UiA University of Agder

Playing endgame chess with the Tsetlin Machine

Sander J. Otterlei
Joel A. H. Reiersøl

SUPERVISORS
Dr. Ole-Christoffer Granmo

Master's thesis
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 report is about training an Artificial Intelligence(AI) that is able to play out endgames, using the existing solved endgame of chess to train the Tsetlin Machine on.

This report describes the methods used to train and test a Tsetlin Machine using both the convolutional and multiclass implementation. We have further tested out different methods to handle the data it trains on to investigate what methods work best. Where these methods are; to split the data for two machines for either white or black starting player, transforming the data to only be from one starting players perspective and one splitting based on results by first having one machine looking at win versus draw and loss, then a second machine for looking at draw versus loss.

The results showed that some of the methods used, involving only looking at one players perspective, worked well for predicting if the board would lead to a win with perfect play. Since several om the methods achieved over 90% accuracy in the testing, while the best achieves an accuracy of 95%. However the playing off the endgame was lacking, as the games mostly ended in draws even when the Tsetlin Machine should have been able to win. Such as only drawing against each other, and only drawing against Monte Carlo Tree Search.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In machine learning, or more specifically Neural Networks, there is a problem of interpretability. Which is that the Neural Networks can achieve better results than humans, but that the reasoning of the choice is not known or not very clear. That the Neural Network manages to make optimal or better choices, but it is not completely understood why. There is work being done in order to try to achieve understanding, or be able to understand why a given choice is made. Where Olah, et al.[25] is one of the most recent when it comes to understanding Neural Networks and why they make a certain choice.

Why is interpretability so important though? As written by Olah, et al.[25], it gives a better understanding of why a set of choices were made, and can help improve the choices made by the machine learning agent. Such as qualitative comparisons of the solutions learnt in order to achieve better designs and results[10, 19]. It can also be used in order to make better analysis and comparisons of different Neural Network models.

The Tsetlin Machine is a machine learning algorithm which learns patterns, and uses the patterns with propositional logic in order to decide its outcome. Which is similar to a Neural Network. In the fact that they both consist of some medium which stores information about a state, and makes their choice based on that state. Where Neural Networks stores it as neurons that activate when certain conditions are met. While the Tsetlin Machine learns several patterns.

The biggest difference is that one would have to investigate every neuron and connection in order to get a full overview of how the Neural Network decides. While the Tsetlin Machine is designed such that the patterns can be visualized, and used in order to understand what the machine learning agent is looking for. In order to make its choice.
Because the Tsetlin Machine is relatively new there is research required in order to measure its capabilities, limitations and how interpreteable the patterns become.

Chess has been chosen for this project since it is a fairly complex game. Where this complexity arises from approximately 50 moves being available on average every turn, and that there are different types of pieces on the board. Where the pieces have values depending on their type[16]. This value comes from how the piece can capture other pieces and move on the board.
Chess is also a game where one has tried to create Artificial Intelligence(AI) that can play the game on the same level or better than humans. The biggest breakthrough came in 1996 with Deep Blues victory against then reigning world champion Garry Kasparov[3]. More recent breakthroughs in chess AI is Alphazero[6], a neural network based AI developed by Deepmind, which masters chess, Go and shogi[27]. Where Go has, in the recent years, been considered the more complex game for AI to play and solve.

This report details the work and tests done to train and evaluate a Tsetlin Machine that can play chess. More specifically how to play endgame chess. Since endgame chess has perfect solutions, meaning that the optimal moves for a given position are known, making it easier to evaluate the Tsetlin Machine.

1.1 Thesis definition

This section defines the goals we had in mind when we set out to test the machine.

1.1.1 Thesis Goals

Goal 1: Train a Tsetlin Machine too play a chess endgame through a tree-search. By seeing if the moves would result in win, loss or a draw, and analyze if it can play or not.

Goal 2: Test different types of Tsetlin Machines; Multiclass, Convolutional and weighted, and compare the results.

Goal 3: Test different methods for splitting the data and setting up the Tsetlin Machine; one class against the others, change the player and split by player.

Goal 4: Visualisation of clauses.

Goal 5: Testing other machine learning algorithms and comparing the Tsetlin Machine to them.

1.1.2 Limitations

Limitation 1: Because of the size of 7-pieces and 6-pieces, only 5-pieces or fewer boards have been used. This refers to the amount of pieces in which a chess game has a known solution. This is considered a limitation since the dataset is reduced, and one also has to play boards with few pieces since the kings are 2 of the 5 pieces.

Limitation 2: This solution is not for playing a full game of chess. The dataset and the machine learning algorithm will therefore be for endgames, in which there is an already known solution.

1.1.3 Summary

The main goal for this report is to train a Tsetlin Machine that can play a game of chess, given that there are 5 or less pieces left on the board. The solution should also be able to visualize the clauses generated by the Tsetlin Machine.

1.2 Data set

In order to train the Tsetlin Machine a data set was generated using datasets from Lichess[21]. The dataset consists of endgames for chess. Endgame means that the solution for the board is already known, and one can therefore play perfectly if one knows the solution. Perfect play has currently only been solved for 7 or fewer pieces, though our dataset is limited to 5 or fewer pieces, since there were space limitations for our equipment.

1.3 Thesis outline

This report will be divided into several chapters, these chapters are as follows:

Chapter: 2 Background
This chapter describes the underlying technology used, being the Tsetlin Automata, Tsetlin Machine and K-fold cross validation and more.

Chapter : 3 State of the art
This chapter describes the already existing best chess AI's and some history surrounding AI's made for chess.

Chapter: 4 Proposed Solutions
This chapter describes our solution for the problem. Meaning what we implemented, and how this implementation was done.

Chapter: 5 Results and discussion
This chapter describes how we tested the Tsetlin Machines we trained, and discuss the results and what they mean.

Chapter: 6 Conclusion
This chapter is used to conclude the works. Looking at what went well with the project, what did not go so well and also what was not finished or other tasks that would be useful to complete in the future, to optimize the solution to the problem we tried to solve.

Chapter 2

Background

2.1 Tsetlin Automata

The Tsetlin Automaton is a simple state machine structure, it does one of two actions based on what state it is in [14]. Simply put, if the Tsetlin Automaton has 2N states. It will do action one if it is in a state less than N, and action 2 if it is in a state greater than N.
The Tsetlin Automaton gets a reward or a penalty based on the result of the action. If it gets a penalty it will move towards the middle, or it will go towards a state of N. If it gets a reward it will go towards the edges. This makes it so that for each time it does the correct action it gets more certain that this action is correct. For every wrong action it will get less certain, and swap what action to do if it crosses the middle. The Tsetlin Automaton is very simple computational wise and it also has a low memory footprint, as it just needs to maintain an integer.

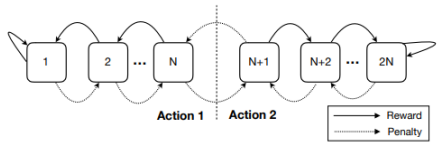

Figure 2.1: Tsetlin Automaton illustration, reprinted from Granmo, et al.[14].

Figure 2.1 shows a Tsetlin Automaton in a two action environment. If it is

on the left side being N or less it would do action 1, and if it is on the right side being N+1 or more it would do action 2. Given the result of the action it would follow the arrows shown in the figure accordingly.

2.2 Tsetlin Machine

The Tsetlin Machine is constructed using a set of clauses for each class, or classification, in the data. These clauses are patterns or sub patterns used for deciding the output [14]. The clauses for the class consist of negated and non-negated clauses. Meaning that half the clauses votes for the input being of the class, and the rest votes against. Where the first half of the clauses are non-negated, or vote for, while the other half is negated, vote against. When the clauses have voted, they are tallied up, and the classification with the highest score is the final output of the Tsetlin Machine.

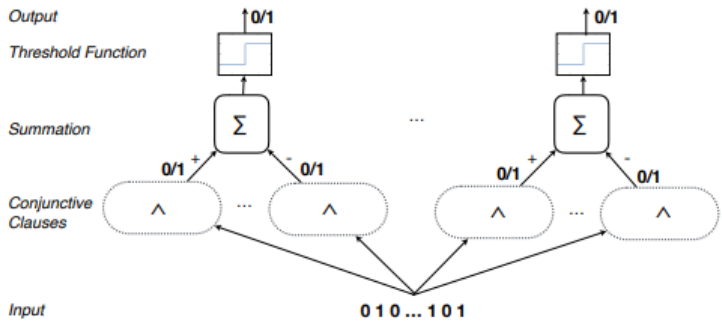

Figure 2.2: Tsetlin Machine illustration, reprinted from Granmo, et al.[14]

Figure 2.2 shows how the Tsetlin Machine is put together, with having an added threshold function that arbitrates the final output.

2.2.1 Clause

A clause is one of the patterns created and used by the Tsetlin Machine in order to do classification. A clause consists of several Tsetlin Automata, which says what bits to include in the given input. The amount of Tsetlin Automatas is decided by the features given when training the Tsetlin Machine. Where 2 Tsetlin Automatas are used to represent one feature. A

feature also means a bit, meaning that two Tsetlin Automatas are used to include or exclude a bit. Include means that the given feature has to be present for the given clause to vote, while exclude means the given feature cannot be present for the clause to vote. If both the include and exclude bit are set it means that the clause has lead to false positives, and in an effort to combat this the Tsetlin Machine has set both bits to invalidate the clause. If none of the bits are set, it means the given feature is not looked at[14].

2.2.2 Multiclass Tsetlin Machine

The Multiclass Tsetlin Machine is an implementation of the Tsetlin Machine which allows for classification of several classes. Such that one can classify more types of data rather than only one or two.[14]

2.2.3 Convolutional Tsetlin Machine

The Convolutional Tsetlin Machine is another implementation of the Tsetlin Machine, in which it learns smaller patterns. These patterns are used to look at smaller portions at a time. If a specific pattern is found, it would be able to only look at that pattern and not the entirety of the input. This pattern would also receive possible positions where it looks for the pattern[15]. The pattern is done by arranging the input as a window, or list of lists. In which the Tsetlin Machine will also learn the position for where the pattern should be found.

2.2.4 Weighted Tsetlin Machine

Weighted Tsetlin Machine is an addition to the Tsetlin Machine in which the clauses are weighted. Meaning that often appearing clauses will have a greater weight. A clause will be assigned a weight when created and the weight will increase or decrease depending on how often the clause is seen. Creating something similar to the weights of a Neural Network, in which a more important clause will have a greater impact on the decision[1]. The weighted Tsetlin Machine should be able to reduce the amount of clauses

needed to achieve the same accuracy as a regular Tsetlin Machine, and also reduce the training and evaluation time.

2.2.5 Parallel Tsetlin Machine

Parallel Tsetlin Machine is an implementation of the Tsetlin Machine library, which turns the training and predicting into a multi-threaded job in order to improve the speed of the algorithm.

2.2.6 Hyper-parameters

Hyper-parameters is used for optimizing the learning and accuracy of the Machine Learning algorithm. These hyper-parameters can affect the learning rate and how well it learns, and is useful for mitigating or avoiding overfitting. Tsetlin Machine has clauses, threshold and s as hyper-parameters.

Clauses are the number of clauses, and are the patterns that the Tsetlin Machine learns. If there is a high enough number, as many as there are possible combinations, the Tsetlin Machine will overfit and learn all the patterns.

Threshold is used with the threshold function in order to arbitrate the final output of the Tsetlin Machine. This is used together, with the summation of the clauses, to replace the or operator that was in the basic model. Which is done in order to reduce the impact of noise. The threshold function decides the output based on the result of the conjunctive clauses. It looks at the result of the Clauses that voted for and the clauses that voted against.

S is a parameter that deals with reducing the impact of false positives and false negatives, and reinforcing true positives. Such that a larger s leads to clauses with more literals. Where literals are also defined as the bits which decide what feature to include or exclude.[14]

2.3 Chess

Chess is a turn based board game with two players, where the player with the white pieces start. Each player has 16 pieces, and there are 8 pawns, 2 knights, 2 horses, 2 rooks, 1 queen and 1 king. The classification of the piece refers to how the piece can move. On a board the horizontal line of squares is called a rank, the vertical file of squares is called a lane and the diagonally collection of same colored squares is called the diagonal. The goal of the game is to put the other players king in such a state that they can't prevent themselves from being captured. In chess one is not allowed to put one self in check. Check is a state in which ones king is under direct threat of being captured. If a player is not able to do any legal move the game ends in a draw. Chess is a perfect information game, since all the knowledge of the game is available to both players.

2.3.1 Chess rules

In chess the rules are simple. The goal is to capture the opposing sides king. The pieces on the board have different movement abilities. Most of the pieces can not pass through pieces. Pieces are captured by moving the pieces onto squares where the opponent has a piece, when captured a piece is taken out of the game.

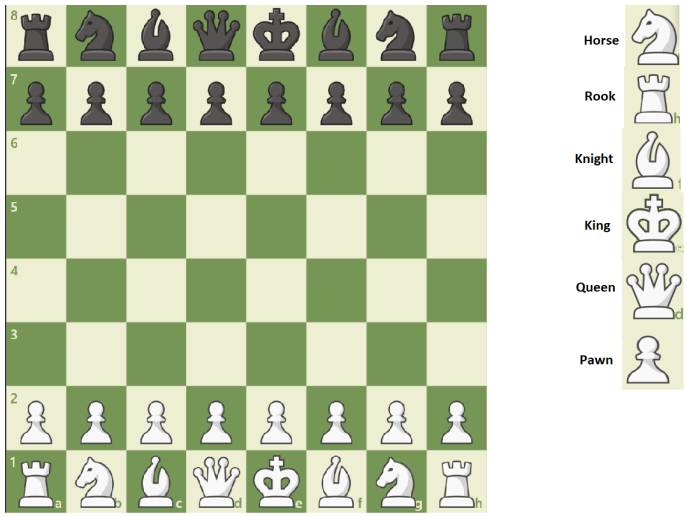

Figure 2.3: Start position of a Chess board, figure made using chess.com[5]

Normal moves

The King(k) can move one square in any direction.[23]

The Queen(q) can move along the rank, along the file and diagonally for as many squares as wanted.

