be highly accurate. The main idea for doing this was to expand the existing data into more data, but alternative use of the data was also attempted. How these dataset compositions were created can be seen and read about below.

4.2.1 Base structure

The base structure for creating boards out of the matches found in OCA 2.0, aimed to extract the moves and match result from the source, transform these into legal boards in binary form and distribute them across K-Folds along with each board's respective result given by what match from the data source they were generated from. In an attempt to clean the data as much as possible, all duplicate data were removed. That is, for each combination of board and result; only one were kept. The tweaking of the duplication checks did lead to multiple dataset compositions down the line.

The base transformation process looked like this:

Algorithm 1 Base Transformation for Dataset Composition

```
1: Pure data (22622 data points)
2: Moves and results extracted (22614 data points remaining)
3:     Abnormal matches omitted, typically where comments about overlooked rules, forfeits
   or wrong results were present or the moves listed did not follow the normal pattern for the
   dataset
4: Transformed into legal boards (22578 data points remaining)
5: Transformed into binary
6: Duplicate data are removed (21702 data points remaining)
7: Data is distributed to K-Folds
8:     10 K-Folds provides 10% test data. Data with loss result (21,9%), win result (16,9%)
   and draw result (61,5%) are distributed evenly across all 10 slices
```

4.2.2 Base structure for extraction of multiple boards

In order to generate more data from the data found in OCA 2.0, it was attempted to create multiple boards from the same match by omitting moves from the list of moves before creating the board. A particular dataset by the name of EndSecondThird used each match to create three boards; one with all the moves performed, one with one whole turn removed and another with two whole turns removed. Whole turns was removed, which means two moves due to each player performing a single move each per turn, in order to keep the turn order intact. This made it so that whenever the Tsetlin Machine was trained or tested, the data was based on boards where the black side moves next. The way the data were distributed among the K-Folds, and therefore train and test data, had to be altered in order to avoid having boards generated from the same match exist in both the train data and test data for the same K-Fold run. The reason for this is that having data from the same match be distributed across multiple K-Folds would lead to the Tsetlin Machine being trained and tested on some data originating from the same match for at least one K-Fold run, which would make the test data impure and the resulting accuracy inaccurate. In other words; it would be like giving students parts of the answers for some questions during an exam.

The base transformation process for multiple boards looked like this (numbers given for the EndSecondThird dataset composition):

Algorithm 2 Base Transformation for Multiple Boards Dataset Compositions

```
1: Pure data (22622 data points)
2: Moves and results extracted (22614 data points remaining)
3:     Abnormal matches omitted, typically where comments about overlooked rules, forfeits
   or wrong results were present or the moves listed did not follow the normal pattern for the
   dataset
4: Duplicate data are removed (22288 data points remaining)
5: Data is distributed to K-Folds
6:     10 K-Folds provides 10% test data. Data with loss result (21,8%), win result (16,8%)
   and draw result (61,4%) are distributed evenly across all 10 slices
7: New lists of moves are added, but some of the moves are removed
8: Transformed into legal board
9: Transformed into binary
10: Duplicate data are removed
```

4.2.3 Base structure without duplication checks

After not getting sufficient results, changes were made to the structure of how new datasets were made. The removal of duplicate data remove information the Tsetlin Machine may learn from. Datasets without duplication checks should in theory increase the accuracy of all usages of the data in OCA 2.0, especially when including non-final boards. Non-final boards has an increased risk of being identical with other boards as removing enough moves would eventually end in a board equal to the starting position of the board; which many boards share. That many moves were never omitted, but some of the effect were still there. These identical boards could lead to either loss, win or draw. Keeping only a single copy of each of them would train the Tsetlin Machine to believe that there would be an equal chance of either of the three results when in fact two hundred boards resulting in draw could have been removed and just a few boards resulting in win and loss; which could have otherwise made the Tsetlin Machine understand that these boards are likely to lead to a draw. The no duplication check versions of dataset compositions may be shortened to NoDupeCheck.

The base transformation process for multiple boards looked like this without the removal of duplicates (numbers given for EndSecondThird dataset composition):

Algorithm 3 Base Transformation for NoDupeCheck, Multiple Boards Dataset Compositions

```
1: Pure data (22622 data points)
2: Moves and results extracted (22614 data points remaining)
3:     Abnormal matches omitted, typically where comments about overlooked rules, forfeits
       or wrong results were present or the moves listed did not follow the normal pattern for the
       dataset
4: Data is distributed to K-Folds
5:     10 K-Folds provides 10% test data. Data with loss result (21,2%), win result (16%) and
       draw result (62,8%) are distributed evenly across all 10 slices
6: New lists of moves are added, but some of the moves are removed
7: Transformed into legal boards
8: Transformed into binary
```

4.2.4 Transformation into boards

As a measure to make sure the Tsetlin Machine used the best possible data to learn from, the moves from OCA 2.0 were not just transformed into boards; but also screened for rule breaking moves. If the script noticed an illegal move during the assembly of a board, the board were trashed and it started assembling the next board. Unfortunately, not every single rule were screened for, but an adequate amount of rules were implemented.

Implemented rules:

1. Piece check
   (a) Ensure that the piece to move actually exist (Not an empty space).
2. Consistent turn order
   (a) Black start, and no player may perform two actions in a row.
3. Direction check
   (a) Ensure that pieces move in the allowed direction, kings are omnidirectional.
4. Capture check
   (a) Ensure that the piece(s) to capture actually is(are) owned by the opponent and that empty spaces are not attempted captured. This rule also ensure that the landing spot(s) is(are) unoccupied.
5. Diagonal check
   (a) Ensure that the piece(s) that is(are) attempted captured is(are) diagonal to the piece attempting the capture(s).

Unimplemented rules:

1. Force capture check
   (a) A rule of American Checkers states the following: If a capture move may be performed, a capture move must be performed. If there are multiple possible captures, one of them must be chosen. The data was not screened for this rule.

4.2.5 Transformation into binary

The Tsetlin Machine did only accept binary input, excluding the result which could be just about anything. For simplicity, this research transformed the three results: loss, win and draw into the numbers *0*, *1* and *2* respectively. The data itself was transformed into *0*'s and *1*'s. This transformation was not as simple. A space on a Checkers board may be empty or contain one of the following pieces: *black pawn*, *black king*, *white pawn* or *white king*. The board itself contain 32 squares where a piece may be placed. Boards with placed pieces were represented in binary form by having 32 bits for each type of piece where *1* represent the fact that specific type of piece is there and *0* represent its absence. This resulted in 32*4 bits which may be visualized as four boards, where if all four boards has a 0 on bit 2; that square is empty. A visualization of this transformation can be seen below; Figure 4.3 shows the board before it was transformed into binary, while Figure 4.4, 4.5, 4.6 and 4.7 shows the binary boards for black pawns, black kings, white pawns and white kings respectively. Some dataset compositions utilize metadata. Metadata were made into integers which were easily transformed into binary form.

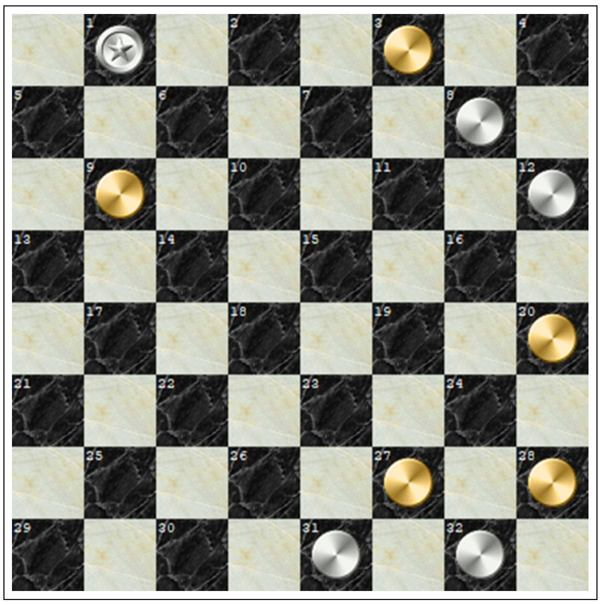


Figure 4.3: A visual representation of a board. Black pieces are golden and white pieces are silver in this picture. Picture adapted from CheckerBoard [1].

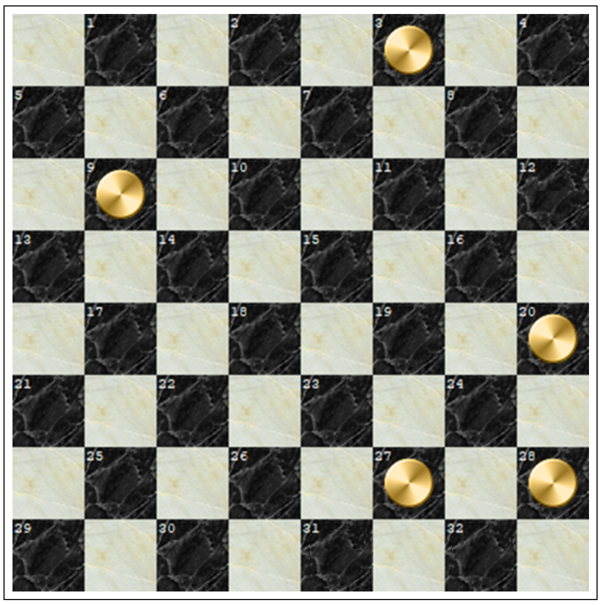


Figure 4.4: A visual representation of the first 32 bits of a board in binary form; displaying black pawns. Figure adapted from CheckerBoard [1].

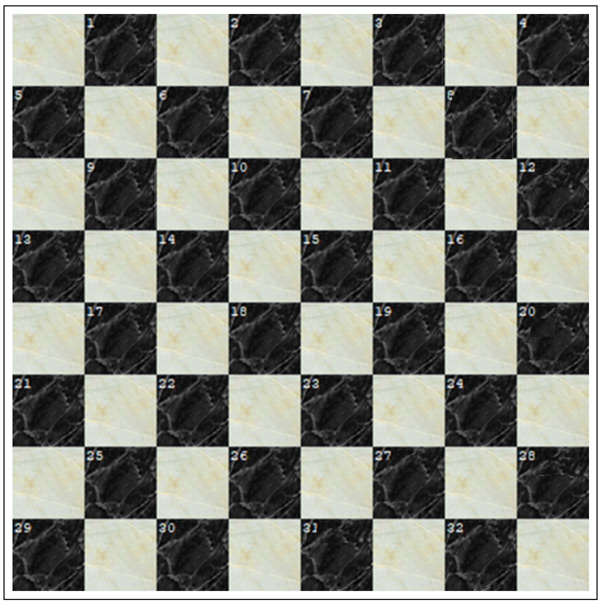


Figure 4.5: A visual representation of the second 32 bits of a board in binary form; displaying black kings. Figure adapted from CheckerBoard [1].

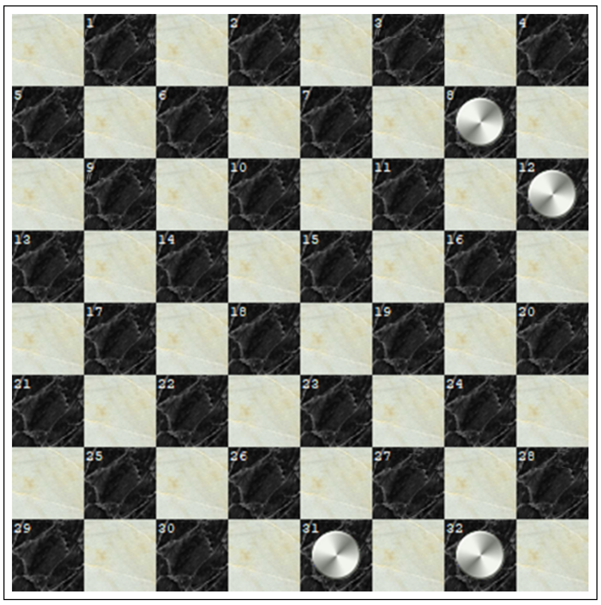


Figure 4.6: A visual representation of the third 32 bits of a board in binary form; displaying white pawns. Figure adapted from CheckerBoard [1].

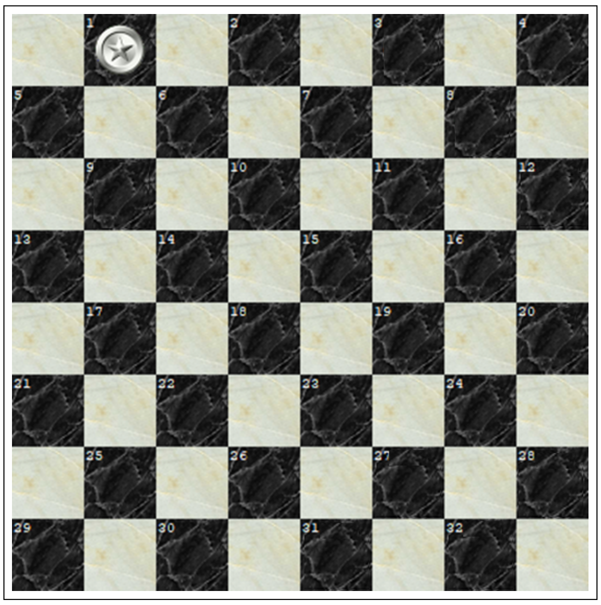


Figure 4.7: A visual representation of the fourth 32 bits of a board in binary form; displaying white kings. Figure adapted from CheckerBoard [1].

4.3 Dataset Compositions

Various dataset compositions were made in an attempt to get the best possible results with the data available in OCA 2.0. At first, quite a few unique dataset compositions were made; but were then remade when it was discovered that datasets with no duplication checks were showing an improvement in the accuracy score when compared to similar datasets having duplication checks. The reasoning for the duplication checks to exist in the first place was the assumption of duplicates having a negative impact in the quality of the data as well as compromising the integrity of the test data. After discussion with this research's supervisor, it was noted that removing duplicates from datasets were not the norm and keeping duplicate data does not necessarily compromise the test data.