The Rook(r) can move along the rank and along the file for as many squares as wanted

The Horse(h) can move to the squares that are two steps along the rank and one step along the file, and also too the squares that are two steps along the file and then one step along the rank. This piece can move through pieces and only interacts with the final square it lands on.

The Knight(n) can move diagonally for as many squares as wanted.

The Pawn(p) can move 1 step along the file towards the opponents side of the board, and has the option to step 2 squares on its first move. The pawn can not capture a piece by moving along the file. The way for a pawn to capture a piece is when there is an enemy piece diagonally 1 square in

front of the pawn, then it can capture it by moving to that square.

Special moves

Other than normal moves the pieces can do, there are some special moves that are conditional. Some rules exist to speed up and prevent endlessly stalling games. These rules are often optional rules outside the game, like having a clock to prevent a player using more than a certain amount of time.

En passant[18] is a pawn capture move that is only available after a pawn has used the 2 step move. When a pawn uses this 2 step move, an enemy pawn will be able to capture the pawn if it could capture it if it took 1 step. If the pawn captures the other this way it would end up on the square behind the pawn that did the 2 step move.

Castling is a chess move in which 2 pieces, rook and king, are moved. This move can only be done if the king and the rook involved hasn't been moved. You can castle on both of your rooks. In addition to not having moved them, the squares between the king and the rook cannot be occupied, and the king cannot be in check. If the move is done the king moves two squares towards the rook, and the rook moves to the opposite side of the king.

Promotion is a rule in chess where when a pawn reaches the end of the board on the opponents side, it will be able to upgrade into a Knight, Rook, Horse or Queen. The upgraded piece will take the pawns place in the same move as the pawn reaches the square. The promotion does not rely on pieces that are captured, so a player can for example have more than one Queen on the board at the same time.

2.3.2 Forsyth-Edwards Notation (FEN)

FEN is a single line of symbols used to represent the state of the chess game[9]. In this project FEN has been used a lot as it is an easy way to handle looking at board states, and manipulating them by executing moves using libraries that support FEN.

```
8/8/1Kp5/3b4/4R3/5k2/8/8 b - - 0 49
```

Figure 2.4: Example of a FEN

The first part of the FEN defines the pieces on the board[24], where numbers represent the number of empty spaces. The slashes represents the rows, and the pieces are represented with letters uppercase for white pieces and lowercase for black pieces.

The part after the information about the pieces is who is next to do a move. As seen in the example FEN from above 2.4 the black pieces has the next move.

The first of the two "-" in the example tells if the player is able to castle, it is "-" if the player cannot castle, but if it can castle it will be notated with "q" for queen side or "k" if it can castle on kings side. The next "-" in the example is about en passant target squares, this being the square behind a pawn after it does a two step move. Neither of these are present much in the endgame of chess.

The second last element of the FEN is a halfmove clock, this number defines how many half moves have been done since the last pawn move, or capture. This is used to determine the 50 move rule which is a rule that states; if no pawn has moved or no capture has been done in the last 50 moves a player can claim that it is a draw.

The last element of the FEN is the fullmove number records how many full turns have been done. It starts at 1 and increments every time black does a move.

2.3.3 Syzygy tablebases

Syzygy tablebases is a tablebase containing pre-calculated information about endgames in chess. Currently it has up to 7 pieces available. In 2003 6 piece tables was released by Ronald de Man[22]. He also released the code for generating and probing the tablebase. In 2018 Bojun Guo spent several months of computation time on his supercomputer to generate the 7 piece tablebase[22].

Two different types of information are made. One is WDL(Win/Loss/Draw) which contains the information of if the current player to move wins, loses or draws the game[13]. But in this there is no good way to handle the 50 move rule. Therefore the second set of information DTZ(depth to zero) describes how many moves before a capture or a pawn move, where the winner tries to minimise the DTZ and the loser tries to maximise it.

Pieces Amount	WDL	DTZ	Total
3-5	378.1 MiB	560.9 Mib	939.0 MiB
6	67.8 GiB	81.4 GiB	149.2 GiB
7	8.5 TiB	8.3 TiB	16.7 TiB

Table 2.1: Size of the Syzygy tablebases[22]

Above, in table 2.1, the sizes of the different tablebases are shown. For this project only the WDL would be needed to get the desired effect. Still with only needing this, the nearly 70GB of needed space to include 6 piece endgames made it not get included in this project, because of the limited space when the code had to be run on a virtual machine.

2.3.4 Python-Chess library

The Python-Chess library is a python library that contains multiple useful functions that supports looking into chess[12]. Some of these functions are used in this project when looking at the chess boards.

Two functions that has been used in this project is probing Syzygy tablebase[13], and handling the PGN(portable game notation) files[11]. Even though these are the main things the library was used for, it also proved useful for working with FEN's for finding the next legal moves and other board interactions like executing moves.

2.4 Monte Carlo Tree Search

A method that finds optimal decisions by the use of tree search and random sampling[4, 2]. Where it simulates a multitude of random games in order to find a good solution or strategy. It works by trying to find a leaf node. Where it either follows a known path that leads to good results, or explores unknown paths in order to possibly find better solutions. When it reaches one of the current leaf nodes of the tree, or states that does not exist, it will add one or more of the new states found. This way the tree is expanded incrementally. The initial state that it works from is usually only one node.

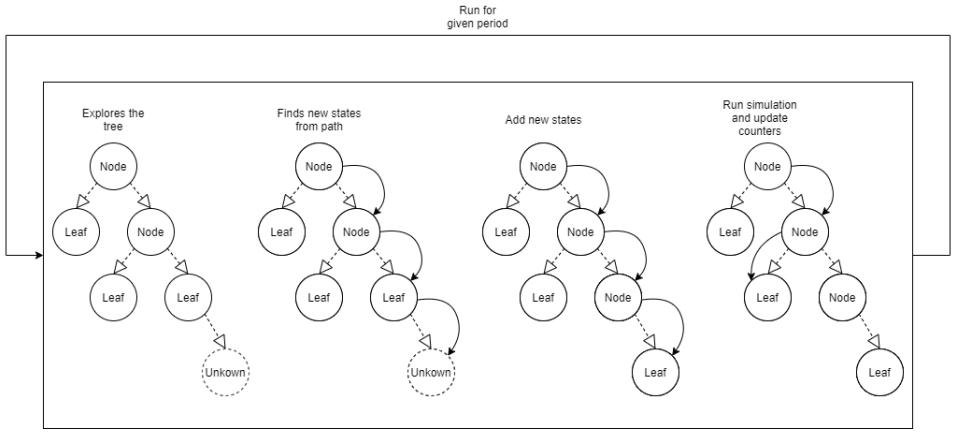

Figure 2.5: Adapted figure 1 from Chaslot, et al.[4]

The method will then run simulations, where it will choose the moves randomly. When it reaches the end of the game it will update the nodes visited, where a counter, for the given nodes, will increase. The final solution is the most visited tree nodes.

The implementation from mcts[1] has been used in order to implement the Monte Carlo Tree Search. In which one only has to define evaluation, termination and functions for progressing the game. From there one only has to give an initial state and how long the method is to run.

[1] https://github.com/pbsinclair42/MCTS

2.5 Precision

Precision is qualitative, it returns the amount of correct predictions for a class and divides it by the amount of correct predictions and incorrect predictions for a given class. Showing how good the classifier is at predicting correctly.[7].

$$\frac{TP}{TP + FP}$$

Figure 2.6: Formula for calculating precision

Figure 2.6 shows the formula for calculating precision. TP is the amount of True positives, which is the amount of correctly identified positives. Where positives is decided by the person. An example would be that positives is wins. Therefore true positive is the amount of correctly predicted wins. FP is false positive, which would be the amount of incorrectly predicted wins.

2.6 Recall

Recall is quantitative, where it takes the amount of correct predictions, and divides it by the amount of correct predictions and the amount of predictions that should have been the given class[7]. Showing how good the classifier is at finding the correct predictions.

$$\frac{TP}{TP + FN}$$

Figure 2.7: Formula for calculating recall

Figure 2.7 shows the formula for calculating recall. FN is false negative, which would be the amount of incorrectly predicted losses and/or draws.

Chapter 3

State Of The Art

For a long time AI has been used in order to try and solve or play chess as optimally as possible. This is hard because the complexity of chess makes it difficult to completely solve the game from start to finish, with the tools available.

Deep blue

Deep blue is not currently the top of the available chess AI's, but it was the one that made the first huge breakthrough by winning against the then reigning world champion in chess Garry Kasparov[3]. The AI is made by a team from IBM who worked with input from several chess experts to program its value based approach[17]. It is in its core an AI that calculates values of the chess board from many different factors. The AI's evaluation was based on around 8000 factors for comparing the boards against each other. These factors were both simple ones as to which pieces were left on the board, and more nuanced ones that looked into the positions and interactions between the pieces. Although Deep blue is currently not considered the best chess AI, it made huge progress for it's time into mastering the game of chess with computers.

Alphazero

Alphazero is an AI developed by Deepmind that masters the game of chess, shogi and go, using self training methods.[6] It is using Reinforcement learning to learn through playing against itself, without information other than

the rules of the game. Alphazero uses a general purpose Monte Carlo tree search algorithm, to search for moves to make while playing the game. Deepminds Alphazero AI is able to achieve superhuman level off play in chess within a day, proven by beating world class chess programs like stockfish and elmo.

Stockfish

Stockfish is a world class open source chess engine, able to beat top class chess players. Unlike Alphazero's self learning approach the Stockfish program focuses more on rules and calculations of predetermined values[26].

Stockfish works by searching for moves, and calculates the value of the positions based on predefined values. The Alpha-beta pruning is used to limit the amount of calculations needed to be done, as a full tree search in chess gets very big in just a few steps. The Stockfish algorithm does this type of search as it goes for being a real time chess engine, getting the best possible solution within a reasonable time frame.

Open spiel

Open spiel is an open source framework for reinforcement learning in games made by Deepmind, the code can be found on Github[8]. It is able to work for many different games, chess being one of the supported games. It is a collection of environments and algorithms for doing reinforcement learning to learn how to play the games[20]. Open spiel is not state of the art in the same way as the other algorithms mentioned. But it is state of the art in the way it enables the different environments and algorithms to work together using its framework.

Differences and similarities to this projects proposed solution

Unlike the AI's mentioned in this chapter this projects Tsetlin machine does not attempt to play an entire game off chess, it just focuses on the solved endgame. The solution will not be able to compete against the state of the art AI's in playing the game, but it can bring with it an AI that can be interpreted by humans, by looking at the Tsetlin Machine clauses. These can be examined to find patterns in the way it plays that can help understand the game.

Chapter 4

Proposed Solutions

4.1 Overview

This chapter details the implementations created or used in order to solve the tasks. Figure 4.1 shows general execution of the training and testing. More detailed explanations are found in the remainder of the chapter.

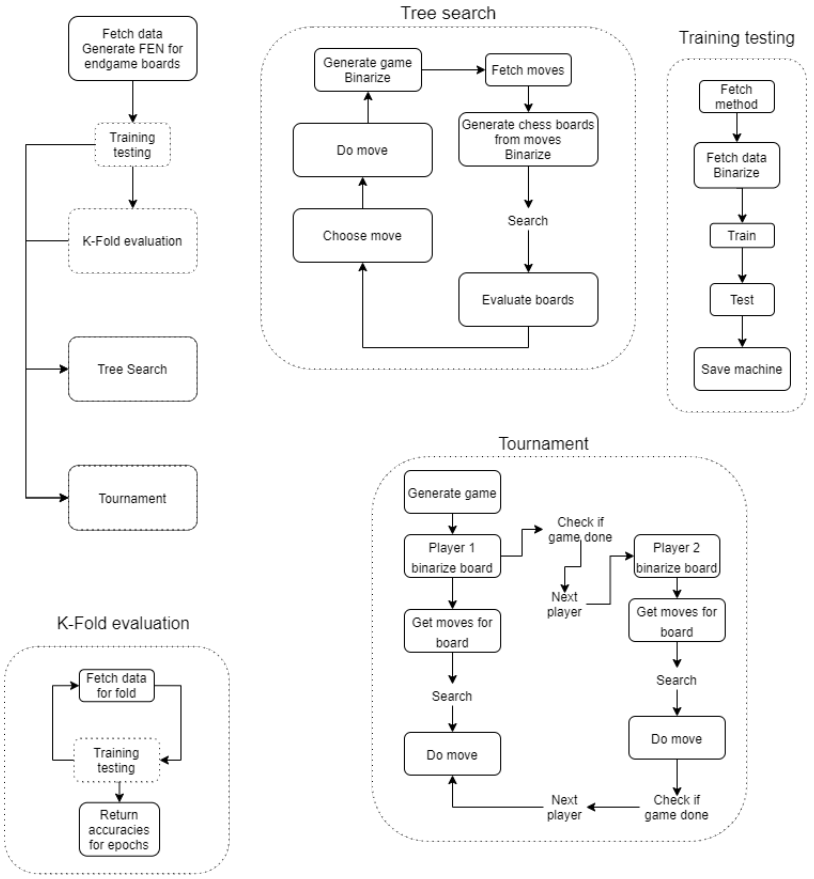

Figure 4.1: General overview of execution

4.2 Data

For the data the Lichess database[21] was used. Where this data is recorded games played on the platform. This data is stored as a pgn(portable game notation) file. The Python chess library[12] includes ways to handle such files and is used to iterate through it. The data is separated into years so three separate years worth of games for the different piece datasets was used

for the data generation. For determining the outcome of endgames we use Syzygy tablebases[22].