It is worth noting that the Convolutional Tsetlin Machine is incompatible with any dataset containing anything else than a game board and were therefore not tested for these dataset compositions. The reason for this is that the Convolutional Tsetlin Machine compares the data in chunks, for example a square of two by two positions.

A couple things should be noted. The training data and test data split was made to be about 90%/10%. Also, the dataset compositions were made into 10 K-Folds in order to mitigate the effect a lucky train/test-split and shuffle.

Details regarding the data within each dataset composition is listed in tables like Table 4.1 below, in each of the dataset compositions' chapters respectively. The following table contains information of the StandardEnd dataset composition without any duplication checks. "Total data" is the total amount of data in the dataset composition. "Avg. train/test ratio" shows the amount of test data compared to train ratio. This differs for each of the ten K-Folds and is therefore averaged between these K-Folds. "Loss ratio", "Win ratio" and "Draw ratio" shows the amount of each specific result compared to the rest of the results.

StandardEnd Without Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
22578	11.11%	21.2%	16.0%	62.8%

Table 4.1: Dataset data of StandardEnd without dupe check.

4.3.1 StandardEnd

StandardEnd is the most basic of all the dataset compositions created for this research. Its generation code used the base structure listed in 4.2.1 Base structure, but it could just as well have used the base structure for multiple boards shown in 4.2.2 Base structure for extraction of multiple boards; but without the addition of non-final boards. This dataset composition simply contain boards generated by performing every move in each match found in OCA 2.0 as well as the result. This dataset composition therefore strictly contain endgame boards. This dataset composition were put together both without and with duplicates, and more in-depth information of both of these datasets can be seen in Table 4.2 and Table 4.3 respectively.

With Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
21701	11.11%	21.9%	16.6%	61.5%

Table 4.2: Dataset data of StandardEnd with dupe check.

Without Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
22578	11.11%	21.2%	16.0%	62.8%

Table 4.3: Dataset data of StandardEnd without dupe check.

As this dataset was the simplest dataset composition in this research, the information given in the tables above is valuable for understanding what type of data is available in OCA 2.0. As can be seen in the tables above; OCA 2.0 does contain matches that mostly have a unique ending and the amount of end-game boards that ended in a win for Player 1 is quite low.

4.3.2 EndSecondThird

As OCA 2.0 represents matches by a series of moves and not simply a final board, it is possible to have each match/datapoint represented by multiple datapoints in a dataset composition by simply not performing all the moves. EndSecondThird has utilized this in order to generate more data for the Tsetlin Machine to hopefully learn better. EndSecondThird contains the final boards just as StandardEnd, but also contains boards for the second-last and third-last rounds resulting in a dataset composition that should in theory contain three times as many boards as StandardEnd. This was done by not only including the final boards, but also boards where the moves of the last round has been omitted and boards where both the moves of the final round and second final round has been omitted. Naturally, this resulted in more data. This dataset composition were put together both without and with duplicates, and more in-depth information of both of these datasets can be seen in Table 4.4 and Table 4.5 respectively.

With Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
38143	10.8%	20.0%	16.7%	63.3%

Table 4.4: Dataset data of EndSecondThird with dupe check.

Without Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
67745	11.11%	21.2%	16.0%	62.8%

Table 4.5: Dataset data of EndSecondThird without dupe check.

By comparing the amount of data of both versions of EndSecondThird, it can be seen that this dataset composition does indeed contain more data than StandardEnd. A significant amount of data however; are duplicates.

4.3.3 EndToEight

EndToEight can be considered an extension of EndSecondThird. EndSecondThird contain nine boards for each board in OCA 2.0. One with all moves performed, one with one round worth of moves removed and so on until eight moves have been removed. This was the largest dataset composition used in this research. This dataset composition were put together both without and with duplicates, and more in-depth information of both of these datasets can be seen in Table 4.6 and Table 4.7 respectively.

With Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
183701	10.61%	23.0%	17.4%	59.7%

Table 4.6: Dataset data of EndToEight with dupe check.

Without Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
203302	11.11%	21.2%	16.0%	62.8%

Table 4.7: Dataset data of EndToEight without dupe check.

Naturally, even more duplicates exist in this dataset composition than the other dataset compositions.

4.3.4 StandardPureMetaData

StandardPureMetaData differs from the previously mentioned dataset compositions. As the name implies, this dataset composition does not contain boards; but data about boards; hence the name ”metadata”. In short, StandardPureMetaData was the dataset composition named ”StandardEnd”; transformed into somewhat more vague descriptions of boards. Each data point in StandardPureMetaData consists of seven attributes as well as the result of the board it was extracted from. The attributes were adapted/interpreted from a github repository by the user SamRagusa [37]. The attributes are listed below.

1. Number of own uncrowned pieces
2. Number of opponent uncrowned pieces
3. Number of own kings
4. Number of opponent kings
5. Number of own pieces (crowned and uncrowned) on left and right edges of board
6. Integer value of own vertical center of mass
7. Integer value of opponent vertical center of mass

The four first attributes are quite self-explanatory. However; the fifth, sixth and seventh attributes might be a bit more open to interpretation. Figure 4.8 shows what squares were considered edges when own pieces for attribute number five were counted. The sum of owned pieces, both crowned and uncrowned, in the squares marked with the red circles is what made up attribute number five.

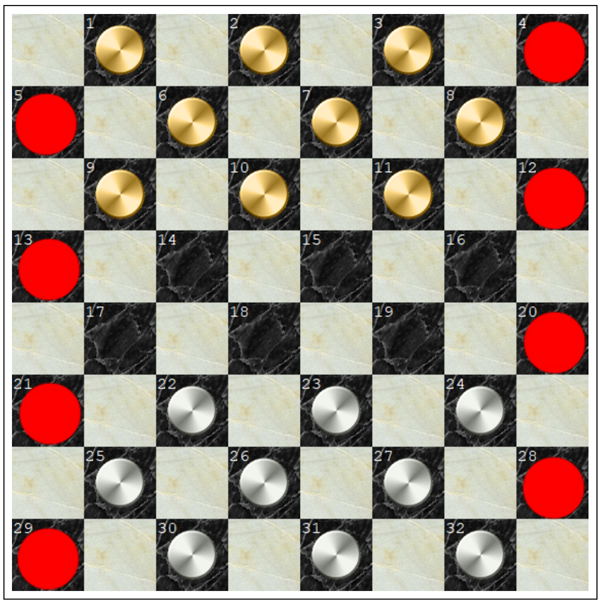


Figure 4.8: A visual representation of the left and right edges taken into account for metadata attribute number five. Picture adapted from CheckerBoard [1].

Metadata number six and seven were calculated the same way, except metadata number six only accounted for own(black) pieces while metadata number seven only accounted for the opponent's(white) pieces. The center of mass was for simplicity's sake rounded to a single row, and not for example somewhere in between row three and four. In order to find the vertical center of mass for a player's pieces, the following were performed. For each row, as shown in Figure 4.9, a weight score was gathered. Pawns were worth one "point" while kings were worth two "points".

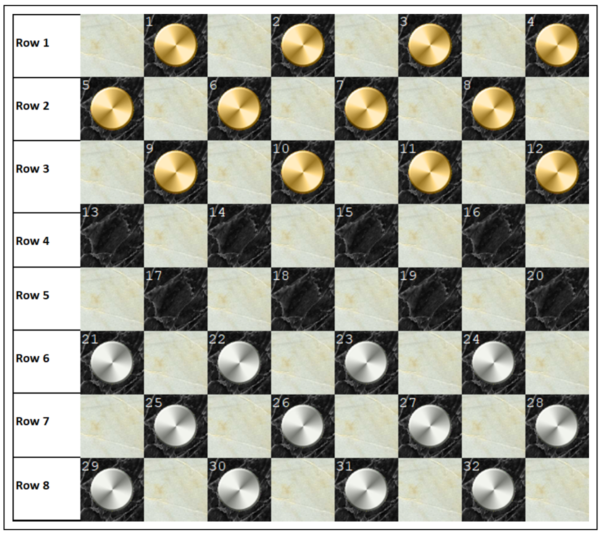


Figure 4.9: A visual representation of the rows the checkers board was split into when finding metadata number six and seven. Picture adapted from CheckerBoard [1].

The sum of these values were stored, which will be referred as *verticalsum*. Another value, referred to as *verticalmultiplied*, took into account which row the value was gathered from as well as the value of the pieces on that row. *verticalmultiplied* was the sum of the scores for each row multiplied with the row number the score came from; scores from row number one were multiplied by one before being added to *verticalmultiplied*, scores from row number two were multiplied by two and so on.

The final equation was as follows:

$$Verticalweights = roundDown(\frac{verticalmultiplied}{verticalsummed})$$

The answer to this equation gave the vertical center of mass of a checkerboard for a given player.

As StandardPureMetaData does not consist of boards directly, the Convolutional Tsetlin Machine can not be used. This dataset composition were put together both without and with duplicates, and more in-depth information of both of these datasets can be seen in Table 4.8 and Table 4.9 respectively.

With Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
6907	11.11%	29.9%	25.9%	44.3%

Table 4.8: Dataset data of StandardPureMetaData with dupe check.

Without Duplication Check

Total data	Avg. train/test ratio	Loss ratio	Win ratio	Draw ratio
22578	11.11%	21.2%	16.0%	62.8%

Table 4.9: Dataset data of StandardPureMetaData without dupe check.

The size difference between the two variations of StandardPureMetaData is very noticeable as 15671 datapoints were removed in the duplication check for the version with removal of duplicates. The reason for there being a lot of duplicates is that this dataset composition contains generalizations of the boards instead of the complete boards themselves. If each data point had more varied and/or more attributes, more datapoints would be unique.

4.4 Predictor

The Tsetlin Machine has, as mentioned in Chapter 2.2, quite a few configurations and hyper-parameters. In the process of finding the configuration of the Tsetlin Machine most suited for predicting the outcomes of Checkers boards, multiple configurations of the Tsetlin Machine was tested mainly on the dataset composition StandardEnd; hyper-parameter testing was performed on the configurations individually and their K-Fold accuracies were compared. All Tsetlin Machine configurations tested during this research were configurations of the multiclass Tsetlin Machine, which allows for predicting multiple outcomes of boards: loss, win and draw. The Non-convolutional Tsetlin Machine with weighted clauses and positive boost enabled proved to be more accurate than other Tsetlin Machine configurations tested. The dataset composition making this Tsetlin Machine configuration the most accurate, after further dataset composition specific hyper-parameter testing, proved to be StandardEnd without duplication checks (NoDupeCheck StandardEnd). Furthermore; the third K-Fold was shown to be the dataset the chosen Tsetlin Machine was the most accurate for when trained and tested against. Table 4.10 and 4.11 shows an advanced view of the accuracies for each of the three outcomes when testing against this dataset for the Checkers predictor for Player 1 (Black) and Player 2 (White) respectively. Precision refers to how accurate the predictions are, recall refers to the amount of the specific result was correctly predicted and the FScore is a result of the two former accuracies which can be read more about in Chapter 2.5.

	Precision	Recall	FScore
Win	57%	44%	49%
Loss	67%	51%	58%
Draw	76%	87%	81%

Table 4.10: Accuracy data for Tsetlin predictor, Player 1 (Black) trained and tested on NoDupeCheck StandardEnd K-Fold number 3 where it reached the accuracy of 72,21% during training.

	Precision	Recall	FScore
Win	60%	52%	55%
Loss	62%	49%	55%
Draw	77%	85%	81%

Table 4.11: Accuracy data for Tsetlin predictor, Player 2 (White) trained and tested on NoDupeCheck StandardEnd K-Fold number 3 with inverted loss and win data results where it reached the accuracy of 72,12% during training.

While the Tsetlin Machine for Player 1 (black) was trained on a particular K-Fold of NoDupeCheck StandardEnd, the Tsetlin Machine for Player 2 (white) trained on an an altered version of this K-Fold where losses and wins were inverted in an attempt to make sure the data did not favor the black player when training a Tsetlin Machine to play in the position of the white player.

It is worth noting that only the mean accuracy for all ten K-Folds were used for comparing different configurations of the Tsetlin Machine, not recall, precision and FScore nor by testing a single K-Fold. The hyper-parameters for both predictors are shown in Figure 4.12.

Clauses	Treshold	S
19000	40000	9

Table 4.12: The hyper-parameters used for both Checkers predictors.

4.5 Checkers Player

The Checkers player itself functions using tree search, utilizing both predictions of the Tsetlin Machine and rules for picking what move to make. The Tsetlin Machine predictors used is mentioned in Chapter 4.4, where Player 1 (black) is used whenever the Checkers player plays as the black player and Player 2 (white) is used whenever the Checkers player plays as the white player. The Checkers player has functionality for self-play, play against a machine selecting random moves and play against a human opponent; for both sides. The functionality for playing against the Tsetlin Machine requires the use of the command line interface as no GUI has been developed. The Checkers player visualizes Checkers boards as shown in Figure 4.10 where **1** represents black pawns, **2** represents black kings, **3** represents white pawns and **4** represents white kings.

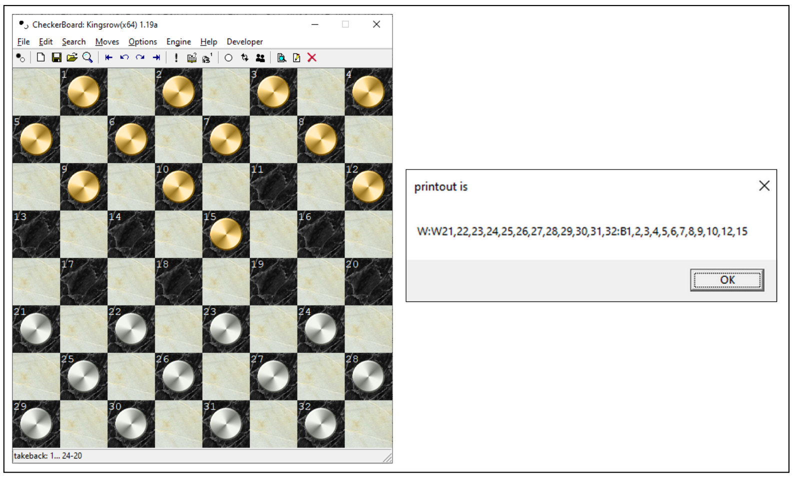
```
  1   1   1   1
1   1   1   1
  1   1   1   1
0   0   0   0
  0   0   0   0
3   3   3   3
  3   3   3   3
3   3   3   3
```

Figure 4.10: Checkers board as represented by the Tsetlin Checkers player.