4.2.1 Data Generation

The data generation is done by iterating the files filled with millions of games gotten from the Lichess database. Like shown in figure 4.2 the code iterates through the games of the file it currently is examining. For each of the games it iterates the moves done on the board of the game. This is done until either the wanted amount of pieces 3,4 or 5 is left on the board, or if the game ends before the amount of pieces left is reached, where the code will then go onto the next game. If the correct amount of pieces is reached the code stores the FEN of the board into an array, and then goes to the next game if the amount of FEN's wanted is not reached yet. The code goes to the stage of storing the FEN's to a file when it either reaches 50 000 FEN's or it runs out of games in the file. When this happens the array with the FEN's gets iterated, and for each FEN it checks, using the Python chess library, if it is a legal move. Which is done before using the library to probe the Syzygy tablebase, to get the result of the game. After this probe both the FEN and the result of the game gotten from the probe is written into a csv file that is used when training and testing the Tsetlin Machine.

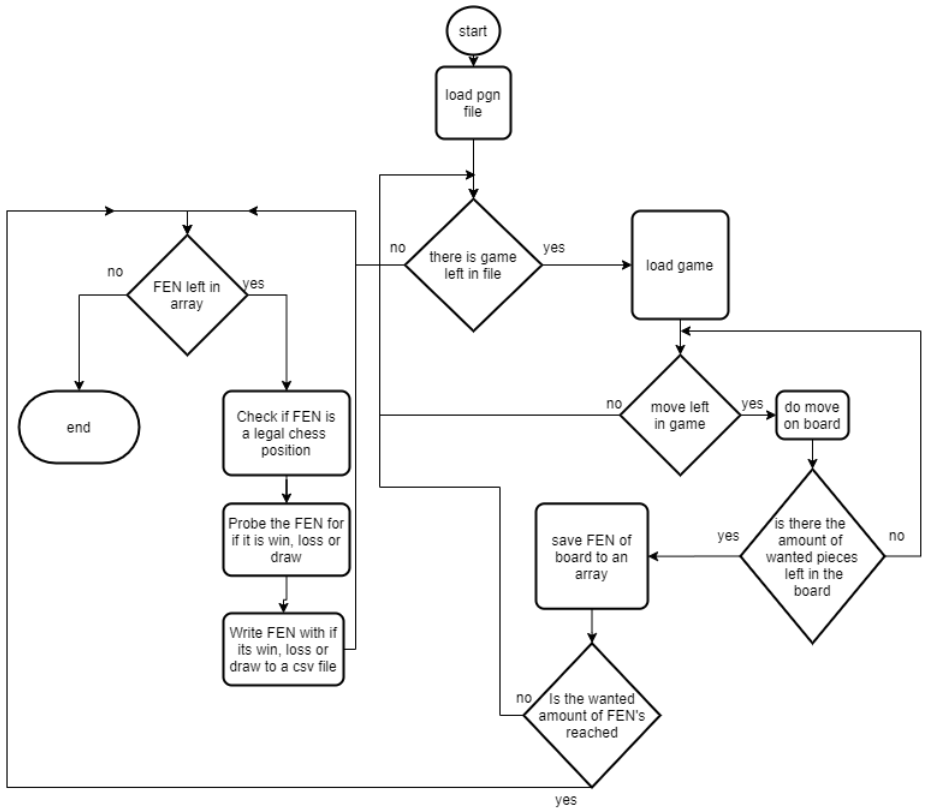

Figure 4.2: Flowchart for generation of data

The end result of the data generation was to get 4 different csv files, one for each of the amount of pieces and one csv file that had all the three files added together to make one bigger dataset for all pieces. Both the 4 and 5 piece file contains 50 000 entries, while the 3 piece file contains a little less than 40 000 entries.

4.2.2 Data Transformation

Since the form of the chess boards uses FEN it had to be translated from FEN to some bit representation. Such that it can be used by the Tsetlin Machine. Considering the fact that the Tsetlin Machine learns patterns, the representation would consist off all squares on the board. The representation would therefore be an 8x8 board where all the squares have been turned into

some bit form to represent them. An empty square would therefore have 0 bits set, while squares with pieces on them would have some combination of bits to represent them. The final look of the board would therefore depend on the bit representation of the pieces.

The first representation considered was a 7 bit encoding. Where the first bit represents which side the piece is on, the next bits represent what type of piece it is. The big problem found with this approach was that the first would represent either white or black, and could not represent neither, nor both. Which would not be important for the training, but was more for the representation of the clauses. Since it would become harder to represent a board that would vote if there was any piece on a square, or if there where to be none on a square.

The second representation was therefore a 12 bit encoding. Where the 6 first bits represents a piece that is white, and the 6 last bits represent a piece that is black. Figure 4.3 shows an illustration of how what each bit represents. The white letters means a white piece, and the black a black piece.

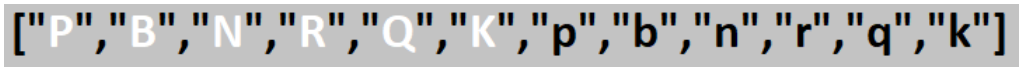

Figure 4.3: Illustration of what piece each bit represents

This represents a chess board, but does not include information about who is moving. This was included two different ways. One which adds a bit for the player making the move, and another where the amount of bits, the board itself, are doubled. Where the first part represents white making the move, and the second half represents black making the move. The different methods are both used.

Non Convolutional does not need reshaping of the bit board in order to train on it. It is easier to add an extra bit, where boards for Convolutional needs reshaping. For one of the methods of Convolutional, adding the extra bits for the moving player, was not included since the basis of the method is to have different Tsetlin Machines based on the player making the move.

When creating the different methods it takes FEN boards as an input, where the transformation is handled from there. Such that one can easily give the methods games, that they then handle themselves.

4.3 Tsetlin Machine implementation

The Tsetlin Machine implementation uses the pyTsetlinMachine implementation, which is available at Github[1], in order to create and train Tsetlin Machines of the different versions. This library is also used when getting the clauses from a Tsetlin Machine. Though this is the main library pyTsetlinMachineParallel was used for most of the training. Since this library allows faster training by leveraging more threads or cores of the CPU rather than only using one. Apart from this it uses the same implementation as pyTsetlinMachine. It is also available at Github[2].

The weighted clauses are also a part of the pyTsetlinMachine library and are therefore used, since it provides greater speed[1]. The weights for each clause are gotten through the built-in function get_state. Which gets both the state of the clause and its weight.

For this report multiple methods was created in order to see if one could achieve a Tsetlin Machine that could be used for playing endgame chess. Where the methods refer to different ways of handling the data. To make this easier the Tsetlin Machine libraries were used in order to create a class that would allow for easier creation of these methods, to make training and testing easier to handle. The implementation also provides a way of storing and loading trained machines to files. Where that storing and handling automatically handles the parameters and version of Tsetlin Machine it is to load. This is intended to give easier access to the various trained machines without having to retrain them for a test.

[1] https://github.com/cair/pyTsetlinMachine
[2] https://github.com/cair/pyTsetlinMachineParallel

4.3.1 Clauses

To get clauses for the Tsetlin Machine, the tm_action function from the Tsetlin Machine library was used. This function gets the action that the various Tsetlin Automata will take. In order to use the function one has to provide the ta_action function with the class, clause and the Tsetlin Automatas for that clause. One therefore has to take into account the size of the amount of Tsetlin Automata for the clause, since a clause consists of more than one Automata. For the Convolutional version one has to specify the size of the window given, but also extra bits for the placement of the clause on the board.

Clauseoutput	Meaning
-1	The given bit must be excluded
0	The given bit is not looked at
1	The given bit must be included
2	The given bit has both the include bit and exclude bit set. The clauses is therefore invalid

Table 4.1: The representation used in the clauses

Apart from this the amount of bits for the clause is double the specified window. Since a clause consists of both include and exclude bits. The include bit and exclude bit was put together into one value in order to reduce the total size of the clause, and to better visualise the clause. Table 4.1 shows the representation used when looking at the clauses. Figure 4.4 shows how a clause will look like for a Convolutional Tsetlin Machine with a window of 3x3.

```
[ 0  0  0  0  0  1  0  1  1  0  0  0], [ 0 -1 -1  0  0  0  0  0 -1 -1 -1 -1], [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
[-1 -1 -1 -1 -1 -1  0 -1 -1 -1 -1 -1], [-1  0  0  0 -1 -1 -1 -1 -1 -1 -1 -1], [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
[-1  0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1], [-1  0 -1 -1 -1  0 -1 -1 -1 -1 -1 -1], [-1  0 -1  0 -1 -1 -1 -1 -1  2 -1]
```

Figure 4.4: Visualization of a clause

4.3.2 Convolutional Tsetlin Machine

The difference between the usage of the Multiclass and Convolutional versions are not big. Since the functions needed for both are similar. The biggest differences would be the initialization of the versions, and how the clauses are gotten for each. The difference of initialization can be seen in 4.5. Where the Convolutional version needs a tuple that decides the size of the patterns it is to learn. The functionality and behavior of them are different. Since one needs a different set of hyper-parameters, and that the accuracy would be different. Though the usage of the versions are similar.

When working with the Convolutional Tsetlin Machine one also needs to reshape the data. Since the Convolutional version works with windows and learning patterns from these windows. In order to reshape the arrays, numpy was used, since it has functionality that manipulates data and arrays, and that the Tsetlin Machine takes numpy arrays as input.

```
if convolutional:
    from pyTsetlinMachine.tm import MultiClassConvolutionalTsetlinMachine2D as TM
else:
    from pyTsetlinMachine.tm import MultiClassTsetlinMachine as TM

def MakeTestlin(Clauses,t,S,Epochs):
    def GetMachine():
        if convolutional:
            return TM(Clauses, t, S, (WindowX, WindowY), weighted_clauses=True, boost_true_positive_feedback=0)
        else:
            return TM(Clauses, t, S, weighted_clauses=True, boost_true_positive_feedback=0)


    tm = GetMachine()
```

Figure 4.5: Difference between Multiclass and Convolutional

4.4 Methods

This section specifies the various methods used in the Tsetlin Machine. They are created in order to see what would be the best method for the Tsetlin Machine, when it comes to chess. For the most part they are intended to split or change the data.

4.4.1 Non Convolutional

An implementation which only changes the data so it can be used in a non Convolutional Tsetlin Machine. It is meant to show the differences with the other methods. It will only split the data into training and testing.

For each method, the data will be transformed or changed before training. This can be seen in figure 4.7. Though the way the data is transformed is different.

4.4.2 Convolutional

An implementation which changes the data so it can be used with the Convolutional Tsetlin Machine. It is meant to show the difference with the other methods. It splits the data into training and testing sets. It also reshapes the data so it becomes an 8x8 window for the Convolutional Tsetlin Machine.

4.4.3 Reversing a player

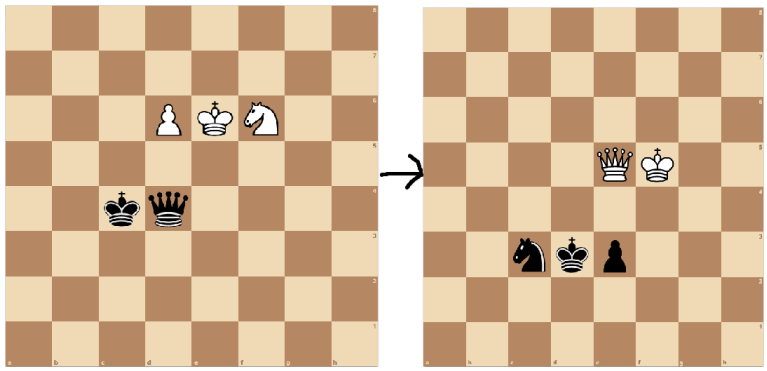

Figure 4.6: Illustration of flipping the board

The main point of this method is to change the given player. Such that the machine only gets the results of seemingly one player, and reduce the variance and overlap between the different results.
If the given player is white, the pieces will have their side changed and their position on the board will be flipped. A piece that was placed on A1 will now be placed on H8. Figure 4.6 shows how this would be done.

4.4.4 Split by moving player

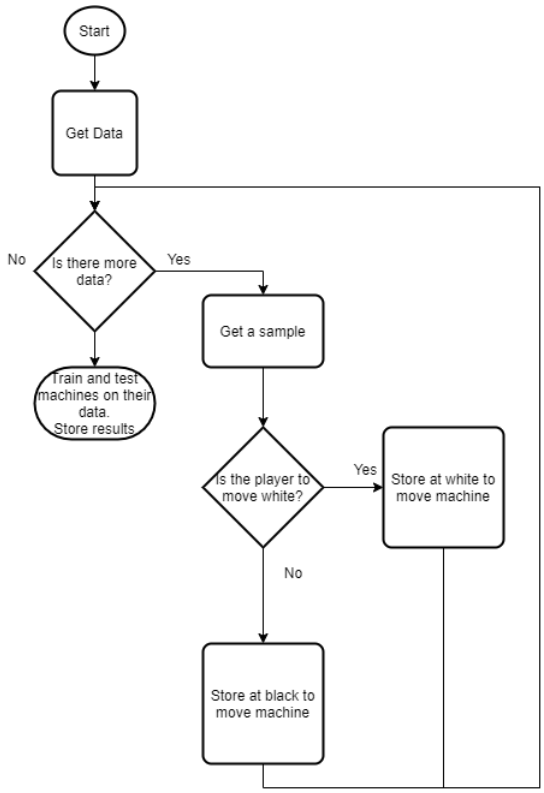

Figure 4.7: Flowchart for splitting data

The main point for this method is to create two Tsetlin Machines. Where one would be given all the samples for the white player, while the other for the black player. To see if these machines can then improve by only being shown samples for one player, and not having data from the other.