The moves played by the human opponent have to be input as a complete board, in one of two formats. Either by the format used by CheckerBoard as demonstrated in Figure 4.12, which allows for generating this format from the board it shows in its GUI, making it simple to extract the board for input to the Tsetlin Checkers player. If CheckerBoard is not available for the human player, a more visual input method of input may be used instead; as shown in in Figure 4.11.

```
  1   1   1   1
1   1   1   1
  1   1   0   1
0   0   1   0
  0   0   0   0
3   3   3   3
  3   3   3   3
3   3   3   3
```

Figure 4.11: Checkers board as accepted as input without the use of CheckerBoard.


CheckerBoard: Kingsrow(x64) 1.19a
File Edit Search Moves Options Engine Help Developer
takeback: 1... 24-20

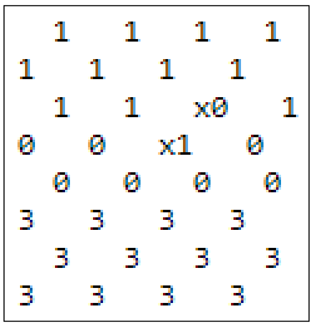
printout is

W:W21,22,23,24,25,26,27,28,29,30,31,32:B1,2,3,4,5,6,7,8,9,10,12,15

OK

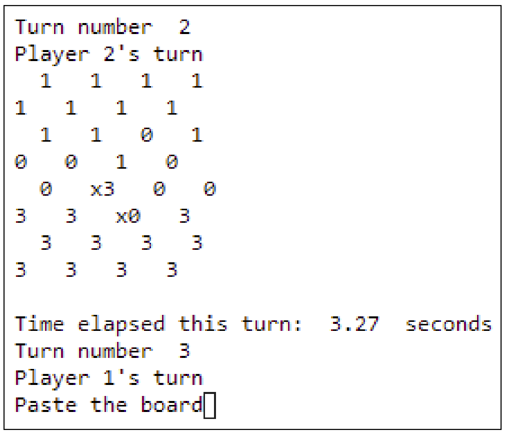
Figure 4.12: Checkers board as accepted as input by the use of CheckerBoard. Picture adapted from CheckerBoard [1].

Both input methods will show the user what squares was changed as shown in Figure 4.13. While CheckerBoard does stop the human opponent from making illegal moves, the Checkers player has no such rule checks for the human Checkers player for either of the two input methods. Figure 4.14 show how the move the Tsetlin Checkers player performs is portrayed.



```
  1   1   1   1
1   1   1   1
  1   1   x0   1
0   0   x1   0
  0   0   0   0
3   3   3   3
  3   3   3   3
3   3   3   3
```

Figure 4.13: Changes from the previous Checkers board displayed by the Tsetlin Checkers player.



```
Turn number  2
Player 2's turn
  1   1   1   1
1   1   1   1
  1   1   0   1
0   0   1   0
  0   x3   0   0
3   3   x0   3
  3   3   3   3
3   3   3   3

Time elapsed this turn:  3.27  seconds
Turn number  3
Player 1's turn
Paste the board
```

Figure 4.14: The move the Tsetlin Checkers players has performed as shown in its command line interface.

4.5.1 Structure

The whole solution consists of a single script; which loads the Tsetlin Machine predictor(s), handle the turn order between both players and contain the Checkers player itself. Even though the Checkers player is contained within this Checkers script, it is highly independent and could easily be replaced by two human players for example.

The Checkers player use a combination of rules and a Tsetlin Machine predictor to pick the optimal move. This is guided by a tree search algorithm performing Left-To-Right Breadth-first search with a depth of three moves. The search tree in question is not a preexisting tree, but a tree to be built while performing the tree search by searching for possible moves to perform, for both players. The search is performed as follows: the Checkers player's legal moves are found, the opponent's legal moves are found for these and then the Checkers player finds legal moves it can react with before the Tsetlin Machine predictor predict whether the resulting Checkers boards are a loss (**0**), a draw (**0,5**) or a win (**1**). These scores are averaged and represent the value of the move that lead to these boards, which is the possible moves the opponent could perform. The values of the possible moves the opponent could perform are then averaged giving a value to each of the possible moves the Checkers player could perform. The move with the highest score is then selected, resolving ties with random selection. The Left-To-Right Breadth-

First search order is illustrated in Chapter 2.4. In code, all moves are stored as boards where the move in question has been performed. Each node has both this board/move and the value/score of it. If the score is not resolved yet and the node in question is not a leaf node, it will contain both the move it represents as well as its child's moves in code. This is due to the whole tree is stored in a single list.

Additionally, guaranteed wins are given a score of **9** and guaranteed losses are given a score of **-9**. Guaranteed losses may happen in depth two if the opponent cannot perform any legal move in response to the Checkers player's move. Guaranteed losses may happen in depth three if the Checkers player cannot perform any legal move in response to a possible move performed by the opponent. The implemented Tsetlin Machine predictor has no way of stating that a move is guaranteed to lead to a win or a loss as all predicted losses, draws and wins are treated the same respectively. Placing such an extreme value hinders the Checkers player from committing huge mistakes. The term "committing" is used as the Tsetlin Machine might have an internal sense of how correct it think it is if it has implemented weights which could, if interpreted correctly, allow it to not only tell what the result of the board most likely will be; but also how certain it is in its prediction.

Figure 4.15 shows an illustration of how the tree may look after reaching depth three, showing only the moves. This illustration is not simplified, these are the actual moves that would be found in the tree search if the "Current Checkers Board" were to be input. Normally, the tree would be much larger as only two pieces are left on the board in this illustration.

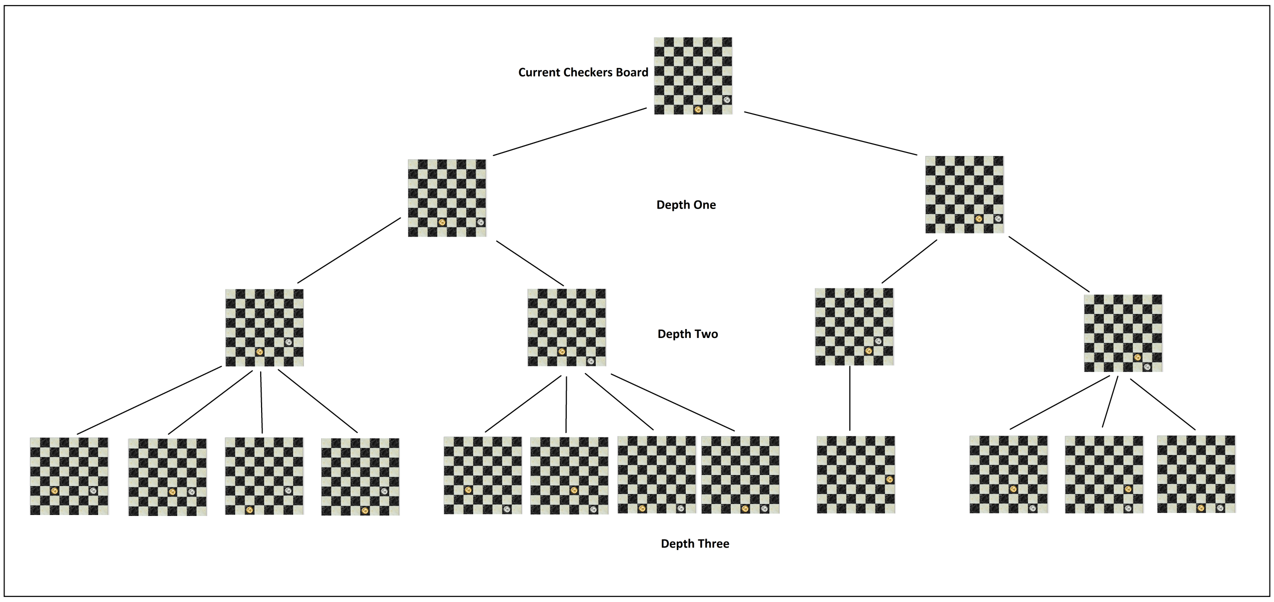


Figure 4.15: The tree search process performed by the Checkers player, showing only the moves (boards). Board pictures reprinted from CheckerBoard [1]

Figure 4.16 shows an illustration of the tree search, showing how the score values are given and how they affect the final decision of what move to make. This illustration is a bit extravagant as having this many moves leading to guaranteed losses or wins (values of -9 and 9) is not common. Having a tree search searching even deeper would likely provide more accurate predictions making the Checkers player play better, however; the cost of processing time would increase drastically the deeper the search is performed.

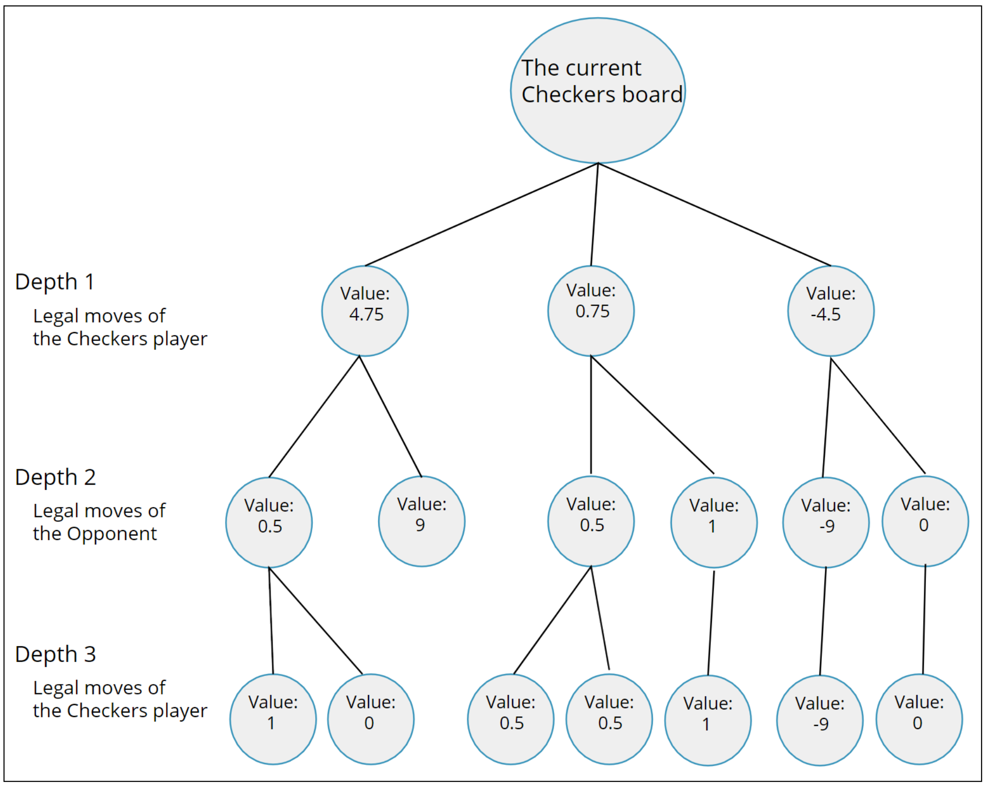


Figure 4.16: The tree search process performed by the Checkers player, showing how predictions affects the choice of what move to make.

The process of finding legal moves follows the rules of Checkers where moving a piece may only be performed if no capture moves may be made. For the simplicity of programming's sake, this is performed for every piece; omitting non-capture moves if any legal capture moves was found after finding all legal moves. If a capture move leads to another possible capture, this capture move must be performed; continuing until the piece in question can perform no more capture moves. This may be called a chain-capture, or multi-capture. If a piece is able to perform multiple capture moves, within or outside of chain-capture moves, all the legal variations of the captures the piece may perform is stored as separate moves as all of them lead to different boards once performed. Below, a more structured explanation of how legal moves are found is listed.

Algorithm 4 The Process of Finding Legal Moves

```
1: Find all owned pieces
2: for Each of the owned pieces do
3:     Find all possible moves
4:         Find all possible directions the piece is allowed to move
5:         Find all legal capture moves for this piece
6:             for Each legal capture move for this piece do
7:                 Continue each capture move completely
8:                 Store each legal path
9:             end for
10:         if No legal capture moves for this piece is found then
11:             Find all legal moves for this piece; if any
12:         end if
13:         if any legal capture moves for this piece is available then
14:             Store these
15:         end if
16: end for
17: if any legal capture moves are found for any piece then
18:     These are considered legal moves
19: end if
20: if no legal capture moves are found then
21:     Non-capture moves are considered legal; if any
22: end if
```

Part III

Experiments and Results

Chapter 5

Tests and Findings of the Tsetlin Machine

This chapter show both the capabilities of the Tsetlin Machine in this domain, as well as show how the testing for finding the best Tsetlin Machine configuration and dataset composition was performed. But first, here are some general information regarding all the tests of the Tsetlin Machine shown in this chapter.
Tsetlin Machine configurations refers to the various settings, or configurations, the Tsetlin Machine may have; such as having weights enabled, positive boost and being convolutional.
All Tsetlin Machines are multiclass, they may output **0** (loss), **1** (win) or **2** (draw).
Tsetlin Machines are trained and tested by K-Folds, more precisely; ten copies of the same data where the difference is dictated by the split between train and test data.
The Tsetlin Machines are compared by their accuracy for a given dataset composition. This accuracy may sometimes be referred to as ”mean accuracy”. The accuracy refers to the average of the accuracies achieved in the test-procedure during training on each of the ten K-Fold datasets.
Training are run with the adaptive epochs method as shown in Chapter 5.2, with the maximum amount of epochs set to 500 unless specified otherwise.
NoDupeCheck refers to the version of the given dataset composition which had no duplicate data removed during its creation. For example; NoDupeCheck StandardEnd is the version of StandardEnd that includes all duplicates.

Additionally; the Tsetlin Machine has an element of random within. This means; by training a Tsetlin Machine multiple times without any change of dataset, configuration or hyper-parameters, the achieved accuracy may not be the same. This is explained further in Chapter 5.1.

5.1 Hyper-parameter Testing

An important part of making the Tsetlin Machine the most accurate it can be, is to provide it with the optimal hyper-parameters. What the optimal hyper-parameters are is affected by the configuration of the Tsetlin Machine as well as the dataset it is trained and tested on. The process of finding optimal, or near optimal, hyper-parameters for the Tsetlin Machine relied on a rough estimation using past experience and methodical testing. An initial set of hyper-parameters was set, and one hyper-parameter was tweaked at a time. The Tsetlin Machine was trained multiple time, where the hyper-parameter in focus was the only hyper-parameter to be tweaked between each run. A single script typically was set to run between 3 to 7 different values for the hyper-parameter in focus. The values typically varied between a third of the guessed value to a number as large as three times larger, in ascending order. If the accuracy continue to increase the lower, or the higher, the value; more tests with the same hyper-parameter in focus was performed with numbers covering the direction providing the higher accuracy. This was continued until the accuracy started to decrease. Then the hyper-parameters with the best accuracy was selected and the procedure continued with another hyper-parameter in focus.

The combination of hyper-parameters might be just as important as their individual values, making this procedure unsuited for finding the absolute best set of hyper-parameters. However; in order to find the absolute best set of hyper-parameters, every possible combination of hyper-parameters; within reason, might have to be investigated. Given the time cost and likely low improvement in accuracy, this might not be worth it.

There are some general effects that are noticed, most of which can be seen by seeing the figures in Chapter 5.3. The lower the amount of clauses, the faster the training will be at the cost of accuracy. Having the optimal amount of clauses is better than having a high amount of them, increasing the amount of clauses too much negatively affect both accuracy and speed. Weighted variants of the Tsetlin Machine generally favors a higher amount of treshold than their non-weighted counterparts. Positive boost may affect the preferred hyper-parameters. Positive boost do generally boost the accuracy of the Tsetlin Machine.

When comparing two Tsetlin Machine configurations where the difference is reasonably small, such as *0.3* percent points, it might be the case that the conclusion of which one is better is wrong due to the randomness of the Tsetlin Machine. Table 5.1 show how varying the resulting accuracies can be using the best found Tsetlin Machine, dataset composition and hyper-parameters. The left-most column display the variance when comparing the achieved accuracies for each of the 10 K-Folds, over 10 runs of training the Tsetlin Machine. The right-most column display the variance when comparing the mean accuracy of the 10 K-Folds for each of the 10 runs respectively. The latter is the accuracy score typically used for comparing Tsetlin Machine results. As the variance for this accuracy is as low as *0.5*, a difference by more than one percent may be considered a clear lead; while a difference of *0.2* for example; is not sufficient to differ the two.

	Avg. for each full K-Fold	For each run
Variance	4.23	0.5

Table 5.1: Data of how many percent points the resulting accuracy varies. Data gathered from 10 train/tests, 100 K-Folds in total. Using the best found dataset composition, Tsetlin Machine configuration and hyper-parameters and adaptive epochs, max 200 epochs. Data from Table 5.2.

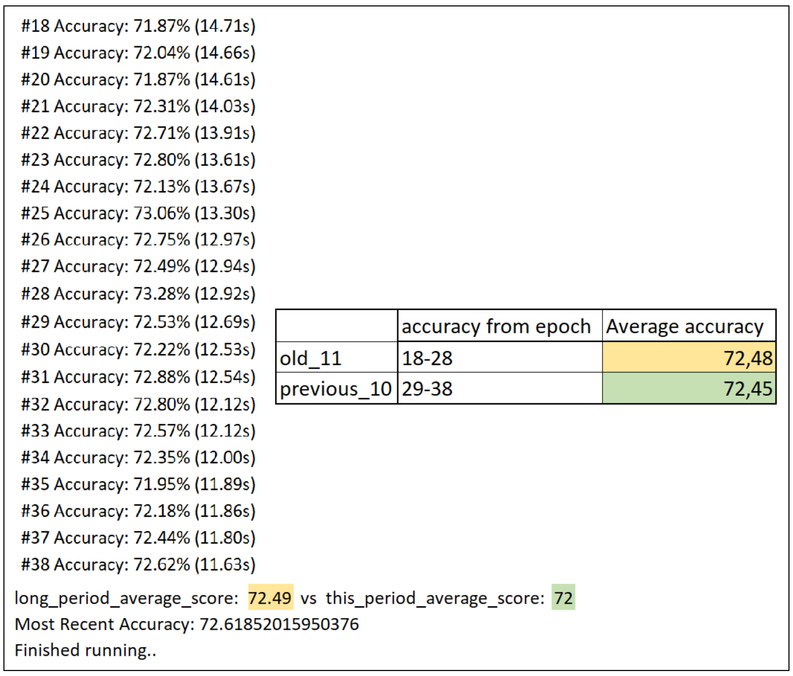
5.2 Adaptive epochs

Setting the optimal amount of epochs to be run is important, but is also more demanding than other hyper-parameters to tweak as studying the progress over a set amount of epochs is needed in order to investigate how the epoch should be tweaked. In order to make this process less demanding as well as making it more optimal, adaptive epochs was implemented.

Adaptive epochs was meant to allow the Tsetlin Machine to run for as many epochs as it need for getting optimal results. "As many epochs as it need" was defined as "as long as the accuracy increases". Two implementations of adaptive epochs were tested, but the second version proved to be more reliable and was therefore used for the majority of this research. Both versions cut off at 500 epochs as 50 has been proven to be overkill in similar projects as well as early testing of early versions of the Checkers dataset. Both methods compare the accuracy of recent epochs with the accuracy of less recent epochs. The epochs are stored in a list, if there is no data recorded yet; the accuracy is considered 0 for the missing epochs. As long as the accuracy of an epoch is not way below the previous epochs' accuracy, both versions of adaptive epochs will run at least 20 (21 for the second version) epochs as at least one value of zero would be in the list of the accuracies of the less recent epochs which has a significant impact on the average of the list.

- **First version:** if the average of the 20 last epochs is higher than the average of the 10 last epochs, no more epochs are run for that K-Fold.
- **Second version:** if the average of the 10 previous epochs are lower than the average of the 11 epochs before those, no more epochs are run for that K-Fold.

The second version seem to increase the accuracy by 1-2 percent with the same parameters and dataset when compared to the first version. Running 200 epochs proved to give worse results for tests run by the use of the same dataset with the same hyper-parameters without the use of adaptive epochs; StandardEnd, Clauses: 15000, Treshold: 12000, S: 25, Epochs: 200. The adaptive epoch testing ended after around 40-60 epochs per k-fold, but had the ability to continue up to 500 epochs if consistent improvement was shown. Figure 5.1 show what information the adaptive epochs, version two, function store when it decide to cut off training.



```
#18 Accuracy: 71.87% (14.71s)
#19 Accuracy: 72.04% (14.66s)
#20 Accuracy: 71.87% (14.61s)
#21 Accuracy: 72.31% (14.03s)
#22 Accuracy: 72.71% (13.91s)
#23 Accuracy: 72.80% (13.61s)
#24 Accuracy: 72.13% (13.67s)
#25 Accuracy: 73.06% (13.30s)
#26 Accuracy: 72.75% (12.97s)
#27 Accuracy: 72.49% (12.94s)
#28 Accuracy: 73.28% (12.92s)
#29 Accuracy: 72.53% (12.69s)
#30 Accuracy: 72.22% (12.53s)
#31 Accuracy: 72.88% (12.54s)
#32 Accuracy: 72.80% (12.12s)
#33 Accuracy: 72.57% (12.12s)
#34 Accuracy: 72.35% (12.00s)
#35 Accuracy: 71.95% (11.89s)
#36 Accuracy: 72.18% (11.86s)
#37 Accuracy: 72.44% (11.80s)
#38 Accuracy: 72.62% (11.63s)
long_period_average_score: 72.49 vs this_period_average_score: 72
Most Recent Accuracy: 72.61852015950376
Finished running..
```

	accuracy from epoch	Average accuracy
old_11	18-28	72,48
previous_10	29-38	72,45

Figure 5.1: Weighted Tsetlin Machine with positive boost, clauses 19000, treshold 40000, s 9, NoDupeCheck StandardEnd using adaptive epochs. Text to the left show printouts during training of the Tsetlin Machine with the achieved accuracies for each epoch. The table show what information the adaptive epochs function store. This training was cut off at 38 epochs due to adaptive epochs kicking in.

Further testing was continued once it was decided what was the best found dataset composition, Tsetlin Machine configuration and hyper-parameters. The following tables was found by testing the weighted Tsetlin Machine with positive boost using the dataset composition NoDupeCheck StandardEnd with the following hyper-parameters: *clauses* 19000, *treshold* 40000, *s* 9 and a maximum epochs of 200. Table 5.2 show the advantage of using adaptive epochs. The data was gathered from running ten train/test cycles, of ten K-Folds each. Adaptive epochs was programmed not to stop the training, but to store the accuracy it would have ended with if adaptive epochs was enabled. It shows the lowest achieved accuracy, highest achieved accuracy, the average accuracy and the difference between the highest and lowest achieved accuracy for both the use of adaptive epochs and without. This information is shown when using the resulting accuracy of each of the ten K-Folds for each of the ten runs (100 data points), but also when using the mean accuracies for each of the 10 full K-Fold runs in the two left-most columns (10 data points).

For each K-fold:	200 epochs	Adapt. Epochs	Avg. for each run:	200 epochs	Adapt. Epochs
Lowest	68.64%	69.7%		70.13%	71.38%
Highest	73.22%	73.93%		70.8%	71.88%
Average	70.46%	71.59%		70.50%	71.59%
Variance	4.58	4.23		0.67	0.5

Table 5.2: Comparison of no adaptive epoch vs adaptive epoch both for each run and for each individual K-Fold. Data gathered from 10 train/tests, 100 K-Folds in total. Using best found dataset composition, Tsetlin Machine configuration and hyper-parameters, 200 epochs.

During the ten train/test runs performed to gather data for Table 5.2, adaptive epochs would try to cut-off training at epoch number 29 the lowest, and epoch number 81 the highest which is an amount significantly smaller than 200. By studying this table, it can be seen that adaptive epochs beat training this Tsetlin Machine for 200 epochs. Table 5.3 show exactly how much better adaptive epochs are than training this Tsetlin Machine for 200 epochs. The numbers displayed are the difference between the scores for and without the use of adaptive epochs using the scores from Table 5.2 above.

For each K-fold:	Gain by Adapt. Epochs	Avg. for each run:	Gain by Adapt. Epochs
Lowest	1.06		1.25
Highest	0.71		1.08
Average	1.13		1.09
Variance	-0.35		-0.17

Table 5.3: Comparison of no adaptive epoch vs adaptive epoch both for each run and for each individual K-Fold. Data gathered from 10 train/tests, 100 K-Folds in total. Using best found dataset composition, Tsetlin Machine configuration and hyper-parameters, 200 epochs. Complimentary table for Table 5.2, showing the advantage of using adaptive epochs.

This table, Table 5.3, show that the use of adaptive epochs in this scenario improve the accuracy by more than one percent on average while providing just a little more consistent results, seen by comparing the *Average* and *Variance* scores respectively. Figure 5.2 show the average accuracy training was cut off at compared to the average accuracy achieved at each epoch for a single full 10 K-Fold run using the best found Tsetlin Machine, dataset and hyper-parameters. This graph show that adaptive epochs may be beaten by more fitting amounts of epochs, and could be tweaked to increase its effectiveness. However, the perfect amount of epochs varies for each K-Fold as seen in Figures 5.3 and 5.4 which makes it tricky to pick the perfect amount of epochs without the use of an adaptive method.

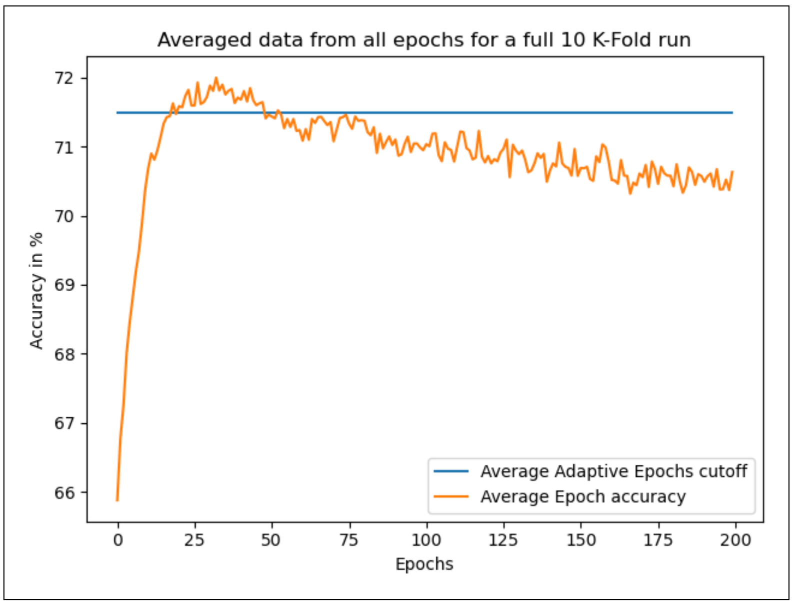


Figure 5.2: Weighted Tsetlin Machine with positive boost, clauses 19000, treshold 40000, s 9, dataset NoDupeCheck StandardEnd using adaptive epochs. Data fetched from a complete 10 K-Fold run. This graph show a blue line indicating at what percent the adaptive epochs function cut-off the training on average during the testing of the 10 K-Folds. The orange plot show the accuracy that was achieved during training at each epoch, averaged for each of the 10 K-Folds.