4.4.5 Split by result

This method splits the data into two. In set one all losses and draws have been turned into the same class/sample. Meaning that win is 1 and loss+draw is 0. The other set only consists of the losses and draws. The first set would then be used to see if the position is a win or not. If it is not

the other set would be used to see if the result is loss or draw. This is done in order to see if the differences between the samples are similar or not. It is also done in order to see if splitting the data, and trying to make it focus on one class will help the classification. Figure 4.8 shows the 2 machines, and how the data would be split.

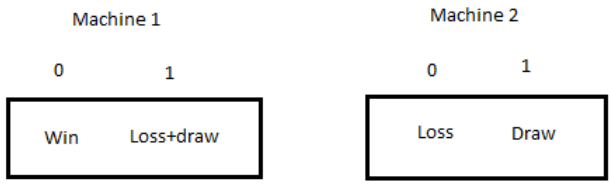

Figure 4.8: The two machine in the Split results method

4.5 Testing

For each set of tests the better performing Tsetlin Machine versions will be used for further testing. This is done in order to reduce training and testing time, and also to better focus on the methods that could provide better results.

4.5.1 10-Fold Cross Validation

This implementation uses the Scikit-learn implementation in order to get data sets used for validation. Cross validation means that it splits the data into different partitions. Where the partitions depend on the amount of folds one want. For this project 10 folds was used, which means the data was split into 10. 9 of these will be used for training while the last partition will be used for testing. The partitions will then switch around until all the different partitions have been used for testing, as shown in Figure 4.9. This is done in order to be sure that the results from training and testing where not from just getting a good split when training the data, and to be sure that the accuracy gotten is correct.

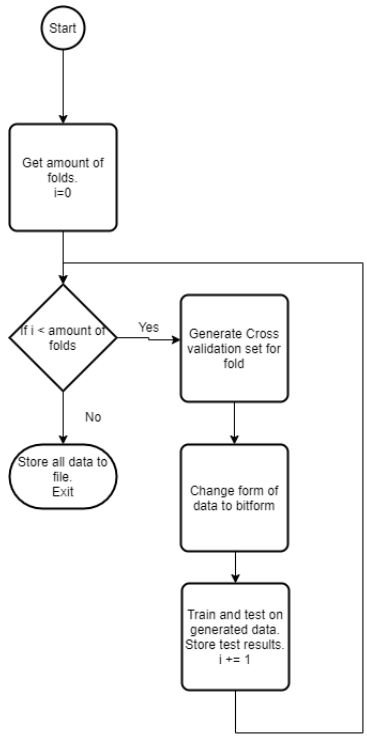

Figure 4.9: Flow chart of how the testing is done for the methods

Stratified means that the ratio of classes are kept for the different partitions. Meaning that if 10% of the data consists of draws, 10% of the training data will be draw and 10% of the testing data will be draw. Which is helpful for data where there is an unequal amount of different samples in the data.

10-folds was chosen since it uses most of the data for training. This means there are fewer testing examples, which can impact the validity of the results, since a small data-set will mean that there is an even smaller amount of testing cases. If there were 100 examples, 10 of these examples would be the testing. Where it could be correct 50% of the time, but would have a lower percentage correct if more test cases were included. Though the splits or folds used would mitigate this since they use all the data as test data. Meaning that the percentage would be lower in other folds tested if the inclusion of certain examples would impact the result. Therefore the mean has been calculated from the results gotten from the test-cases. The

result given for each fold is the highest percentage correct that the machine got over it's epochs.

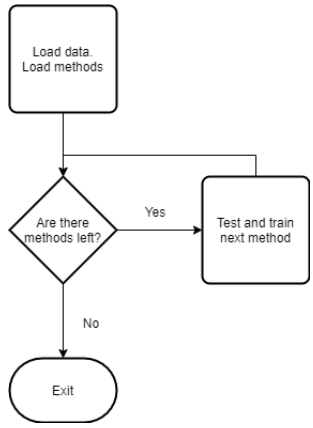

Figure 4.10: Flow Chart showing more how testing and training are done

Figure 4.10 shows how the testing is done with the different methods being chained. For each piece dataset and each window, all methods would be run for every execution of the program.

The testing will first be done for the weighted and unweighted Tsetlin Machine and Convolutional Tsetlin Machine. In order to see which of the differing versions is the better one for the task. The machines used for further testing would be the version performing the best for the different data-sets. For every method one will be chosen for every set of data. This will be based on the highest accuracy achieved by a machine. For the methods moving player and split player, which has 2 machines, the accuracy achieved for both machines will be taken into consideration. For moving player both machines would have to be weighted equally, since they are for different moving players. While for Split result the first machine would have a greater weight than the second one. Since the second machine is used if the machine determines a board as not win.

4.5.2 Tree search recall

This test is about testing if the various Tsetlin Machines can correctly evaluate the result of a game, and then calculate the recall and precision for the Tsetlin Machines. Where 100 random games will be used. For these 100 games all moves found 3 moves deep will be used. The prediction for the move will also have a score, which is based on the amount of clauses that voted against and the amount of clauses that voted for the prediction.

The moves and predictions will be sorted based on the scoring and a given amount of moves, for each game, and will be used for calculating recall and precision. It will first start with one move for each game, then two, and so on. Until it reaches 100 moves for each game. Which is done in order to see how the confidence of the Tsetlin Machine changes based on the tolerance in the amount of moves.

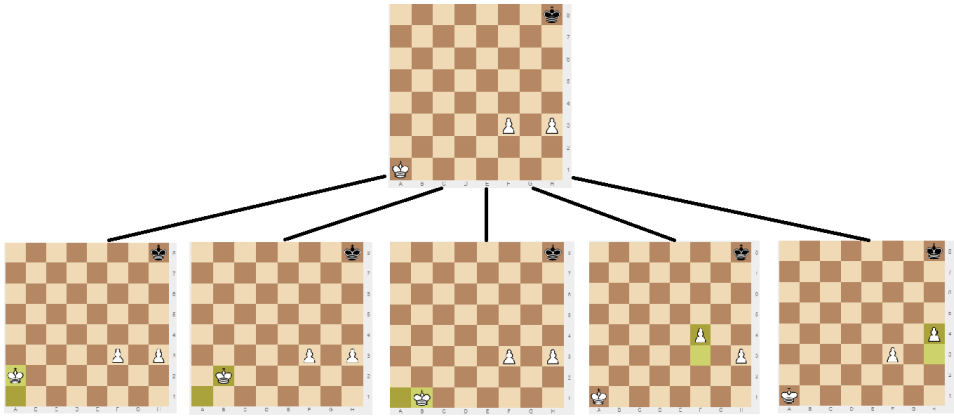

Figure 4.11: Visualised example of a 1 step three search where white is next to move

4.5.3 Playing Games

For this test the methods chosen from the previously will play against each other, and other methods for playing chess. The results gotten from this can be used in order to compare how the methods perform, and how they perform compared to each other. The other methods used is:

1. Random
    - Chooses a move randomly
2. Monte Carlo Tree Search
    - Does a Monte Carlo tree search, in which it has been limited to a minute of computational time, in order for the testing to not take too long. It should not require too much time either, since it is endgames where there are fewer variations of good moves.

In order for the Tsetlin Machine to play the games a tree search will be used in order to find moves. So that the Tsetlin Machine systematically evaluates all the moves possible from a given position. The Tsetlin Machine will then evaluate the moves, and the best move predicted as win is chosen. In order to find the best move two methods will be used to sort the evaluated moves. Where the top most, or the move with the highest score, will be chosen.
The first method is to score the moves by how confident the Tsetlin Machine is about the prediction. The second is to take the first set of moves. Evaluate the moves that can be done from these moves and predict what they are supposed to be. Finally a score is calculated for the original moves, where the percentage of predicted wins is the score.

The players or methods will play a chosen amount of games against each other. Where they will play the amount of games, switch sides and play the same amount. The Tsetlin Machine methods will therefore have their own entries when they are the starting side. The Monte Carlo Tree Search and random methods will not, and the results from these games will be placed in the same entry. Since the point is to evaluate the Tsetlin Machine methods.

One play will only be done for a certain amount of moves, since the Tsetlin Machine can move back and forth without any progression. The tablebase probe from the python-chess library is used in order to find out if the Tsetlin Machine still can obtain a win, given perfect play, from that position or rather if it has lost it would only lose. The results from the playing are written to a file. Where finishing the game within the given amount of move gives a tuple consisting of the result and a 1, and not finishing the game within the given amount of moves gives a tuple with the result taken from the endgame tablebase and a 2.

Chapter 5

Results and discussion

This chapter contains all the results from the various tests conducted. Here a subset of the Tsetlin Machines that performed well from the previous test will be used in the next test. So that the more interesting or higher performing Tsetlin Machines can be further tested. In order to try and find methods that perform well.

When Multiclass Tsetlin Machine is mentioned it refers to the Non-convolutional Tsetlin Machine. Even though the Convolutional Tsetlin Machine is also multiclass.

5.1 10-Fold Cross Validation

This test is a cross validation of different windows for the Convolutional Tsetlin Machine and the Non-convolutional Tsetlin Machine. It is also used in order to see the difference between a Tsetlin Machine with weights and a Tsetlin Machine without weights.

In all tables Pieces Amount refers to the data used for that test. Where 3 means there are 3 pieces left, and all means all the datasets have been used (meaning it is a combination of 3, 4 and 5 pieces data sets). Avg. Accuracy means the mean of the highest accuracy from the folds for that test. Window refers to the size of the pattern that the Convolutional Tsetlin Machine learns.

5.1.1 Multiclass

Table 5.1 shows the result of the non weighted Multiclass Tsetlin Machine. The result shows under 50% accuracy at the best. Which is close to what would be reached by just guessing the most common occurrence. The machine chooses one class over the others a significant amount of times. So this setup with using the normal Multiclass Tsetlin Machine is clearly not a good way to solve how to classify chess using a Tsetlin Machine.

Pieces Amount	Avg. Accuracy
3	46.59 ± 0.68
4	43.86 ± 0.40
5	45.14 ± 0.37
All	43.07 ± 0.20

Table 5.1: Result of 10-fold validation on non weighted Multiclass Tsetlin Machine

From the table 5.1 the 3 pieces dataset had the highest accuracy, though this does not correlate with its ability to play the game. Apart from this the accuracy is about the same for all datasets.

5.1.2 Convolutional

Pieces Amount	Window	Avg. Accuracy
3	8x8	46.20 ± 0.42
3	7x7	45.97 ± 0.96
3	5x5	58.53 ± 0.65
3	3x3	63.5 ± 0.7
3	2x2	64.13 ± 0.73
4	8x8	43.74 ± 0.32
4	7x7	44.76 ± 0.51
4	5x5	53.95 ± 0.53
4	3x3	54.96 ± 0.43
4	2x2	54.97 ± 0.58
5	8x8	45.1 ± 0.29
5	7x7	46.78 ± 0.45
5	5x5	49.87 ± 0.35
5	3x3	51.42 ± 0.42
5	2x2	52.13 ± 0.49
All	8x8	43.15 ± 0.18
All	7x7	43.13 ± 0.27
All	5x5	48.25 ± 0.54
All	3x3	50.61 ± 0.38
All	2x2	51.07 ± 0.39

Table 5.2: Result of 10-fold validation on non weighted Convolutional Tsetlin Machine

The Convolutional Tsetlin Machine's result showed in table 5.2 shows that the results for all the different datasets increase when looked at in smaller windows. This probably has to do with the complexity of chess, where many different possible positions exist even in the endgame. This makes it hard for the Tsetlin Machine to learn. When using a window the different patterns created from a window can be used to match different places on the board, instead of having to match the entire board. More information can be added to the various clauses or patterns, or more patterns can vote on a given board. This usage of more smaller clauses spread across a board, would result in a combined effort to detect more cases.

5.1.3 Multiclass w. Weights

Pieces Amount	Avg. Accuracy
3	47.10 ± 0.62
4	43.86 ± 0.32
5	45.1 ± 0.28
All	43.05 ± 0.35

Table 5.3: Result of 10-fold validation on weighted Multiclass Tsetlin Machine

The accuracy is still about the same for the different datasets, and is about the same as Tsetlin Machine without weights.

5.1.4 Convolutional w. weights

Pieces Amount	Window	Avg. Accuracy
3	8x8	47.02 ± 0.7
3	7x7	46.30 ± 0.58
3	5x5	57.08 ± 0.71
3	3x3	61.52 ± 0.65
3	2x2	62.41 ± 0.88
4	8x8	43.90 ± 0.46
4	7x7	44.53 ± 0.5
4	5x5	52.2 ± 0.4
4	3x3	54.4 ± 0.47
4	2x2	62.43 ± 0.81
5	8x8	45.06 ± 0.28
5	7x7	46.8 ± 0.5
5	5x5	49.91 ± 0.50
5	3x3	51.26 ± 0.51
5	2x2	52.17 ± 0.59
All	8x8	43.15 ± 0.24
All	7x7	43.08 ± 0.21
All	5x5	47.78 ± 0.32
All	3x3	50.37 ± 0.24
All	2x2	50.62 ± 0.31

Table 5.4: Result of 10-fold validation on weighted Convolutional Tsetlin Machine

As with the previous test without weights, the accuracy on the various datasets increases with smaller window sizes.