In the two figures below, Figure 5.3 and 5.4, the cutoff accuracy of the adaptive epochs are shown for two K-Folds ran for the same K-Fold evaluation session. The accuracies shown for these K-Folds show that the optimal amount of epochs to train for is different for both of these. Therefore; the optimal amount of epochs cannot be set manually.

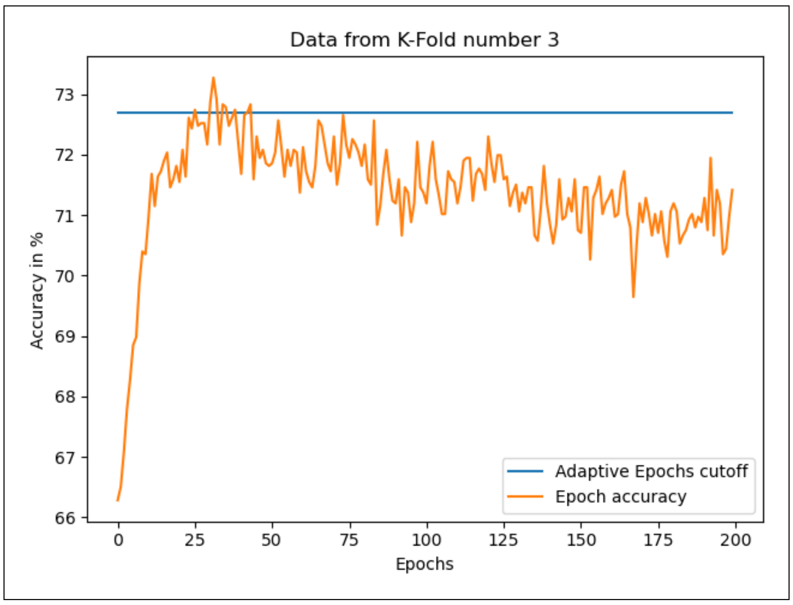


Figure 5.3: Weighted Tsetlin Machine with positive boost, clauses 19000, treshold 40000, s 9, dataset NoDupeCheck StandardEnd using adaptive epochs. Data fetched from K-Fold number 3. This graph show a blue line indicating at what percent the adaptive epochs function cut-off the training. The orange plot show the accuracy that was achieved during training at each epoch.

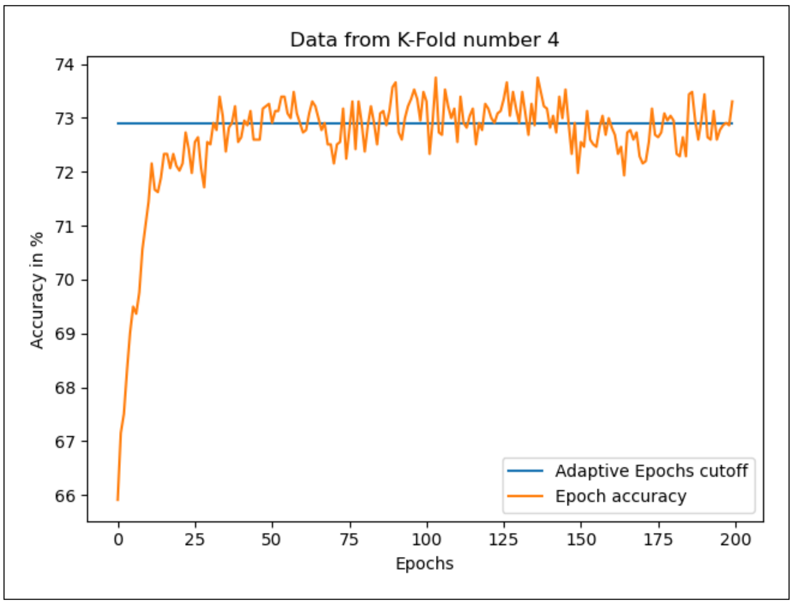


Figure 5.4: Weighted Tsetlin Machine with positive boost, clauses 19000, treshold 40000, s 9, dataset NoDupeCheck StandardEnd using adaptive epochs. Data fetched from K-Fold number 4. This graph show a blue line indicating at what percent the adaptive epochs function cut-off the training. The orange plot show the accuracy that was achieved during training at each epoch.

5.3 Tsetlin Machine Configurations

By testing on various dataset compositions, the weighted Tsetlin Machine with positive boost proved to be the most accurate Tsetlin Machine configuration; detailed in Chapter 5.4. The dataset compositions tested on was mainly StandardEnd, EndSecondThird and EndToEight; all of which were created using strict duplication removal. This Tsetlin Machine configuration was therefore chosen as the one Tsetlin Machine configuration to move forward with while developing and evaluating new dataset compositions.

Once the NoDupeCheck version of StandardEnd proved to be the dataset composition allowing the weighted Tsetlin Machine with positive boost to achieve the highest accuracy out of all the created dataset compositions as shown in Chapter 5.4; it would be interesting to see exactly how much higher of an accuracy this Tsetlin Machine configuration achieve compared to other Tsetlin Machine configurations for this dataset composition. This allowed for validating the theory of the weighted Tsetlin Machine with positive boost being the most accurate Tsetlin machine configuration for this domain, which turned out to be true.

5.3.1 Data

This chapter present the achieved accuracy of each Tsetlin Machine configuration tested with the best hyper-parameters found for that particular Tsetlin Machine configuration using the dataset composition NoDupeCheck StandardEnd, except the Tsetlin Machine result displayed in the first row of Table 5.4. This Tsetlin Machine use the best found hyper-parameters for the weighted Tsetlin Machine with positive boost, which was the Tsetlin Machine configuration which achieved the highest accuracy for this dataset composition. This data may be used for demonstrating the importance of well-fitted hyper-parameters, both for the specific dataset composition and for the specific Tsetlin Machine configuration. These accuracies was achieved by the use of adaptive epochs during training as explained in Chapter 5.2.

The *Tsetlin Machine configurations* and the three hyper-parameters used, in addition to the mean accuracy of the ten K-Folds is presented in Table 5.4. The *convolutional Tsetlin Machine configurations* used these settings for their search window: *Shape x:* 4, *Shape y:* 8, *Shape z:* 4, *Frame x:* 2 and *Frame y:* 2. All of these parameters are explained further in Chapter 2.2. The full name of each of the *Tsetlin Machine Configurations* presented in the table below, are displayed in Table 5.5

TM Configuration	Clauses	Treshold	s	Average accuracy
TM	19000	40000	9	56.84%
TM	12000	50	27	67.39%
wght TM	15000	12000	27	70.68%
TM pos	12000	50	27	68.16%
wght TM pos	19000	40000	9	71.69%
conv TM	16000	30000	30	64.07%
wght conv TM pos	10000	30000	30	67.23%
conv TM pos	10000	30000	30	64.53%
wght conv TM pos	10000	30000	30	67.16%

Table 5.4: Best found hyper-parameters, as well as the achieved accuracy, for various configurations of the Tsetlin Machine ran on the dataset composition: the NoDupeCheck version of StandardEnd.

Shortened Name	Full Name
TM	Tsetlin Machine
wght TM	Weighted Tsetlin Machine
TM pos	Tsetlin Machine with positive boost
wght TM pos	Weighted Tsetlin Machine with positive boost
conv TM	Convolutional Tsetlin Machine
wght conv TM	Weighted Convolutional Tsetlin Machine
conv TM pos	Convolutional Tsetlin Machine with positive boost
wght conv TM pos	Weighted Convolutional Tsetlin Machine with positive boost

Table 5.5: The full names of the Tsetlin Machine configurations presented in Table 5.4

5.3.2 Findings

There are numerous observations that may be made off of the performance of the Tsetlin Machine configurations presented in Table 5.4 in Chapter 5.3.1, these are presented below.

When comparing the accuracies achieved by the standard Tsetlin Machine, found in row one and two of Table 5.4, the configuration shown in row one; using the best found hyper-parameters for the weighted Tsetlin Machine with positive boost, achieved a far lower accuracy than when using the hyper-parameters presented in row two even though it used more clauses. This show that well-fitted hyper-parameters has a significant impact on the accuracy the Tsetlin Machine is able to achieve.

The best convolutional Tsetlin Machine configuration, the weighted Convolutional Tsetlin Machine, performs worse than the worst performing non-convolutional Tsetlin Machine; which is the Standard Tsetlin Machine. If the hyper-parameters, and the other variables, are close to optimal; then the Convolutional Tsetlin Machine is not as useful for predicting games in this domain as its non-convolutional counterpart.

Positive boost generally lives up to its name, during testing it normally showed some increase in the accuracy of various Tsetlin Machine configurations. This is reflected in Table 5.4 shown above, but the increase in accuracy for the various Convolutional Tsetlin Machines does not hold true. When comparing the accuracies of the Standard Tsetlin Machine and the weighted Tsetlin Machine with their respective counterparts with positive boost, it can be seen that positive boost increases the achieved accuracy by 0.89% on average. The weighted Convolutional Tsetlin Machine did not benefit from having positive boost enabled, showing a decrease in accuracy by 0.07%. But the Convolutional Tsetlin Machine do show an increase in accuracy with positive boost. The average increase by positive boost shown with the Convolutional Tsetlin Machine variants is 0.195%. It is worth noting that the results are heavily affected by the randomness of training the Tsetlin Machine; the weighted Convolutional Tsetlin Machine may have had a lucky training and the weighted Convolutional Tsetlin Machine with positive boost may have had an unlucky training. The randomness of training the Tsetlin Machine is described in Chapter 5.1.

Weights significantly increases the accuracy of all configurations of the Tsetlin Machine. When comparing the achieved accuracies of the Tsetlin Machines with their weighted counterparts, it can be seen that the average increase in accuracy is 3.15% and the increase in accuracy does not seem to be affected by the use of positive boost.

Given these findings, it should be clear why the weighted Tsetlin Machine with positive boost was chosen for the Checkers player's Tsetlin predictors.

5.4 Accuracy

This chapter shows how accurate the weighted Tsetlin Machine with positive boost is when predicting boards of the investigated dataset compositions. The weighted Tsetlin Machine with positive boost was shown to be the most accurate of all the tested Tsetlin Machine configurations in Chapter 5.3. This chapter will show that the dataset composition NoDupeCheck StandardEnd is the dataset composition allowing this Tsetlin Machine configuration to achieve the highest accuracy of the investigated dataset compositions. More dataset compositions was created and tested on, but these are the most promising and interesting.

The results of the Tsetlin predictors for Player 1 and for Player 2, shown in Tables 5.7, 5.8, 5.9 and 5.10, was achieved by the same pre-trained Tsetlin Machines used for the Checkers player's Tsetlin predictors. They were both trained on K-Fold number three and the scores shown was acquired by testing on the third K-Fold's test data as to not allow them to be tested on the exact same data they were trained on. All other scores shown was generated from a full ten K-Fold train/test. The reasoning for selecting the third K-Fold for the Checkers player's Tsetlin predictors to be trained on was due to the fact that K-Fold number three of NoDupeCheck StandardEnd was showing a general trend of providing a slightly higher accuracy than those of other K-Folds. Although, the accuracies achieved did vary significantly and no K-Fold in particular provided the highest accuracy consistently. Table 5.6 shows the accuracies achieved for each K-Fold by the weighted Tsetlin Machine with positive boost on NoDupeCheck StandardEnd using its best found hyper-parameters: *clauses:* 19000, *treshold:* 40000 and *c:* 9, where it can be seen the highest accuracy achieved was achieved on K-Fold number three.

K-Fold	1	2	3	4	5	6	7	8	9	10
Acrcy	70.77%	72.4%	72.57%	72.29%	71.35%	71.18%	72.33%	71.72%	71.07%	71.26%

Table 5.6: Accuracy (Acrcy) for each of the ten K-Folds of NoDupeCheck StandardEnd achieved by the weighted Tsetlin Machine with positive boost, using the hyper-parameters Clauses: 19000, Treshold: 40000, S: 9, achieving the mean accuracy of 71.69%.

5.4.1 Data

This Chapter contains accuracies achieved by the weighted Tsetlin Machine with positive boost for each of the investigated datasets, and also shows the hyper-parameters used in order to achieve these results. This information is found in Table 5.7. Additionally, more specialized scores are also displayed in order to better understand how good this Tsetlin Machine is for predicting the three outcomes; win, loss and draw. These scores may be referred to as *STATS* for simplicity. This information can be found in Table 5.8 for win STATS, Table 5.9 for loss STATS and Table 5.10 for draw STATS. The STATS scores are *precision*, *recall* and *FScore* as explained in Chapter 2.5. These three stats explained shortly in terms of the accuracy of predicting wins: *precision* show the percentage of predicted wins being actual wins, *recall* show the percentage of wins which was predicted to be wins and *FScore* is a way of summarizing both *precision* and *recall* into one score by calculating their harmonic mean [17][18].