5.1.5 Comparison

Type of Machine	Pieces Amount	Accuracy
Multiclass	3	46.59 ± 0.68
Multiclass w. Weights	3	47.10 ± 0.62
Convolutional	3	64.13 ± 0.73
Convolutional w. Weights	3	62.41 ± 0.88
Multiclass	4	43.86 ± 0.4
Multiclass w. Weights	4	43.86 ± 0.32
Convolutional	4	54.97 ± 0.88
Convolutional w. Weights	4	62.43 ± 0.81
Multiclass	5	45.14 ± 0.37
Multiclass w. Weights	5	45.03 ± 0.26
Convolutional	5	52.13 ± 0.49
Convolutional w. Weights	5	52.17 ± 0.59
Multiclass	All	43.07 ± 0.2
Multiclass w. Weights	All	43.05 ± 0.35
Convolutional	All	51.07 ± 0.39
Convolutional w. Weights	All	50.62 ± 0.31

Table 5.5: Comparison of the Multiclass and Convolutional tests

From table 5.5 the weighted Tsetlin Machine versions perform about the same as the unweighted Tsetlin Machine versions. Except for the 4 pieces dataset, where the Convolutional version with weights achieves 62.43% versus the no weights 54.97%.

This does go against the assumption of weighted outperforming the non weighted. Since the weighted version should allow for more clauses, since

clauses that have been seen before gets their weight increased. These added clauses should allow for better detection of general cases and also more edge cases, since it allows more uncommon clauses to appear. Though the big difference in the 4 pieces dataset might suggest that the hyper-parameters hasn't been optimized enough to make the difference between the methods clearer, or that the datasets themselves doesn't allow for the learning of more cases.

Where the weighted Tsetlin Machine should reduce the amount of needed clauses in order to achieve the same accuracy, while also reducing the computation time[1].

5.1.6 Method Classification Experiments

In this section three different methods are looked at for the Tsetlin Machine. All the tests used the Convolutional Tsetlin Machine since it performed better on average. Each of the methods was tested for different window sizes with the same hyper parameters from the previous test being 4000 clauses, 8000 Threshold and 10 S for 50 epochs. There are a couple of the tests that include a bit for moving player and one test without draws, to see how this would impact the accuracy.

Flipped Player

Pieces Amount	Window	Inc. draw	Moving player bits	Avg. Accuracy
3	8x8	Y	N	69.21 ± 0.85
3	7x7	Y	N	66.66 ± 0.91
3	7x7	Y	Y	72.78 ± 0.37
3	5x5	Y	N	86.82 ± 0.58
3	3x3	Y	N	90.97 ± 0.35
3	2x2	Y	N	88.45 ± 0.31
4	8x8	Y	N	67.24 ± 0.39
4	7x7	Y	N	64.66 ± 0.48
4	5x5	Y	N	76.95 ± 0.67
4	5x5	N	Y	94.18 ± 0.31
4	5x5	Y	Y	75.42 ± 0.79
4	3x3	Y	N	80.24 ± 0.54
4	2x2	Y	N	88.39 ± 0.5
5	8x8	Y	N	66.64 ± 0.32
5	7x7	Y	N	70.53 ± 0.31
5	5x5	Y	N	77.09 ± 0.34
5	5x5	Y	Y	76.7 ± 0.42
5	3x3	Y	N	79.32 ± 0.52
5	2x2	Y	N	80.64 ± 0.39
All	8x8	Y	N	63.4 ± 0.35
All	7x7	Y	N	62.63 ± 0.37
All	5x5	Y	N	74.55 ± 0.41
All	3x3	Y	N	77.62 ± 0.72
All	2x2	Y	N	78.13 ± 0.61

Table 5.6: Results from the Flipped player method

Flipped player flips the board so that all boards become the viewpoint of a specified player. It might therefore be easier for the Tsetlin Machine classify, as it only has to classify from the viewpoint of 1 player. The added data also gives more examples on what the different classifications are. Since the flipping is a mirroring of the boards and both players should be able to reach most of the same positions.

Moving Player

Pieces Amount	Window	Inc. draw	Moving player bits	Avg. Accuracy White	Avg. Accuracy Black
3	8x8	Y	N	68.56±0.67	68.58±0.75
3	7x7	Y	N	67.42±1.09	67.21±0.82
3	7x7	Y	Y	67.38±0.65	67.24±0.64
3	5x5	Y	N	87.08±0.47	86.59±0.62
3	3x3	Y	N	91.42±0.65	90.49±0.81
3	2x2	Y	N	88.44±0.79	87.93 ± 1.0
4	8x8	Y	N	68.47±0.53	68.31±0.60
4	7x7	Y	N	65.35±1.13	65.47 ± 1.0
4	5x5	Y	N	77.23±0.71	76.97±0.77
4	5x5	N	Y	95.27±0.53	94.68±0.66
4	5x5	Y	Y	77.43±1.04	76.87±0.71
4	3x3	Y	N	80.61±0.76	80.42±0.62
4	2x2	Y	N	88.34±0.75	88.1 ± 1.29
5	8x8	Y	N	67.55±0.41	66.93 ± 0.5
5	7x7	Y	N	66.59±0.70	66.2 ± 0.64
5	5x5	Y	N	76.58±0.66	77.09±0.73
5	5x5	Y	Y	76.83±0.49	77.08±0.75
5	3x3	Y	N	79.09±0.80	79.47±0.73
5	2x2	Y	N	80.48±0.80	80.93±0.44
All	8x8	Y	N	64.52±0.43	64.01±0.36
All	7x7	Y	N	63.24±0.54	62.98±0.67
All	5x5	Y	N	74.62±0.43	74.63±0.42
All	3x3	Y	N	77.88±0.59	77.87 ± 0.5
All	2x2	Y	N	78.1 ± 0.41	78.15±0.54

Table 5.7: Results from the Moving player method

Moving player splits the job into two different Tsetlin Machines. This means that one machine is dealing with the viewpoint of a certain player. It does something similar to Flipped player, but splits the datasets for the two machines. When reviewing the performance, one has to take both machines into consideration when looking at the percentages. Where the highest scoring machine is from Moving player, though its other machine scores a bit lower. A singular machine with the same accuracy will probably perform

better overall, since the machines for Moving player are only tested on data that is specific for them.

Split Result

Pieces Amount	Window	Inc. draw	Moving player bits	Avg. Accuracy Win	Avg. Accuracy Other
3	8x8	Y	N	67.82±0.38	62.38±0.39
3	7x7	Y	N	67.17±0.37	60.80±0.64
3	7x7	Y	Y	90.73±0.65	66.87±0.94
3	5x5	Y	N	72.99±0.73	69.42±0.86
3	3x3	Y	N	74.84±0.70	78.99±1.01
3	2x2	Y	N	73.88±0.78	78.63±0.80
4	8x8	Y	N	68.35±0.15	62.88±0.50
4	7x7	Y	N	68.61±0.17	63.31±0.45
4	5x5	Y	N	69.86±0.44	71.30±0.61
4	5x5	Y	Y	86.45±0.46	83.02±0.46
4	3x3	Y	N	70.41±0.42	72.21±0.59
4	2x2	Y	N	73.97±0.77	79.13±0.84
5	8x8	Y	N	66.05 ± 0.1	68.02±0.24
5	7x7	Y	N	65.99±0.13	69.72±0.43
5	5x5	Y	N	66.38±0.24	73.27±0.59
5	5x5	Y	Y	88.08±0.23	82.41±0.47
5	3x3	Y	N	67.40±0.32	74.52±0.58
5	2x2	Y	N	67.79±0.47	75.14±0.65
All	8x8	Y	N	66.48±0.03	61.88±0.23
All	7x7	Y	N	66.60±0.17	62.0 ± 0.43
All	5x5	Y	N	67.67±0.27	64.14±0.77
All	3x3	Y	N	68.93±0.27	66.63±0.68
All	2x2	Y	N	69.69±0.34	66.56±0.48

Table 5.8: Results from the Split result method

Split result splits the data according to what the result is. There are two Tsetlin Machines, where the first predicts if it is win or not, and the second

predicts if it is draw or loss. The first is consistently used for all the data, while the second is only used for data that the first machine did not predict as a win. Because of this splitting there could be a bigger difference in what is considered win versus what is not. Since one machine would only have to look at the difference between it and the rest, and there being only 2 classifications to vote on as well.

Comparison

Type of method	Pieces Amount	Avg. Accuracy 1	Avg. Accuracy 2
Flipped Player	3	90.97 ± 0.35	-
Flipped Player	4	94.18 ± 0.31	-
Flipped Player	5	80.64 ± 0.39	-
Flipped Player	All	78.13 ± 0.61	-
Moving Player	3	91.42 ± 0.65	90.49 ± 0.81
Moving Player	4	95.27 ± 0.53	94.68 ± 0.66
Moving Player	5	80.48 ± 0.8	80.93 ± 0.44
Moving Player	All	78.1 ± 0.41	78.15 ± 0.54
Split Result	3	90.73 ± 0.65	66.87 ± 0.94
Split Result	4	86.45 ± 0.46	83.02 ± 0.46
Split Result	5	88.08 ± 0.23	82.41 ± 0.47
Split Result	All	69.69 ± 0.34	66.56 ± 0.48

Table 5.9: Comparing the best results from the methods

From what can be seen in table 5.9 the different methods manage to achieve high accuracy on the different datasets. It can also be noted that several methods achieved over 90% on a dataset. These percentages are much higher than the previous tests, where there were no changes to the data.

Some of the percentages given have added information or there are other factors. These are; which side to move is added to the bit representation of the board, removing draw from the dataset, or a combination of both. Where Moving player for 4 pieces has had the draws removed from the

dataset. Even though this improved the results for a given method there where other methods that achieved almost the same accuracy without removing draw.
The added bits to the representation also increased the percentage in most instances. This suggests that the added data, or added information, in the representation could also further increase the overall performance of the different methods. Though this was not tested thoroughly.

Though the best versions have certain extra factors, such as adding extra information or removing draws, the Moving player and Flipped player methods perform about the same, and both perform better than the Split Result method. It can also be noted that one of the two methods performs better on some datasets, while the other performs worse and vice versa. Also when looking at the highest performing versions of the methods, they perform about the same. Showing that either splitting the machines based on the moving player or flipping the board for one of the players would give better results.

5.2 Tsetlin Machine with tree search

In order to better see how the highest rated methods perform on a game of chess, the Tsetlin Machines shown in the comparison table 5.9 has been used in this testing. Below are some of the graphs from different three search tests to look at how the prediction and recall changed based on how many of the top scoring moves was included.

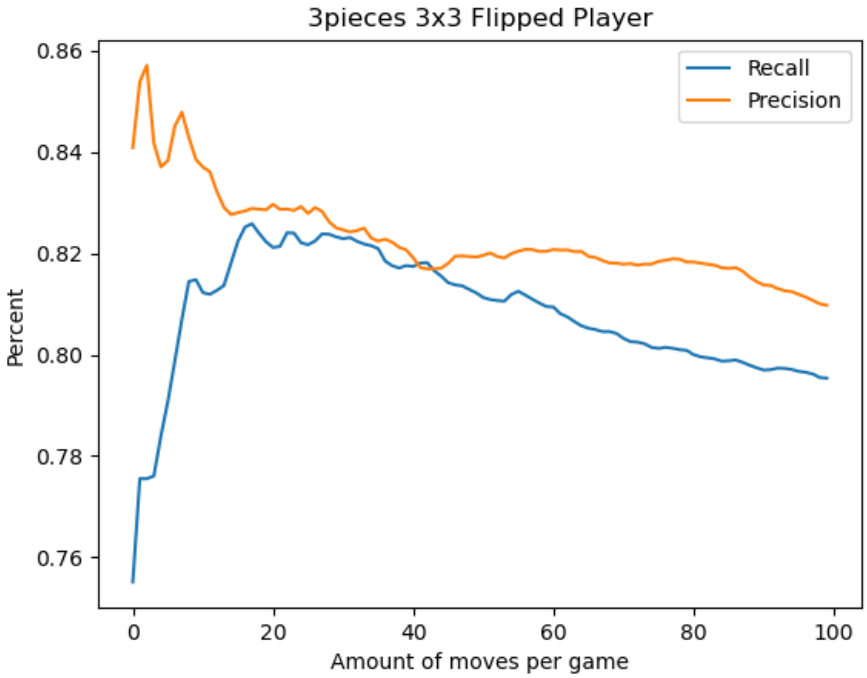

Figure 5.1: Graph off recall for a Tsetlin Machine

In figure 5.1 shows a test done on the machine trained with 3 pieces dataset, with the flipped player method and a window size of 3x3. From the figure it can be seen that the recall is at its highest around 20 moves before it slowly decreases. Meanwhile the precision starts at its best with the least amount of moves from each game, and slowly decreases from there. Showing that the first set of moves are accurately predicted, even though it does not find many of the possible winning moves. As the amount of moves increases the Tsetlin Machine starts to be less accurate, though it finds more of the winning moves.

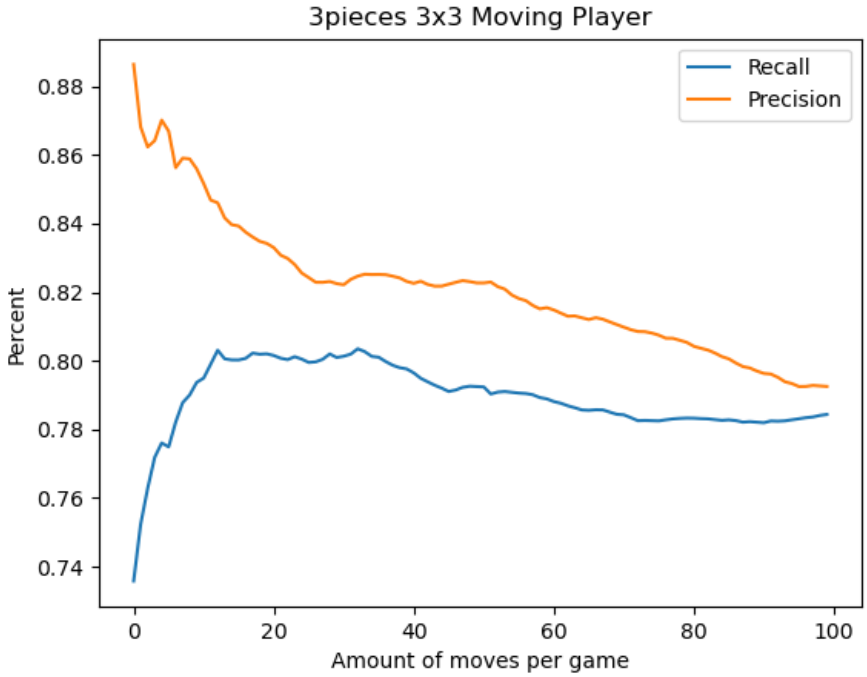

Figure 5.2: Graph off recall for 3 piece Moving player window 3x3

Figure 5.2 shows a test done on the machine trained with the 3 piece dataset, with the moving player method and a window size of 3x3. The figure shows that as with the previous test the precision slowly decreases as more moves are added. The recall increases until around 20 moves included where it stays almost the same but decreases slightly for the rest of the moves.

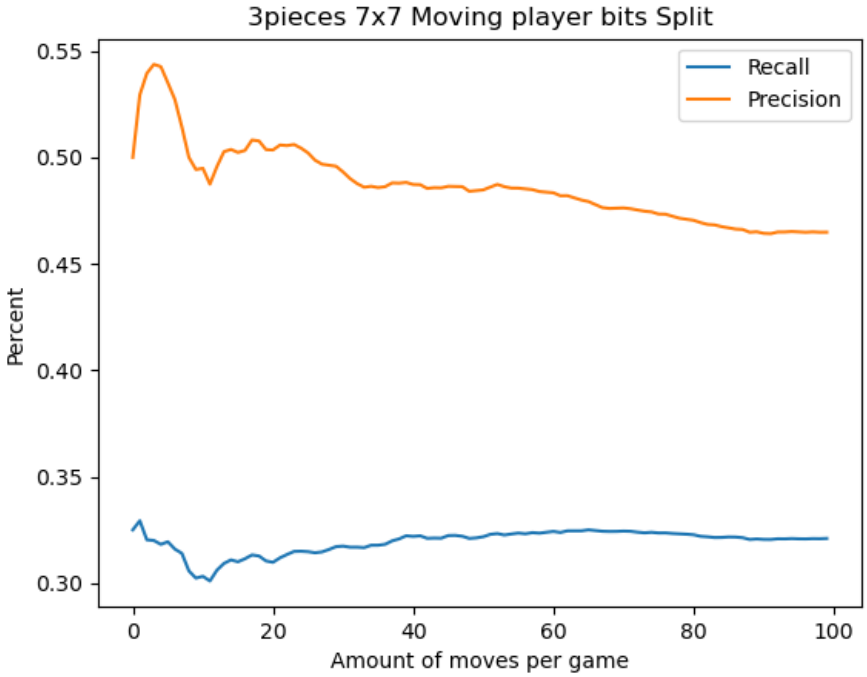

Figure 5.3: Graph off recall for 3 piece split results window 7x7

The recall of this method is low, where only around 30% of the total amount of winning positions are correctly classified. Apart from this the graph 5.3 shows that out of all the ones it found, meaning that it classified more as win, around half of the ones classified were actually win. Showing that the total amount of win classifications were relatively low. The graph also shows that the machine performed best on its most confident predictions, while its precision became worse as more moves were added.

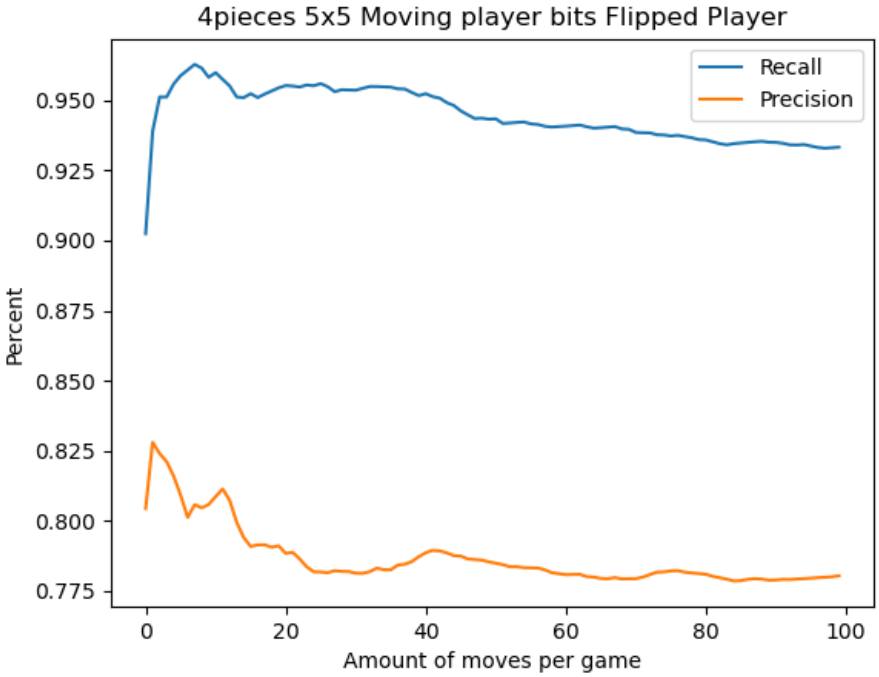

Figure 5.4: Graph off recall for 4 piece Flipped player window 5x5

The graph 5.4 shows that the method has a generally high percent recall, though it peaks early and gets lower as more moves are added. Though it is generally above 90%, this might come from not having draws included in this machines training. This is one of the best results for these tests, apart from the precision starting out at 80% and getting lower as more moves are added. It shows that while the method manages to find most of the correct wins, it also manages to classify the other moves wrongly. Meaning that if one has to search through more than the first 10-20 moves, the method becomes more uncertain and gets worse at finding the moves that give a win.

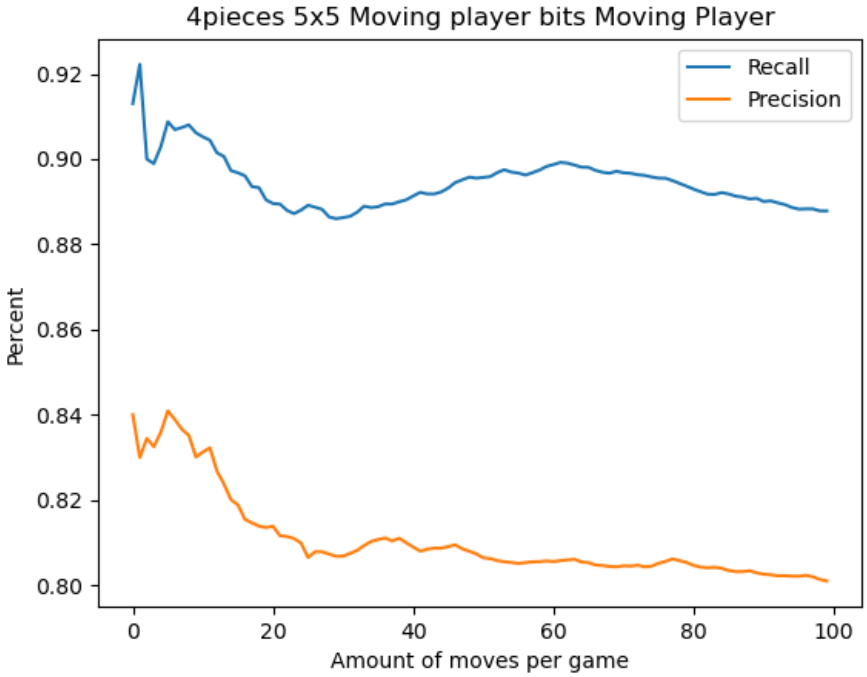

Figure 5.5: Graph off recall for 4 piece Moving player window 5x5

The graph 5.5 starts with a high recall and also a high precision. Then the method becomes more uncertain as more moves are added. Indicating that the highest scoring moves are the most secure moves, but as one includes more it becomes worse. Though this could also stem from the fact that this method was trained on the dataset without draws. Such that it knows how to classify win and loss, but has problems when trying to classify draw.

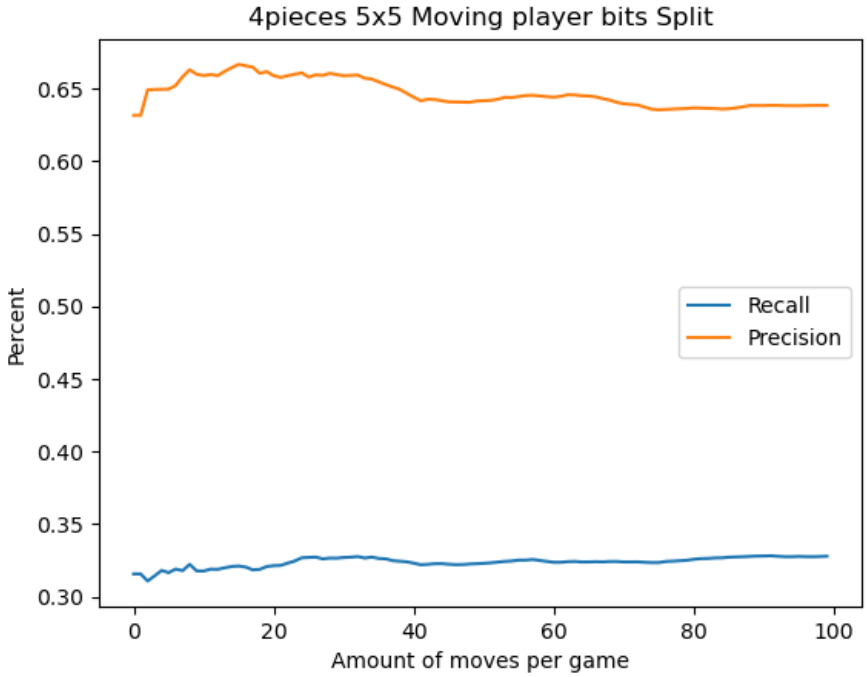

Figure 5.6: Graph off recall for 4 piece split results window 5x5

The graph 5.6 shows a method which has low recall. It finds barely any of the moves, though this improves as one adds more moves. Though recall is low the precision is a lot higher, so that even though it does not find all the moves, it is more accurate in its prediction. But these results are still low.

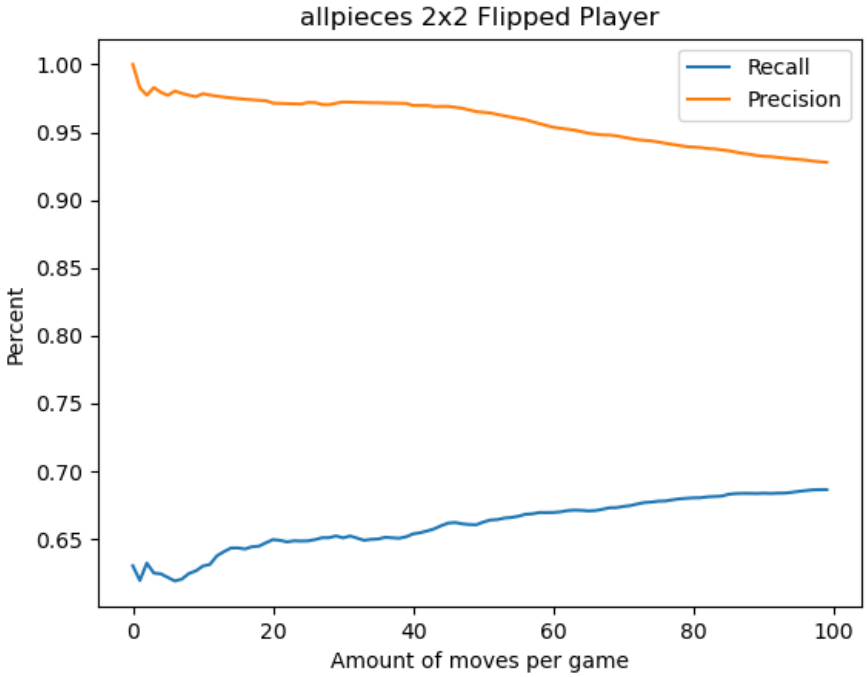

Figure 5.7: Graph off recall for all pieces Flipped player window 2x2

Figure 5.7 shows a test run on the machine that was trained on the all pieces dataset, with the flipped player method and a window of 2x2. The figure shows that the precision is very high, although it drops off the more moves gets added. The recall on the other hand starts out pretty low at around 0.62 but slowly increases when the amount of moves looked at increases.