Dataset Composition	Clauses	Treshold	s	Average accuracy
StandardEnd	15000	40000	15	69.79%
EndSecondThird	13000	80000	20	69.41%
EndToEight	15000	160000	68	63.38%
StandardPureMetaData	20000	40000	9	48.39%
NoDupeCheck StandardEnd	20000	40000	9	71.35%
NoDupeCheck StandardEnd	19000	40000	9	71.44%
$\hat{\text{P}}$layer 1	19000	40000	9	72.21%
$\hat{\text{P}}$layer 2	19000	40000	9	72.12%
NoDupeCheck EndSecondThird	15000	40000	15	70.65%
NoDupeCheck EndToEight	60000	80000	40	67.73%
NoDupeCheck StandardPureMetaData	15000	40000	15	66.53%

Table 5.7: Best found hyper-parameters, as well as the achieved accuracy of the weighted Tsetlin Machine with positive boost, for various dataset compositions. Player 1 and Player 2 was trained and tested on NoDupeCheck StandardEnd's K-Fold number three

Dataset Composition WIN	Precision	Recall	FScore
StandardEnd	58%	36%	44%
EndSecondThird	59%	32%	41%
EndToEight	54%	23%	32%
StandardPureMetaData	47%	32%	48%
NoDupeCheck StandardEnd	59%	42%	49%
NoDupeCheck StandardEnd	59%	41%	48%
$\hat{\text{P}}$layer 1	57%	44%	49%
$\hat{\text{P}}$layer 2	60%	52%	55%
NoDupeCheck EndSecondThird	60%	38%	46%
NoDupeCheck EndToEight	53%	31%	39%
NoDupeCheck StandardPureMetaData	56%	23%	32%

Table 5.8: STATS, from Chapter 2.5, for predicting wins. Player 1 and Player 2 was trained and tested on NoDupeCheck StandardEnd's K-Fold number three.

Dataset Composition LOSS	Precision	Recall	FScore
StandardEnd	62%	48%	54%
EndSecondThird	61%	40%	48%
EndToEight	58%	32%	41%
StandardPureMetaData	49%	41%	45%
NoDupeCheck StandardEnd	62%	52%	56%
NoDupeCheck StandardEnd	62%	52%	57%
$\hat{\text{P}}$layer 1	67%	51%	58%
$\hat{\text{P}}$layer 2	62%	49%	55%
NoDupeCheck EndSecondThird	62%	47%	54%
NoDupeCheck EndToEight	57%	43%	48%
NoDupeCheck StandardPureMetaData	55%	32%	40%

Table 5.9: STATS, from Chapter 2.5, for predicting losses. Player 1 and Player 2 was trained and tested on NoDupeCheck StandardEnd's K-Fold number three.

Dataset Composition DRAW	Precision	Recall	FScore
StandardEnd	73%	87%	79%
EndSecondThird	72%	89%	80%
EndToEight	65%	88%	75%
StandardPureMetaData	49%	63%	55%
NoDupeCheck StandardEnd	76%	86%	80%
NoDupeCheck StandardEnd	76%	86%	81%
$\hat{\text{P}}$layer 1	76%	87%	81%
$\hat{\text{P}}$layer 2	77%	85%	81%
NoDupeCheck EndSecondThird	74%	87%	80%
NoDupeCheck EndToEight	72%	86%	78%
NoDupeCheck StandardPureMetaData	69%	90%	78%

Table 5.10: STATS, from Chapter 2.5, for predicting draws. Player 1 and Player 2 was trained and tested on NoDupeCheck StandardEnd's K-Fold number three.

5.4.2 Findings

Firstly, when comparing the accuracies of the two NoDupeCheck StandardEnd tests in Table 5.7, it can be seen that a higher amount of clauses does not necessarily allow the Tsetlin Machine to achieve a higher accuracy. This was widely shown during hyper-parameter testing.

Other than NoDupeCheck StandardEnd proving to allow this Tsetlin Machine to achieve the highest accuracy, the detailed accuracies tables, Tables 5.8, 5.9 and 5.10, also show that this Tsetlin Machine does not predict wins very accurately in any dataset composition shown. However, it has shown to be adept at predicting draws. This is most likely due to the data source; and therefore the dataset compositions, containing a large amount of draws as explained in Chapter 4.1.

Additionally, these tables show that the removal of duplicates had a negative impact on all dataset compositions (StandardEnd, EndSecondThird, EndToEight and StandardPureMetaData). Naturally, StandardEnd was affected the least. EndToEight and EndSecondThird was attempts at generating more data out of the limited data available. EndToEight contain more generated data by removing moves just as EndSecondThird, but had some boards with more moves removed compared to EndSecondThird as explained in Chapter 4.3. Logically, this lead to a huge amount of duplicate data which was removed. Additionally; equal boards with different results could have different outcomes which is believed to be even more detrimental to EndToEight with duplicates removed as in some cases; there could be three identical boards among its datapoints all with three different results which would likely teach the Tsetlin Machine that all three outcomes are just as likely. The NoDupeCheck version would allow for multiple identical boards in this case, teaching the Tsetlin Machine which outcome is more likely. StandardPureMetaData do not contain generated data, but is built by the principle of generalization of the boards. This generalization created a large amount of duplicate data which made the removal of duplicates heavily affect this dataset composition and its accuracy. It can be concluded that the removal of duplicate data has a negative impact on the achieved accuracy.

Chapter 6

Tests and Findings of the Checkers Player

This chapter show some of the findings learned through both self-play and through playing against other Checkers players. This is useful both for finding ways to improve the Checkers player and to test its skill. A draw rule misinterpretation is also explained in Chapter 6.1, as this misinterpretation negatively affect the accuracy of the results of all games.

6.1 Draw Rule Misinterpretation

If a player is unable to make any legal move, the player might be out of pieces meaning the player lost. It was not taken into consideration that whenever a player is unable to move any pieces due to their pieces being blocked in, the match should end in draw. In this implementation, having all blocked in pieces resulted in a loss to the player unable to perform any legal move. This rule misinterpretation makes some matches where the result should have been a draw, but was concluded to be a win for the opponent instead. Luckily, a match very rarely ends by blocking all of a player's pieces in. A thousand self-plays has been run, as seen in Figure 6.1, where only three cases was recorded. Additionally; over thirty matches was played with manual observation recording no cases of a match ending in this way. In other words, on average; 0.3% of self-play games is ended with the wrong result. The effect is less if the Tsetlin Checkers player is playing against another machine Checkers player which do enforce this rule correctly. The two Checkers players used in this experiment are the same Checkers player made for this research, using the two pre-trained Tsetlin predictors mentioned in Chapter 5.4 achieving accuracies of 72.21% and 72.12% respectively.

1000 matches	Player 1 wins	Player 2 wins	Draws
Achieved results	717	137	146
Correct results	716	135	149
Change in total	0.3%		

Table 6.1: 1000 matches between of self-play using the Checkers player with the use of the Tsetlin predictors of Player 1 and Player 2 as explained in Chapter 5.4.

It is worth noting that the Tsetlin Checkers player has a mechanism in its tree search where the value of 9 is set for all moves making the opponent unable to perform any legal moves, which increase the chance of this happening as this mechanism do not differentiate between having no pieces left and having all pieces blocked in. The former should be counted as a loss and the latter as a draw, but both are counted as loss in the implemented Checkers player. Therefore, this rule misinterpretation is more likely to happen when the Checkers player is playing against itself than other players as both Tsetlin Checkers players would be inclined to pick moves making the opponent unable to make any legal moves.

6.2 Learnings of Self-Play

The Checkers player's self-play capability was frequently used during this research in order to compare different implementations of the Checkers player itself, and also; to compare different Tsetlin Machine predictors to each other. This chapter discuss some of the findings of self-played Checkers matches, most of which lasted 100 games. Most of the Checkers matches used the Tsetlin Machine predictor for Player 1 or the Tsetlin Machine for Player 2 which are described further Chapter 4.4. Both the Tsetlin Machine predictor for Player 1 and Player 2 were trained using the third K-Fold of the dataset composition NoDupeCheck StandardEnd, although Player 2 was trained with losses and wins reversed in order to train in the place of the white player. Both used the hyper-parameters: *clauses:* 19000, *treshold:* 4000, *s:* 9 and achieved the accuracy of 72.21% and 72.12% respectively. No side-switch was performed during the 100 games presented in this chapter. Each table presented in this chapter shortly show what type of Checkers players was playing against each other, the amount of wins each player achieved, as well as draws.

The Checkers player playing against itself using the Tsetlin Machine for Player 1 and Player 2 respectively show that Player 1 has a clear lead, as seen in Table 6.2. It is unclear whether this is solely due to Player 1 having an accuracy of 0.09 percent points higher than that of Player 2 or if the black; and therefore starting, player has a huge advantage. Nevertheless, tests versus other machine Checkers players should include a side swap half-way through the duel just to be certain no player has side-advantage. As Table 6.2 use the best found Checkers player for both sides, it serve as a baseline for comparisons in this Chapter.

	Black	White	Draw
Players	Player 1	Player 2	
Results	70	13	17

Table 6.2: Results of 100 matches played between two Checkers players where the only difference is what side they play as, as well as the Tsetlin Machine predictor they use; Player 1 and Player 2 respectively.

Testing the Checkers player using Player 1 as its Tsetlin Machine predictor against an opponent selecting a random legal move proves the Checkers player has some skill in Checkers. This can be seen in Table 6.3, showing the results of 100 matches between these Checkers players. With the perfect score of 100 games won out of 100 games, this test is not suitable for investigating any increase in skill when tweaking the Checkers player.

	Black	White	Draw
Players	Player 1	Random Player	
Results	100	0	0

Table 6.3: Results of 100 matches played between the Checkers player using Player 1 as its Tsetlin Machine predictor and an opponent randomly selecting legal moves.

As mentioned in Chapter 4.4, Player 2 is a separate Tsetlin Machine from Player 1 in order to make sure the Tsetlin Machine predicting the outcomes for the white player's moves was trained for this specific purpose. In order to test whether this was an improvement over inverting the wins and losses output from Player 1 instead of using Player 2, 100 matches were played between the Checkers player using Player 1 as its Tsetlin Machine predictor against the Checkers player using Player 1 with inverted loss and win predictions as its Tsetlin Machine predictor. The results can be seen in Table 6.4. The white player did win almost as many times as seen in the baseline Figure 6.2, however; there is a significant difference in the amount of draws. Even though Player 1 has shown a higher accuracy, it performs worse against itself when inverting wins and losses than its counterpart which is trained for the perspective of the white player; Player 2.

	Black	White	Draw
Players	Player 1	Inverted Player 1	
Results	80	11	9

Table 6.4: Results of 100 matches played between the Checkers player using Player 1 as its Tsetlin Machine predictor and the Checkers player using Player 1 with loss and win predictions inverted as its Tsetlin Machine predictor.

As mentioned in Chapter 4.5.1, the Checkers player attempts to force moves that automatically leads to a win as well as avoid giving the opponent the opportunity to do the same by setting the large scores of 9 and -9. Table 6.5 show 100 games between the Checkers player using Player 1 as its Tsetlin Machine predictor and the Checkers player with a predictor claiming all moves to be winning moves. This table show whether the use of the 9 values is more important than a good Tsetlin Machine predictor. Although the white player managed to net a draw, this table show that the use of a predictor is important for reaching the skill of the proposed Checkers player.

	Black	White	Draw
Players	Player 1	Always-win-predictor	
Results	99	0	1

Table 6.5: Results of 100 matches played between the Checkers player using Player 1 as its Tsetlin Machine predictor and the Checkers player using a predictor saying every move leads to win, tree search with -9 and 9 score for guaranteed losses and wins was still in use.

Given the results found in Table 6.5, it would be interesting to see how well the Checkers player using Player 1 as its Tsetlin Machine predictor; giving moves guaranteeing wins and losses the scores of 1 and 0 instead of 9 and -9, would compare against the Checkers player using Player 2 as its Tsetlin Machine predictor. 100 games were played between these two Checkers players to investigate this, and the results can be seen in Table 6.6. Comparing these results to the baseline, Table 6.2, the effect of guaranteeing wins and losses can be seen. It can be seen that the effect of giving moves leading to guaranteed wins and losses the scores of 9 and -9 does not have any significant effect on the skill of the Checkers player. It is worth noting that even if 1 and 0 are set in place of 9 and -9, the way the scores themselves are set makes the scoring of 1 for guaranteed wins more effective than a typical win predicted by the Tsetlin Machine. Forced wins which are given the score of 9, or 1 in this scenario, are given in depth two of the tree search and moves guaranteeing wins have no child nodes. As explained in Chapter 4.5.1, all moves that are not leaf nodes are given the average score of their children. This means that the move in depth one leading to the board in depth two which guarantee a win, is given a score of 1 which is the highest score any move may have in this scenario. For a move to be considered equally as good as a move leading to a guaranteed win; all of its children must lead to boards the Tsetlin Machine predictor predicts to a win. Therefore the rule of giving the score of 1 to guaranteed wins still provide somewhat of a higher score than typically seen in the tree search of the Checkers player. As for the rule of giving 0 to guaranteed losses, in depth three; this provides no more weight in comparison to other scores as the Tsetlin Machine predictor performs predictions of moves' results in depth three. When giving these boards the score of 0 instead of -9, the Tsetlin Machine predictor is merely prevented from wrongly predicting the result as well as save time.

	Black	White	Draw
Players	Player 1 no 9's	Player 2	
Results	69	14	17

Table 6.6: Results of 100 matches played between the Checkers player using Player 1 as its Tsetlin Machine predictor, but guaranteed wins and losses are set to 1 and 0 instead of 9 and -9, and the Checkers player using Player 2 as its Tsetlin Machine predictor.

6.3 Comparisons of Checkers players

In order to measure how competent the Checkers player was in the game of Checkers, it was vital to have it play against other Checkers players. Therefore, the Checkers player played against the following opponents: strong Kingsrow, Weak Kingsrow, human opponent and 247 Checkers. The order the opponents is presented in is also the order of their presumed skill in Checkers. Some of the duels included a side-switch half-way through the match in order to avoid giving any player an unfair advantage. The Checkers player used the Tsetlin Machine predictors; Player 1 when playing as black and Player 2 when playing as white. A total of ten games were played for each duel as all games had to be played by hand, which was very time consuming, due to the Tsetlin Checkers player exclusively running on the operating system Linux, and CheckerBoard; which Kingsrow rely on, was written exclusively for the operating system Windows. Games against 247 Checkers were performed by hand through its website, ”www.247checkers.com” [38]. It is worth noting that these games were completed beyond games found in the dataset used by the Tsetlin Checkers player's Tsetlin Machine predictors; OCA 2.0, which consist of games deemed finished by human players without being fully finished in terms of Checkers rules as mentioned in Chapter 4.1. This means that the Tsetlin Machine predictor may not have been trained on a notable amount of boards that might arise by the end of a Checkers game.

As the Tsetlin Checkers player had functionality for communication with CheckerBoard through the ability to read manual print-outs from CheckerBoard, CheckerBoard was used as a means of playing against the Tsetlin Checkers player even when not playing against Kingsrow using CheckerBoard.

CheckerBoard feature various settings for tweaking the performance of Kingsrow. Strong Kingsrow and weak Kingsrow differ only in the way certain parameters were tweaked. Both strong Kingsrow and weak Kingsrow utilized the full 2 through 10 win-loss-draw dataset without any auxiliary databases [31]. The options tweaked between strong and weak Kingsrow were *hashtable*, ranging from 8 to 2048, *endgame DB*, ranging from 0 to 6912, and *level* which may range from instant to infinite. The two former settings was impossible to maximise due to limited hardware capabilities, and the level parameter was set to a time close to the typical time the Tsetlin Checkers player would use for picking a move when setting up smart Kingsrow. The Kingsrow version used was Kingsrow 1.19a, which use machine learning [8][27].

Table 6.7 show the results of 100 games between the Tsetlin Checkers player using Player 1 as its Tsetlin Machine predictor and Kingsrow using some pretty strong settings: *hashtable* 256, *endgame DB* 3136 and *level* 10 seconds. Kingsrow's level was set to 10 seconds as the Tsetlin Checkers player typically used 10 seconds at most when selecting a move. No side-switch was performed for this duel. The Tsetlin Checkers player has proven to be the most skillful when playing as the black player, as seen in Chapter 6.2, and should therefore have an advantage when only playing as the black side. Unfortunately, the strong Kingsrow won all ten games. This was expected as this was the opponent that was presumed to be the most skillful of the opponents to be challenged.

	Black	White	Draw
Players	Tsetlin	Kingsrow Strong	
Results	0	10	0

Table 6.7: Results of 10 games played between the Checkers player using Player 1 as its Tsetlin Machine predictor, and Kingsrow using the settings: *hashtable* 256, *endgame DB* 3136 and *level* 10 seconds. No side-switch.

Another ten games was played, but this time around; Kingsrow used weaker settings which should decrease its performance. The Tsetlin Checkers player used Player 1 as its Tsetlin Machine predictor, Kingsrow used the following settings: *hashtable* 8, *endgame DB* 64 and *level* 2 seconds. No side-switch was performed for these games either. The results can be seen in Table 6.8. Once again, Kingsrow won all ten games against the Tsetlin Checkers player. This means that, not only is Kingsrow better than the Tsetlin Checkers player, Kingsrow is in another league. It was presumed that the Tsetlin Checkers player would not have performed better in the place of the white player given the outcomes shown during self-play as detailed in Chapter 6.2.

	Black	White	Draw
Players	Tsetlin	Kingsrow Weak	
Results	0	10	0

Table 6.8: Results of 10 games played between the Checkers player using Player 1 as its Tsetlin Machine predictor, and Kingsrow using the settings: *hashtable* 8, *endgame DB* 64 and *level* 2 seconds. No side-switch.

Once it was discovered what the Tsetlin Checkers player cannot do; defeat Kingsrow, it was time to find out what it can do. Two possible opponents it could be able to beat was proposed: a human player and a Checkers player featured in an online Checkers game. The human player was me, the same human performing this research. I had barely played more than ten games of Checkers before this research and I would therefore call myself lacking in skill in this department. The online Checkers game chosen was 247 Checkers, using its hardest mode; expert mode [38]. Also, 247 Checkers had the option *force-jump* on as this rule was enforced both by CheckerBoard and by the Tsetlin Checkers player. In order to gauge the difference in skill between 247 Checkers and me, a duel of ten games with side-switch was played as seen in Table 6.9. As seen in this table, 247 Checkers proved to be much more skillful than me in Checkers.

	Human player	247 Checkers	Draw
Results	1	8	1

Table 6.9: Results of 10 games played between the human player and 247 Checkers using the settings: *difficulty* expert and *force-jump* on. Side-switch half-way through was performed

The Tsetlin Checkers player using Player 2 as its Tsetlin Machine predictor had a duel of ten games versus 247 Checkers on expert mode. The Tsetlin Checkers played at its weakest side, white, all ten matches as 247 Checkers did not feature a mechanism of transforming the board whenever a side-switch were to occur. As the Checkers board is not symmetrical, this made it thoroughly difficult to manually move and interpret what Checkers piece to move and where to place it when reading the output from the Tsetlin Checker player when it attempted to play as the black player. The results can be seen in Table 6.10. The Tsetlin Checkers player certainly proved to be in the same league as 247 Checkers, but was beaten by a significant amount of wins.

	Black	White	Draw
Players	247 Checkers	Tsetlin	
Results	6	3	1

Table 6.10: Results of 10 games played between 247 Checkers using the settings: *difficulty* expert and *force-jump* on, and the Checkers player using Player 2 as its Tsetlin Machine predictor. No side-switch

The final duel presented in this chapter is the Tsetlin Checkers player versus me. Side-switch was performed mid-way through the games, and the Tsetlin Checkers player used Player 1 and Player 2 as its Tsetlin Machine predictors for their intended sides. The results can be seen in Table 6.11. The Tsetlin Checkers player achieved marginally worse results against me than 247 Checkers, even though I had no chance against 247 Checkers.

	Tsetlin	Human player	Draw
Results	4	6	0

Table 6.11: Results of 10 games played between the Checkers player using its appropriate Tsetlin Machine predictor and the human player. Side-switch half-way through was performed

With the results of these games, it can be concluded that the proposed Checkers player is unable to compete with Kingsrow using its 2 through 10 dataset and performs worse than both a very new human Checkers player and the online Checkers game; 247 Checkers. The Checkers player has proven to play better than a player selecting random moves, as seen in Chapter 6.2. Therefore the skill level of the Checkers player is somewhere between a player selecting random legal moves and a very new Checkers player, or a comparable machine Checkers player.

Chapter 7

Alternative Machine Learning Algorithms

This chapter demonstrate and discuss the performance of the weighted Tsetlin Machine with positive boost using its best found hyper-parameters for the eight investigated dataset compositions, compared to the performance of various alternative machine learning algorithms.

Below, a list presents the various machine learning algorithms used for the comparisons of this chapter. Of these, Logistic Regression might be the most important to compare to the Tsetlin Machine as this machine learning algorithm is used by both Kingsrow and Cake; as explained in Chapters 3.2 and 3.3.

- **Log Reg** - Logistic Regression using *lbfgs* and *max_iter:* 1000, elsewise default parameters [39][40].
- **DTC** - Decision Tree Classifier using default parameters [39].
- **SGD** - Stochastic Gradient Descent for Classification using default parameters [39].
- **GNB** - Naive Bayes Gaussian using default parameters [39].
- **MNB** - Multinomial Naive Bayes using default parameters [39].
- **BNB** - Bernoulli Naive Bayes using default parameters [39].
- **TM** - weighted Tsetlin Machine with positive boost using the best found hyper-parameters: *clauses:* 19000, *treshold:* 40000, *s:* 9 and adaptive epochs with maximum epochs set to 500.

The list above also show the shortened names of each of the machine learning algorithms featured in Table 7.1, which show each of these machine learning algorithms' achieved accuracy for each of the eight most interesting dataset compositions. *NDC* refers to *NoDupeCheck* which means the dataset composition was created without removing any duplicate data. Most of the alternative machine learning algorithms use the default parameters, which may make this comparison a little unfair; so this should be taken note of when comparing these results.

	Log Reg	DTC	SGD	GNB	MNB	BNB	TM
StandardEnd	69.33%	54.69%	66.30%	56.34%	63.18%	62.95%	**69.79%**
EndSecondThird	**69.86%**	55.79%	67.48%	58.61%	64.78%	64.21%	69.41%
EndToEight	63.2%	50.52%	59.79%	55.8%	59.42%	57.83%	**63.38%**
StandardPureMetaData	**53.51%**	26.49%	51.46%	26.10%	45.81%	45.58%	48.39%
NDC StandardEnd	69.93%	57.55%	67.66%	59.44%	64.46%	64.02%	**71.44%**
NDC EndSecondThird	68.91%	57.71%	66.24%	59.75%	64.38%	65.72%	**70.65%**
NDC EndToEight	65.98%	57.05%	63.11%	59.08%	63.31%	62.13%	**67.73%**
NDC StandardPureMetaData	64.78%	**66.56%**	63.34%	17.11%	63.23%	62.73%	66.53%

Table 7.1: Achieved accuracies of the most interesting dataset compositions for various machine learning algorithms, including the weighted Tsetlin Machine with positive boost. *NDC* is short for NoDupeCheck

Studying the results shown in Figure 7.1, it can be seen that the variances in accuracies achieved by the machine learning algorithms for the dataset compositions generally vary in the same pattern the accuracy of the weighted Tsetlin Machine with positive boost do. But there are some interesting differences.

Every single machine learning algorithm is more accurate on EndSecondThird than of StandardEnd, except the Tsetlin Machine. Also, the accuracies achieved on the NoDupeCheck versions of the dataset compositions are generally higher than on their counterparts where duplicate data has been removed. However; *GNB* on StandardPureMetaData and both *Log Reg* and *MNB* on EndSecondThird achieved higher accuracies than on the NoDupeCheck versions of these dataset compositions. This may mean that for these machine learning algorithms, more data does not necessarily allow for a higher accuracy.

The Tsetlin Machine performed better overall, but it did not achieve the highest accuracy for every single dataset composition. On NoDupeCheck StandardPureMetaData, the Decision Tree Classifier achieved the highest accuracy, but it did generally achieve accuracies far below the accuracies achieved by the Tsetlin Machine. Logistic Regression achieved a better accuracy than any other machine learning algorithm presented for both EndSecondThird and StandardPureMetaData and typically achieved the second highest accuracy, behind the Tsetlin Machine, on most other dataset compositions. This do include NoDupeCheck StandardEnd, where Logistic Regression achieved an accuracy 1.51 percent points lower than the accuracy achieved by the Tsetlin Machine.

The weighted Tsetlin Machine with positive boost proved to beat all other presented machine learning algorithms on NoDupeCheck StandardEnd, but Logistic Regression proved itself as a strong machine learning algorithm which might be able to beat the Tsetlin Machine if it was to be tested with settings tweaked for this particular dataset composition.

Chapter 8

Conclusion and Future Work

This chapter contains the conclusion of the whole research. It also contains a short discussion of what could have been done differently in this research as well as suggestions on how the solution possibly could be further improved in the future.

8.1 Conclusion

The goal of this research was to research how well the Tsetlin Machine could be utilized in the domain of Checkers, and find out how it would compare to other Checkers players ranging from highly skilled such as Kingsrow; to a beginner-level human Checkers player. How to best configure the Tsetlin Machine and its Checkers player was also to be researched. Due to the lack of accessible data found, it was also important to research how to best utilize the limited data available.

Numerous hypotheses existed, which this research aimed to prove; or disprove. It was hypothesized that the weighted Convolutional Tsetlin Machine would do well in this domain, but the weighted Tsetlin Machine would perform better; and be able to correctly predict the results of Checkers boards most of the time. The Checkers player was hypothesized to be better at Checkers than a human non-professional player, but be nowhere near beating Kingsrow and other Checkers player at this level.

OCA 2.0 was chosen as the data source, as it was the largest collection of Checkers games found which it was possible to extract data from. Various dataset compositions off of OCA 2.0 was attempted, including some utilizing metadata and some which contained multiple boards generated from a single match in OCA 2.0; and variations of these which removed duplicate data. NoDupeCheck StandardEnd, which contained complete boards from OCA 2.0 without removal of duplicate data, proved to be the dataset composition which allowed most configurations of the Tsetlin Machine to achieve their highest accuracy out of the investigated dataset compositions.

Various configurations of the multiclass Tsetlin Machine was investigated; the Tsetlin Machine and the Convolutional Tsetlin Machine; both with and without weights and positive boost respectively. Both the use of weights and positive boost did show an increase in the accuracy achieved by the Tsetlin Machines, and the Convolutional Tsetlin Machine proved to perform worse than its non-convolutional counterpart. The weighted Tsetlin Machine with positive boost was shown to achieve the highest accuracy of the investigated configurations of the Tsetlin Machine, as was hypothesized; with the addition of positive boost. However; the Convolutional Tsetlin Machine was expected to perform better than its performance in this research. Also, I was not aware of the existence of positive boost when starting this research which is why this was not included in the hypothesis.

A function for automating the process of finding the appropriate amount of epochs to train for, and in turn improving the accuracy achieved during the training of the Tsetlin Machine, was proposed and tested. This function was called *adaptive epochs*, which made the Tsetlin Machine stop training when its average accuracy was declining for too many epochs. This also saves time when performing hyper-parameter testing as tests would not have to be redone with a more fitted amount of epochs.

The act of hyper-parameter testing was greatly simplified by the use of adaptive epochs, but it was still challenging and time-consuming to find well-fitted values for *clauses*, *treshold* and *s* for every single Tsetlin Machine configuration investigated. It was concluded that the best found hyper-parameters for the weighted Tsetlin Machine with positive boost, training on NoDupeCheck StandardEnd, was the following: *clauses:* 19000, *treshold:* 40000 and *s:* 9.

In order to save a trained Tsetlin Machine to be used in the Checkers player, it had to be decided which of the ten K-Folds of NoDupeCheck StandardEnd it was going to use. K-Fold number three proved to be the K-Fold of which the selected Tsetlin Machine configuration achieved the highest accuracy for. It was decided that separate Tsetlin Machine predictors was going to be used for each side; black and white. Both Tsetlin Machines trained on data from K-Fold number three of NoDupeCheck StandardEnd. However; the Tsetlin Machine trained for the white player had the wins and losses in its data inverted in order to make it train in the perspective of the white player. This proved to be a better solution than simply using the same Tsetlin Machine, with wins and losses flipped for its output when used for the white player. The Tsetlin Machine for the black player, Player 1, achieved an accuracy of 72.21% and the Tsetlin Machine for the white player, Player 2, achieved an accuracy of 72.12%. This accuracy is far above 50%, which means the Tsetlin predictors was able to predict outcomes of Checkers games correctly "most of the time".