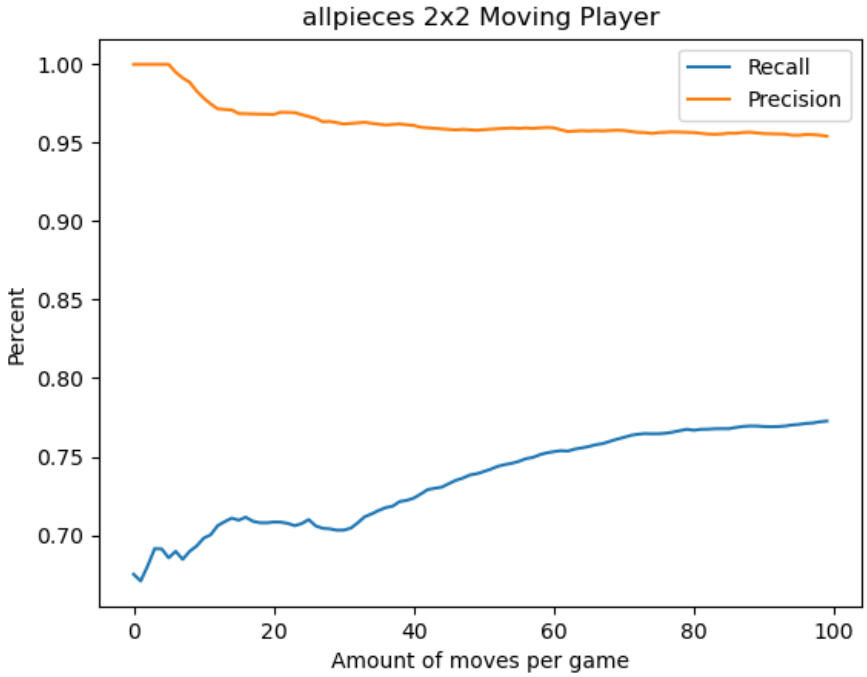

Figure 5.8: Graph off recall for all pieces Moving player window 2x2

Figure 5.8 shows a test run on the machine that was trained on the all pieces dataset, with the moving player method and a window of 2x2. This had a very similar result to the previous test on the same specifications with the flipped player method. As with that method this figure also shows that the precision is very high with a slight decrease with more moves, and a recall that starts around 0.66 that slightly increases the more moves gets added

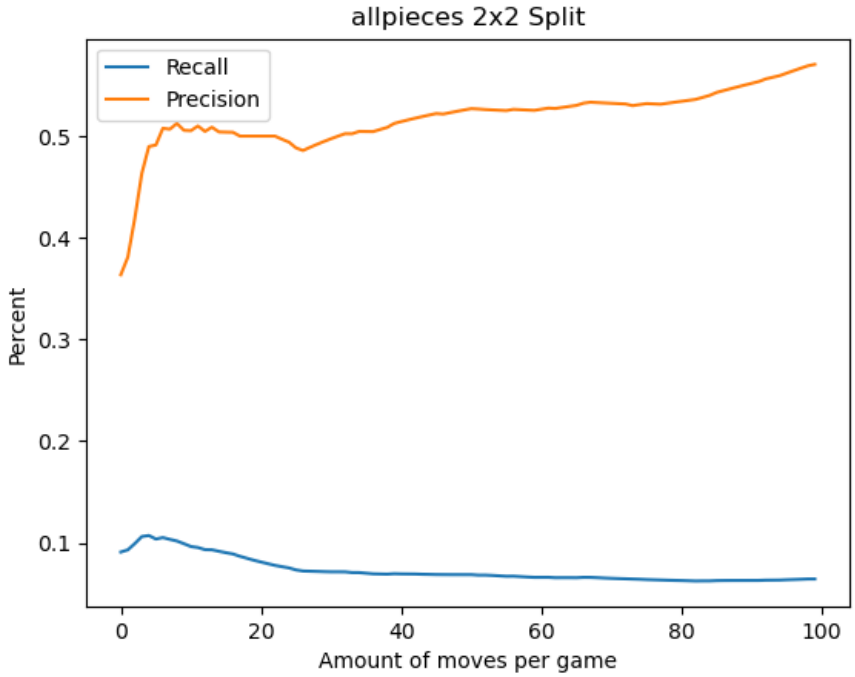

Figure 5.9: Graph off recall for all pieces Split results window 2x2

Figure 5.9 shows a test run on the machine that was trained on the all pieces dataset, with the split results method and a window of 2x2. This figure shows that this function does not have the same results as the moving player and flipped player methods have. The figure shows that the precision is around 0.50, and the recall is at 0.1. This shows that this method is not very suited for the task of determining good moves.

5.2.1 Comparison

This chapter details a comparison of the average precision and recall of the various methods.

Type of method	Pieces Amount	Precision	Recall
Flipped Player	3	82.31	80.81
Flipped Player	4	78.67	94.49
Flipped Player	5	95.37	69.31
Flipped Player	All	95.88	66.07
Moving Player	3	82.12	78.92
Moving Player	4	81.05	89.46
Moving Player	5	93.41	63.6
Moving Player	All	96.4	73.51
Split Result	3	48.69	31.93
Split Result	4	64.67	32.38
Split Result	5	58.15	29.21
Split Result	All	51.78	7.29

Table 5.10: Comparing the results from the different methods used in the tree search

From the tree search tests done that are displayed in 5.10 it can be seen that the split results method performed significantly worse than the other methods. This is likely because it has to determine the game while analyzing for both players, where as the two other methods look at it from only one players perspective.

Both the flipped player and moving player method performed almost the same on all the different tests. For both of them the recall and precision is around 80% when using the version trained on the 3 piece data. The machine trained on 4 pieces has a slight increase in recall, going up to 94.49% for the flipped player method and 89.46% for the moving player method. While the recall did improve the precision did not, going down a tiny bit for both methods. With the 5 piece and all pieces data sets the percentage for the precision seems to increase, going over 90% for all of

them. The recall however dropped off significantly for these machines to around 63-73% for the tests.

Even with a lower recall the machines with a high precision are better. This comes from that even with a medium recall you still pick up winning moves, but if the recall is high and the precision is low, many of the predicted winning moves will be false positives.

Overall most of the methods had a drop in recall and precision as more moves where added. Showing that the Tsetlin Machine manages to perform highly on some moves, but to then starts making more mistakes.

5.3 Visualised Clauses

In this section some of the highest weighted clauses in the Tsetlin Machine is looked at to see if the patterns they have are able to give some insight into the choices made. These patterns are supposed to be interpretable for humans. When looking at the clauses 1 means the piece on that position is to be included for it to vote, and -1 means it should not be there. 0 means it does not look at what that bit is.

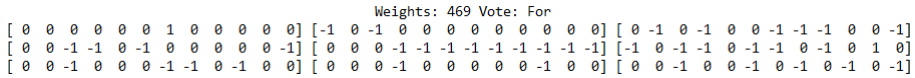

Figure 5.10: 3x3 Clause trained on 3 piece voting for win with 469 weight

Figure 5.10 shows one of the highest weighted clauses for a machine trained on the 3 piece data set with 3x3 window size. It has one bit set to 1, and multiple to -1. The 1 that is set on the 7Th bit makes it so that there has to be a black pawn in that square for the clause to vote. All the -1 means that in all those squares, the piece that the bits set to -1 represent should not be there for the clause to vote. Which piece each bit represents can be found in chapter 4.2.2.

```
      Weights: 1167 Vote: For
[ 0  0  0  0  0  0  0  0  0  0  0  1]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]

[ 0  0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]

[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0 -1  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]

[ 0  0  0  0 -1  0  0  0  0  0  0  0]
[ 0  0 -1  0  0  0  0  0  0  0 -1  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0 -1]

[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0 -1  0  0  0  0  0  0  0  0 -1]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0]
```

Figure 5.11: 5x5 Clause trained on 4 piece voting for win with 1167 weight

Figure 5.11 shows a clause gotten from a Flipped player Tsetlin Machine trained on the 4 piece dataset with a window of 5x5. In this clause there are two pieces that are set to be included, this is the black king and the white knight. For the 5x5 windows each collection of squares in this representation shows one row, and the next collection is the the next row. This was done because showing an entire row in one line would make it hard to read.

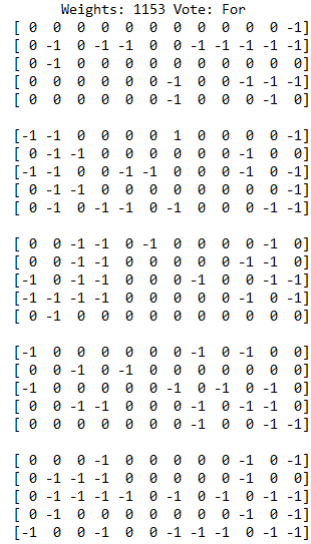

Figure 5.12: 5x5 Clause trained on 4 piece voting for win with 1153 weight

Figure 5.12 shows a clause gotten from a Moving player Tsetlin Machine trained on the 4 piece dataset with a window of 5x5. This clause votes for a win for the white side. In this clause only one piece that has been set to be included, which is a black pawn. Apart from this the rest of the clause has only exclude bits set across the clause. There are also a lot of places in which the black king has been excluded and some places that the white king has to be excluded.

```
       Weights: 915 Vote: For
[ 0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0 -1  0  0 -1  0  0  0]
[ 0  0  0  0  0  0  0  0 -1  0 -1]
[ 0  0  0  0  0  0  0  0  0  0  0]
[-1  0  0  0  0  0  0  0  0  0  0]

[ 0  0 -1 -1  0  0  0 -1  0  0  0]
[ 0 -1  0  0 -1  0  0 -1  0  0  0]
[-1  0  0 -1  0  0  0  0  0  0  0]
[ 0  0  0  0 -1  0  0  0  0  0  0]
[ 0  0 -1  0  0  0  0 -1  0  0 -1]

[ 0  0  0  0  0  0  0 -1  0  0  0]
[ 0  0  0  0 -1 -1  0  0  0  0  0]
[ 0  0 -1 -1  0  0  0  0  0  0  0]
[ 0  0  0 -1  0  0  0  0  0  0  0]
[ 0  0 -1  0  0  0 -1  0  0  0  0]

[ 0  0  0  0 -1  0  0  0  0  0 -1]
[ 0  0  0 -1 -1  0  0  0  0  0 -1]
[ 1  0  0  0  0 -1  0  0  0  0  0]
[ 0  0 -1  0 -1  0  0  0 -1  0 -1]
[ 0  0  0  0  0  0  0 -1  0 -1  0]

[-1  0 -1  0 -1  0  0  0  0  0 -1]
[ 0  0  0  0  0  0  0  0  0  0 -1]
[ 0  0  0  0 -1  0 -1  0 -1  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0]
[ 0  0 -1  0  0  0  0 -1  0 -1  0]
```

Figure 5.13: 5x5 Clause trained on 4 piece voting for win with 915 weight

Figure 5.12 shows a clause gotten from a Moving player Tsetlin Machine trained on the 4 piece dataset with a window of 5x5. This clause votes for a win for the black side. Compared to the one that votes for white, this clause has a lot less exclude bits set, but still has a lot of positions where the black and white king cannot be.

Because of all the excluded bits being the majority of bits set in the clauses one can say that the clauses looks more at what should not be there, in order for it to vote for a win. Which can overall be easier in chess, since the amount of different pieces that would have to be there to get a win, check mate, could be even more. Such as if the pattern is located somewhere in the middle of the board, in which one would need pieces all around if one is to consider it a win. There would also be a need for pieces outside the pattern, which is not known to the clause. Though this can be helped with the combination of other patterns.

From the top clauses gotten there was many of them that was invalid, either having more than one piece on a square or having the same piece both be there and not be there at the same time. Other than these the clauses mostly looked as the examples shown in this section, having one or two piece set to be included and many set to be excluded. From this its hard to interpret of it detects a specific pattern with these clauses.

5.4 Playing Games

5.4.1 Players

Game (FEN)	Player	Amount of games	Reference
8/8/8/4QK2/ 8/2nkp3/8/8 w - - 0 56	Flipped player 3	10	1
8/8/8/4QK2/ 8/2nkp3/8/8 w - - 0 56	Flipped player 4	10	2
8/8/8/4QK2/ 8/2nkp3/8/8 w - - 0 56	Moving player 4	10	3

Table 5.11: The players that will play in a tournament

Table 5.11 shows the different methods chosen and the board or game used for the evaluation. The Player column in the table is the method. The player name refers to which method and also which dataset that is used for the method.

The game used for the evaluation would have white to win with perfect play. Though it is possible for black to win or draw if white makes the wrong moves. Where there are multiple moves that leads to black winning and one move which leads to draw. The reason for the draw would be a lack of materials, meaning that black and white have given away enough of their pieces such that none of them can be check mated.
Once the method picks the draw move. The opponent has to pick a losing move for the method to regain its advantage, or they will have to keep choosing a drawing move to not lose.

5.4.2 Results

Table 5.12 is for the confidence scoring, with depth of 1.

Reference	Opponent	Win	Loss	Draw
1	Random	0	0	20
1	Monte Carlo	0	0	20
1	Flipped player 4	0	0	10
1	Moving player 4	0	0	10
2	Random	1	2	17
2	Monte Carlo	0	0	20
2	Flipped player 3	0	0	10
2	Moving player 4	0	0	10
3	Random	1	1	18
3	Monte Carlo	0	0	20
3	Flipped player 3	0	0	10
3	Flipped player 4	0	0	10

Table 5.12: Results from having the various methods play each other

Table 5.12 shows that the players only manage to draw against each other and against the Monte Carlo method. Where two of them show different results against random. In which one can assume that looking at the moves 1 search depth is not enough for the Tsetlin Machine to make a proper move. The results also show that the different methods manage to make the 1 move that leads to a draw, but still to keep the draw. This means that the methods most likely see the possibility for a draw and predict it as a win, with high confidence. Where this could be happening because the training data was too small in order to properly learn how to play this situation.

Table 5.13 is for the confidence scoring, with depth of 3.

Reference	Opponent	Win	Loss	Draw
1	Random	1	1	18
1	Monte Carlo	0	0	20
1	Flipped player 4	0	0	10
1	Moving player 4	0	0	10
2	Random	2	1	17
2	Monte Carlo	0	0	20
2	Flipped player 3	0	0	10
2	Moving player 4	0	0	10
3	Random	2	0	18
3	Monte Carlo	0	0	20
3	Flipped player 3	0	0	10
3	Flipped player 4	0	0	10

Table 5.13: Results from having the various methods play each other

Table 5.13 shows that giving the machine moves further down doesn't change the results much. The biggest change is that all the methods have different results against the random opponent. Though all the others are still draws. Which just supports the fact that the methods choose one or multiple draw moves instead of a winning move, so that it has no way of winning.

Table 5.14 is for the non confidence scoring, with depth of 1.

Reference	Opponent	Win	Loss	Draw
1	Random	2	2	16
1	Monte Carlo	0	0	20
1	Flipped player 4	0	0	10
1	Moving player 4	0	0	10
2	Random	3	5	12
2	Monte Carlo	0	0	20
2	Flipped player 3	0	0	10
2	Moving player 4	0	0	10
3	Random	3	3	14
3	Monte Carlo	0	0	20
3	Flipped player 3	0	0	10
3	Flipped player 4	0	0	10

Table 5.14: Results from having the various methods play each other

Table 5.14 shows that the methods manage to get more differing results against random, but there are still mostly draws.

Table 5.15 is for the non confidence scoring, with depth of 3.

Reference	Opponent	Win	Loss	Draw
1	Random	5	2	13
1	Monte Carlo	0	0	20
1	Flipped player 4	0	0	10
1	Moving player 4	0	0	10
2	Random	7	8	5
2	Monte Carlo	0	0	20
2	Flipped player 3	0	0	10
2	Moving player 4	0	0	10
3	Random	6	5	9
3	Monte Carlo	0	0	20
3	Flipped player 3	0	0	10
3	Flipped player 4	0	0	10

Table 5.15: Results from having the various methods play each other

Table 5.15 shows again that even given a greater depth to the tree search, it only produces differing outcomes when against the random opponent

5.4.3 Comparison

Though the various methods showed good results when testing them. Though it is not shown in the tables all wins and losses happened outside the specified amount of moves, and all draws happened within the specified amount of moves. Which means that the Tsetlin Machine plays itself to draw against all other methods, except for random. Which could have several explanations:

That the Tsetlin Machine thinks a position or move that actually leads to a draw, leads to a win. Which could explain the wins and losses towards random, since the random doesn't try to make optimal moves. And since the position it thought was draw is not anymore, and random has given it the advantage back, it manages to retain the win advantage. Where Monte

Carlo will see the draw as a possibility to get a better result than a loss, and will therefore not give the Tsetlin Machine a chance to regain the advantage. Though the Tsetlin Machine also losses. Making it seem like the Tsetlin Machine has a classification problem. Where it has not trained on enough positions in order to know that it is actually a draw. From the tree search evaluation, we can also see that it is not perfect in its precision either, where this could show how they miss-classified the draw as a win.

Another explanation would be that the Tsetlin Machine does not train well enough on playing the game. The Tsetlin Machine is good at classification, but even when it classifies a winning board correctly, the machine has not been trained on knowing whether this board is closer to the win or not, just that with perfect play from that point on would lead to a win. The data does not contain boards that show check mate positions, so it would probably not choose these. This is shown from the fact that instead of winning the game, both sides loses their non king pieces instead. Check mate is often a hard board to achieve with only 1 piece other than the king, so this possibly had an impact on the ability of the Tsetlin Machine to reach it.

Chapter 6

Conclusion

6.1 Goals

This projects goals managed to be reach different levels of completion. Following we take a look at the each goal and how well it was achieved.

Goal 1: Train a Tsetlin Machine to play a chess endgame through a tree-search. Seeing if the moves will result in win, loss or a draw, and analyze if it can play or not.

The Tsetlin Machine was able to correctly predict a lot in the tree-search tests and looked promising, but when playing it showed poor results by getting mostly draws. The wins and losses it had was not natural, but forced by exceeding the amount of moves it ran for.

So from this the project can conclude, from what our tests showed, that the Tsetlin Machine is able to identify if the board would lead to a win, loss or draw from perfect play. Even though it is not able to play the game well.

Goal 2: Test different types of Tsetlin Machines; Multiclass, Convolutional and weighted, and compare the results.

This goal was reached by testing the four different types of machines being non weighted Multiclass, weighted Multiclass, non weighted Convolutional and weighted Convolutional.

The results showed that the weighted and non weighted performed similar to each other. Where the Convolutional approaches outperformed the normal Multiclass implementation when the window size was lower than 8x8, which is the full board. Multiclass reaches about 45% on all the different datasets, and for weighted and unweighted. While the Convolutional approach reaches around 55% on average for the different piece amounts. The results also showed that these implementations performed pretty poorly, since it was trained on data that focused on both players at the same time.

When comparing the weighted version versus non weighted, it was shown that it did not provide a significant impact on the accuracy. This could have been because of poor optimization of various factors, such as hyperparameters or maybe by not training the Tsetlin Machine versions enough. Though the weighted versions are supposed to be faster and reduce the amount of clauses needed in order to achieve the same percentages.

Goal 3: Test different methods for splitting the data and setting up the Tsetlin Machine; change the player and split by moving player, and Split results.

This goal was achieved by making the tree different methods of flipped player, moving player and split results. The flipped player and moving player methods both performed well for analyzing the board. Where the two methods achieved about the same results. The best results were around 90% accuracy, which was achieved with the 3 piece dataset and a small window. This are the best results gotten if you do not look at the tests that excluded draw from the data that performed better. As with the other Convolutional approach, they performed better with lower window sizes. The split result method on the other hand showed poor results by having a lower score than the other methods.

This shows that the implementations that focused on one player outperformed the other methods. In this project the goal was looking at multiple different variants, so there was limited time to focus and perfect the parameters and data for one method. So focusing on spending more time improving one method is work that could be done in the future.

Goal 4: Visualisation of clauses.

For this goal the clauses was visualised as wanted, but the clauses was not as understandable as hoped for, because chess is very complex. Looking at

one clause at a time was not enough to get interpretable patterns. To create patterns that would be more sensible, one would probably need to look at more clauses at the same time and see if they work together.

There was also a problem of not being able to have a better representation of a clause, since the squares could contain many different pieces meant one would need to have a better system for representing pieces to include or exclude. This increased the difficulty in making the clauses visually easy to understand.

Goal 5: Testing other machine learning algorithms and comparing the Tsetlin Machine to them.

Even though we had a goal of testing other machine learning algorithms and testing them against the Tsetlin Machine, there was only enough time to test Monte Carlo Tree Search against the Tsetlin Machine. Though Monte Carlo Tree Search is not a machine learning algorithm, it is a smart method that achieves pretty good results. Monte Carlo Tree Search is also used to generate datasets for machine learning algorithms or used along with machine learning algorithms in order to achieve good results for games.

Final remarks would be that even though this project did not yield any significant results, in playing the endgame of chess with the Tsetlin Machine. It did lay some ground work in what methods works better than others for this task. Which would help with future work on this project to be able to focus on what was found to work well. The playing was also tested out and found that the pure Tsetlin Machine was not able to lead to wins when trained on the data it had. Where some ideas to improve this would be reinforcement learning, or changing the dataset to better reflect playing an entire endgame perfectly. These ideas are written more about in the future work section.

6.2 Future Work

Given the results that the Tsetlin Machine seems to consistently draw against other techniques. Future work could be to create or find a larger dataset, which contains more cases. To see if one can increase the accuracy of the Tsetlin Machine and to get it to win more games.
Another option would be to do further testing of the dataset that contains all of the pieces, and try to achieve better results with it. Since this dataset

would contain more information for various endgame scenarios than a more specialised dataset.

To decrease the amount of draws gotten when playing the game. The dataset could be changed from just adding one board per game, to instead add the entire endgame with perfect play of the board found. This would lead to the machine being trained on data that would lead to a win, and not just positions that would be a win given perfect play.

Another way of doing this would be to implement Reinforcement Learning or another method of self play. Where the machine plays against itself, or generate and evaluate boards for endgames. This would allow the Tsetlin Machine to learn board positions that it does not have from the other dataset. One would be able to not have to rely on a self generated dataset that does not contain enough varied cases. With the self play option one would also try to learn the Tsetlin Machine to play games from the beginning.

An interesting idea to test out for endgame chess would be to have different machines trained on different amount of pieces left on the board. Implementing overhead for the playing so that it could swap what machine it used based on the amount of pieces left on the board. For example; if there was 4 pieces left it would use a machine trained on the 4 piece data, but if it was 3 pieces it would instead use a machine trained on the 3 piece data.

Other future work would be to increase the amount of pieces left on the board that it can handle. 6 pieces would be achievable with the current hardware, but would prove troublesome if one wanted to include the 7 piece data. It would either require a hard drive with a lot of space, or some online query option. This would be interesting in order to see how much the increased complexity of adding more pieces could change the results. Looking at how the 5 piece data performed worse than the lower piece amounts, the 6 and 7 piece would probably follow this. Though this can be combined with the previous suggestion of adding the data together to one set, and trying to increase the accuracy for that set.

An approach to look at clauses less individually, as this did not yield the desired results, would be to implement some way of gathering different clauses that vote together on a board. In order to see if one can get a better pattern, or a pattern that contains more information than looking at one clause at a time. Then we can see if we can get a better understanding for the choices,

since it will show what the machine mostly reacts too. One can also make a system where the features that where included or excluded are gathered in a separate pattern, in order to better see what specifically lead to the classification.

For most of the future work mentioned, it would be to chose one method. Either flipped player or moving player and focus on it by testing out different hyper-parameters for this method specifically. Since the hyper-parameters was made at the beginning of the project and used for every method. So the methods could perform better with different hyper-parameters and would be worth looking into.

References

[1] 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.

[2] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

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

[5] chess.com. `https://www.chess.com/no/analysis`, Accessed: 19. May 2020.

[6] Julian Schrittwieser David Silver, Thomas Hubert. A general reinforcement learning algorithm that masters chess, shogi and go through self-play. `https://kstatic.googleusercontent.com/files/2f51b2a749a284c2e2dfa13911da965f4855092a179469aedd15fbe4efe8f8cbf9c515ef8` Accessed: 10. May 2020, 2018.

[7] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240, 2006.

[8] Deepmind. Openspiel code. `https://github.com/deepmind/open_spiel`, Accessed: 05. April 2020.

[9] Steven J. Edwards and readers of rec.games.chess. Portable game notation specification and implementation guide. `https://www.thechessdrum.net/PGN_Reference.txt`, Accessed: 19. May 2020, 1994.

[10] Dumitru Erhan, Y. Bengio, Aaron Courville, and Pascal Vincent. Visualizing higher-layer features of a deep network. *Technical Report, Univeristé de Montréal*, 01 2009.

[11] Niklas Fiekas. Pgn parsing and writing. `https://python-chess.readthedocs.io/en/v0.8.2/pgn.html`, Accessed: 09. March 2020.

[12] Niklas Fiekas. python-chess: a pure python chess library. `https://python-chess.readthedocs.io/en/latest/`, Accessed: 02. March 2020.

[13] Niklas Fiekas. Syzygy endgame tablebase probing. `https://python-chess.readthedocs.io/en/v0.8.2/syzygy.html`, Accessed: 09. March 2020.

[14] 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.

[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] Ami Hauptman and Moshe Sipper. Gp-endchess: Using genetic programming to evolve chess endgame players. In *European Conference on Genetic Programming*, pages 120–131. Springer, 2005.

[17] Barbara Christine Hoekenga. *Mind over machine: what Deep Blue taught us about chess, artificial intelligence, and the human spirit*. PhD thesis, Massachusetts Institute of Technology, 2007.

[18] IchessU. Chess rules for kids & beginners. `http://www.chesscoachonline.com/chess-articles/chess-rules`, Accessed: 19. May 2020.

[19] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[20] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019.

[21] lichess.org. lichess.org game database. `https://database.lichess.org/`, Accessed: 17. March 2020.

[22] lichess.org. Syzygy endgame tablebases. `https://syzygy-tables.info/`, Accessed: 09. March 2020.

[23] myChess.de. Laws of chess - the moves of the pieces. `http://www.mychess.de/ChessMoves.htm`, Accessed: 19. May 2020.

[24] myChess.de. Laws of chess - the notation. `http://www.mychess.de/ChessNotation.htm`, Accessed: 19. May 2020.

[25] Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *Distill*, 2020. https://distill.pub/2020/circuits/zoom-in.

[26] rin. How stockfish works: An evaluation of the databases behind the top open-source chess engine. `http://rin.io/chess-engine/`, Accessed: 21. May 2020, 2014.

[27] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

Appendices

A Code

Code available at https://github.com/jareie/TMChess.

UiA University of Agder
ICT seminar III final project
Faculty of Engineering and Science
Department of ICT