The Checkers player utilized tree search with a depth of three, coupled with simple rules and the use of the Tsetlin Machine to play Checkers. The tree search with a depth of three proved to be the depth needed for the Checkers player to perform better than an opponent picking moves at random; even when reducing the effect of the built-in rules. This of which; proved that the Checkers player had some skill in Checkers, thanks to the Tsetlin Machine.

The Checkers player was tested against two configurations of Kingsrow, an online Checkers game by the name 247 Checkers and me; in the order of their assumed skill level. The Checkers player lost ten out of ten matches against Kingsrow; even when changing its settings for making it easier to beat. Kingsrow was also shown to have a much larger dataset than what OCA 2.0 had stored. The Checkers player narrowly lost against both 247 Checkers and me, a rather inexperienced Checkers player. It was hypothesized that the Checkers player would be proven to be a better Checkers player than a non-professional Checkers player, which proved to be false. With this, the Checkers player created by this research; has proven to have a skill level somewhere between a beginner-level human Checkers player and an opponent selecting legal moves at random.

While the proposed Checkers player was not able to perform on the same level as a highly skilled and fine-tuned Checkers player such as Kingsrow, it did remarkably well given the limited data its Tsetlin Machine predictors was trained on and its rather shallow tree search. Since the Tsetlin Machine performed better than Kingsrow's machine learning algorithm, Logistic Regression, using NoDupeCheck StandardEnd, the Tsetlin Machine may also perform around the same level as Logistic Regression, or possibly better, when using larger datasets also. Therefore; more data could be the solution for making the proposed Checkers player a contender to well-known Checkers players.

8.2 Future Work

There are numerous things that should have been performed differently; given more time, and things that could have been improved both for this research and the proposed solution. Below, each proposed change or improvement are listed in the order of assumed simplicity.

Further hyper-parameter optimization: Hyper-parameter optimization can be a lengthy, but rewarding process. This takes time both in terms of computing time, but labor time as well. Further hyper-parameter optimization should have been performed. Some full ten K-Fold train/test runs took anywhere between four hours to three days. This highly depended the amount of clauses used, the size of the dataset and the processing power available. For this research, a shared computer was used with the following specs: *CPU:* Intel Xeon E5520 2.27GHz, *RAM:* 32155 MB. This computer was available through the web-browser using JupyterLab [41]. The available processing power highly depended on the load from other users using this service as well as the amount of processes I ran at once. Additionally; the parallel library for the Tsetlin Machine had a memory leak which led to processes getting killed. This hindered hyper-parameter testing significantly. The memory leak was manually fixed near the end of the research process, but killed processes still occurred from time to time.

Test dataset without draw: No tests without draw was performed. It is unknown whether the removal or ignoring of draw could have increased the prediction accuracy of the Tsetlin Machine. It was assumed the removal of 60% of the data, which would occur when removing draws, would lead to worse performance.

Fix the draw rule misinterpretation: As explained in Chapter 6.1, the cases where a player has pieces left without any legal place to move them; was considered a loss to the player in question. This should have been a draw. Although this has proven to have little effect in actual games, it should be corrected.

Better data: The ending boards in the games contained in OCA 2.0 were not true end Checkers boards as moves could still be performed. These games were ended because two human players agreed on the result of the match; which neither the proposed Checkers player nor Kingsrow/CheckerBoard does. Therefore; games in this research would continue until one opponent could not move any more pieces. It could be safe to assume that the Tsetlin Machine predictor was not trained on boards similar to those which could be found at the end of games without human interruption.

Deeper tree search: A tree search of depth tree is not very deep, but processing time would increase drastically if the depth would be increased to five; which would be the next logical depth to aim for. But, by making the Checkers player faster; this should be feasible. A deeper search tree should in theory enable the Tsetlin Machine to be more accurate in its predictions as well as enabling the Checkers player to have more values to average which in turn could increase the skill of the Checkers player.

Study clauses: Each clause in a trained Tsetlin Machine contains Tsetlin automata, each of which holds a state. By interpreting these; it is possible to find out what rule that specific clause has learned. Doing this might uncover knowledge which could be used to understand and improve the Tsetlin Machine, or its configuration. As the proposed Tsetlin Machine predictor utilized weights to determine which clauses were the most important, these could be utilized to find the clauses the Tsetlin Machine itself deem the most important.

Optimization: By optimizing the Checkers player, it could be run significantly faster. There are many ways the implementation of the proposed solution could have been improved, for example writing it in C++ instead of Python. One notable change which is not as drastic however; is to have the Tsetlin Machine predictor evaluate all boards at once. During the research, it was noted that the speed of the Checkers player improved drastically by having the Tsetlin Machine predictor evaluate multiple boards at once, rather than one by one. Therefore; in the proposed solution, each board with the same parent node were sent to the Tsetlin Machine predictor together. This could be further exploited by having all boards to be evaluated in depth three, be evaluated at once.

Tsetlin Machine predictor with weighted scores: A prototype of a Checkers player using a modified evaluation function for the Tsetlin Machine predictor was created. This Checkers player tried to utilize the outputs of each state, and their weight, of the Tsetlin Machine in order to take into account the certainty of the Tsetlin Machine. The goal of this was to enable the Tsetlin Machine predictor to differentiate between losses, draws and wins respectively, which was theorized to increase the skill of the Checkers player drastically. A win prediction with low certainty was scored lower than a win prediction with high certainty, while still being scored higher than most draw predictions. A loss prediction with low certainty was scored higher than loss predictions with high certainties as a low certainty loss prediction would be more likely to be incorrect. This prototype had a significantly higher computing time than the proposed Checkers player, and the achieved skill was considerably lower. Whether this prototype was unsuccessful due to inadequate score-tuning or the Tsetlin Machine not being certain in predictions it should have been certain, is unknown. It is worth nothing though, with the little testing this prototype got to perform; it achieved a rather even score after about 90 games against the proposed solution using Player 2 as its Tsetlin Machine predictor.

Further improvement of adaptive epochs: As shown in Chapter 5.2, the adaptive epochs function was showing room for improvement. By investigating the achieved accuracy over the epochs for each K-Fold, it should be possible to improve the tuning of the function in order to increase its performance further.

The implementation of opening moves: In tournament play, games are typically played with a small amount of moves played by drawing from a book of opening moves which are then played two games on; having a side-switch in-between [10][28]. This is done as a measure for reducing the amount of games ending in draw, which in turn could make Checkers matches more exciting. This was not implemented in the proposed solution, but should be implemented in order to better conform with tournament rules.

Implement Monte Carlo tree search: The implementation of Monte Carlo tree search should in theory allow the Checkers player to achieve significantly better results in games as it allows for very deep searches [6]. This was attempted implemented, but was ultimately dropped early in the research due to difficulties in implementation, leading to a high cost of valuable research time. Using Monte Carlo tree search could also increase the difficulty in studying the Tsetlin Machine as this tree search is much more fluid than a constant tree search with a depth of three.

References

[1] Martin Fierz. Checkerboard. http://www.fierz.ch/checkerboard.php, 2008. Accessed: 17.5.2020.

[2] Ole-Christoffer Granmo. Tsetlin machine tutorial 4, 2019. Accessed: 5.4.2020.

[3] Ole-Christoffer Granmo. The Tsetlin Machine - A Game Theoretic Bandit Driven Approach to Optimal Pattern Recognition with Propositional Logic. *arXiv preprint arXiv:1804.01508*, 2018.

[4] Todd Wittman. Lecture 18: Tree traversals. https://web.archive.org/web/20150213195803/http://www.math.ucla.edu/~wittman/10b.1.10w/Lectures/Lec18.pdf, unknown. Accessed: 18.5.2020.

[5] University of Alberta. Lecture 8, tree traversal. http://webdocs.cs.ualberta.ca/~holte/T26/tree-traversal.html, unknown. Accessed: 18.5.2020.

[6] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. 01 2008.

[7] Wikipedia contributors. Draughts. https://en.wikipedia.org/wiki/Draughts, 2020. Accessed: 7.5.2020.

[8] Bob Newell. Machine learning comes to kingsrow. http://www.bobnewell.net/nucleus/checkers.php?itemid=1177, 2018. Accessed: 31.3.2020.

[9] ALEXIS C. MADRIGAL. How checkers was solved. https://www.theatlantic.com/technology/archive/2017/07/marion-tinsley-checkers/534111/, 2017. Accessed: 7.5.2020.

[10] Wikipedia contributors. English draughts. https://en.wikipedia.org/wiki/English_draughts, 2020. Accessed: 7.5.2020.

[11] Y. Bjornsson A. Kishimoto M. Muller R. Lake P. Lu S. Sutphen J. Schaeffer, N. Burch. Checkers is solved. https://science.sciencemag.org/content/sci/317/5844/1518.full.pdf, 2007. Accessed: 1.4.2020.

[12] Centre for Artificial Intelligence Research (CAIR). Repositories. https://github.com/cair, 2020. Accessed: 9.5.2020.

[13] Ole-Christoffer Granmo. pytsetlinmachineparallel. https://github.com/cair/pyTsetlinMachineParallel, 2020. Accessed: 9.5.2020.

[14] K Darshana Abeyrathna, Ole-Christoffer Granmo, and Morten Goodwin. Extending the tsetlin machine with integer-weighted clauses for increased interpretability. *arXiv preprint arXiv:2005.05131*, 2020.

[15] Ole-Christoffer Granmo, Sondre Glimsdal, Lei Jiao, Morten Goodwin, Christian W. Omlin, and Geir Thore Berge. The Convolutional Tsetlin Machine. *arXiv preprint arXiv:1905.09688*, 2019.

[16] Sanjay. M. Why and how to cross validate a model? https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f, 2018. Accessed: 6.4.2020.

[17] Wikipedia contributors. Precision and recall. https://en.wikipedia.org/wiki/Precision_and_recall, 2020. Accessed: 17.5.2020.

[18] Wikipedia contributors. F1 score. https://en.wikipedia.org/wiki/F1_score, 2020. Accessed: 16.5.2020.

[19] Jonathan Schaeffer. *One Jump Ahead: Computer Perfection at Checkers.* Springer Science Business Media, 2008.

[20] Wikipedia Contributors. Marion tinsley. https://en.wikipedia.org/wiki/Marion_Tinsley, 2019. Accessed: 11.5.2020.

[21] World Checkers Draughts Federation. Historical champions. http://www.wcdf.net/champions_hist.htm, unknown. Accessed: 25.5.2020.

[22] M.J.H. Herule and L.J.M. Rothkrantz. Solving games Dependence of applicable solving procedures. *Department of Software Technology, Department of Mediamatica, Faculty of Electrical Engineering, Mathematics and Computer Sciences, Delft. University of Technology*, 2007.

[23] Louis Victor Allis. Searching for Solutions in Games and Artifcial Intelligence. *CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG*, 1994.

[24] World Checkers Draughts Federation. World checkers draughts federation statutes bylaws. http://www.wcdf.net/statutes.htm, 2014. Accessed: 11.5.2020.

[25] University of Alberta. Endgame databases. https://webdocs.cs.ualberta.ca/~chinook/databases/, unknown publish date. Accessed: 31.3.2020.

[26] N. Burch R. Lake P. Lu S. Sutphen J. Schaeffer, Y. Bjornsson. Building the checkers 10-piece endgame databases. https://link.springer.com/content/pdf/10.1007/978-0-387-35706-5_13.pdf, 2003. Accessed: 1.4.2020.

[27] Ed Gilbert. Kingsrow. http://edgilbert.org/Checkers/KingsRow.htm, 2019. Accessed: 31.3.2020.

[28] Martin Fierz. Making of - cake 1.87. http://www.fierz.ch/cake186.php, 2019. Accessed: 11.5.2020.

[29] Martin Fierz. Nemesis wins the computer checkers world championship in las vegas. http://www.fierz.ch/vegas.htm, 2009. Accessed: 11.5.2020.

[30] Wikipedia contributors. Checkerboard download. https://en.wikipedia.org/wiki/Nemesis_(draughts_player), 2016. Accessed: 25.5.2020.

[31] Ed Gilbert. Kingsrow for english/american checkers. http://edgilbert.org/EnglishCheckers/KingsRowEnglish.htm, 2020. Accessed: 11.5.2020.

[32] Martin Fierz. Checkerboard download. http://www.fierz.ch/download.html, 2005. Accessed: 25.5.2020.

[33] Wyllie Online Draughts Club. A database dilema solved. http://www.wylliedraughts.com/Platinum.htm, unknown. Accessed: 25.5.2020.

[34] Martin Fierz. Murray cash and nemesis? http://www.usacheckers.com/forum/viewtopic.php?t=1292, 2007. Accessed: 25.5.2020.

[35] Wiki contributors. Martin fierz. https://www.chessprogramming.org/Martin_Fierz, 2019. Accessed: 17.5.2020.

[36] Martin Fierz. 3rd party add-ons. http://www.fierz.ch/download.php, 2009. Accessed: 31.3.2020.

[37] SamRagusa. Checkers-reinforcement-learning. https://github.com/SamRagusa/Checkers-Reinforcement-Learning, 2017. Accessed: 19.4.2020.

[38] 24/7 Games LLC. 247 checkers. https://www.247checkers.com/, unknown. Accessed: 4.5.2020.

[39] scikit-learn developers. Martin fierz. sklearn.linear_model.LogisticRegression, 2019. Accessed: 23.5.2020.

[40] Jorge Nocedal Jose Luis Morales. Ciyou Zhu, Richard Byrd. L-bfgs-b. http://users.iems.northwestern.edu/~nocedal/lbfgsb.html, 2011. Accessed: 23.5.2020.

[41] Project Jupyter. Jupyterlab documentation. https://jupyterlab.readthedocs.io/en/stable/, 2018. Accessed: 26.5.2020.