Universiteit Leiden

Master Computer Science

MCTS Tetris: an approach to solving the gamestate-reachability problem

Name:	David Lin
Student ID:	s2208407
Date:	21/01/2025
Specialisation:	Foundations of Computing
1st supervisor:	Mike Preuss
2nd supervisor:	Giulio Barbero

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Tetris (1985) is a game where 4-tiled tetrominos are placed in a 2-dimensional 10 by 20 playing field, with the goal of covering entire rows and clearing these rows to earn points. We approach the game with a new gamestate-reachability problem, where we attempt to reach a predetermined gamestate by the placements of tetrominos, starting with an empty board. An attempt to automate the search for a solution to this problem is done using Monte Carlo Tree Search (MCTS), where the reward function is implemented using a Convolutional Neural Network (CNN). The CNN shows a very stable trend regarding the train-test split on the generated dataset, but MCTS displayed seemingly random moves instead of our desired sequence leading to a final state. Improvements such as fixing the CNN scoring predictions, or even removing MCTS and using Reinforcement Learning (RL) instead might potentially fix the currently broken implementation.

1 Introduction

1.1 Tetris versions

Tetris is not one specific game, as there exist many different versions of the game. In fact, it is the video game with the world record number of ports [15]. The original game was made by Alexey Pajitnov and came out in 1985 [14], where the game was relatively bare-bones with only one visible piece in the queue, a simplified rotation system, and no existing hold system. One of the more revolutionary version releases of Tetris came with the release of Tetris DS in 2006 [17], which upgraded the Original Rotation System from the original Tetris to the Super Rotation System (SRS) [18], introduced the hold system where one piece could be saved or swapped for later use, and a longer visible queue of 6. SRS and its alternative version SRS+, which in addition to SRS allows 180-degree spins, are used in currently active competitive Tetris clones, namely Tetr.io and jstris [13, 7].

1.2 Used version

To simplify the transition between different game states, the Tetris environment used is based mainly on the original rotation system, where wall kicks are not allowed. We also do not allow tetromino placements where an overhang is filled.
To further simplify the problem, the algorithm will be able to choose its tetrominos. This means that the implementation of a hold system and visible queue would be redundant.

1.3 Gamestate-reachability: a new approach

The original Tetris games are high score based, where the objective is to fill in as many rows as possible. There are already existing projects that have functioning AI playing Tetris [10, 16]. However, the approach usually revolves around obtaining the highest score possible by clearing as many lines as possible whilst preventing a game over by keeping the board and tetromino placements as low as possible.
As an alternative approach, we introduce a new objective in which we want to reach a specified game state. The problem is defined as follows:

Definition 1.1 (Gamestate-reachability problem)**.** Given a partially filled Tetris board, what is the sequence of tetromino placements that is required to reach that board when starting with an empty board, whilst adhering to the rules of Tetris?

1.3.1 Intuitive solution

This problem can be solved easily if only the number of filled cells in the matrix is checked, and not the positions. Because every tetromino adds 4 tiles to the board and clearing a line by filling a row removes 10 lines, the matrix cannot transition between even number of tiles and odd number of tiles. It can only remain even if the previous state had an even number of filled tiles. This is also the case with states that have an odd number of filled tiles. However, since Tetris is usually played with an empty state at the start, reaching a specified game state with an odd number of tiles would require a modified odd-number-filled starting state.

1.3.2 Manually finding the sequence

Although finding a possible tetromino placement sequence seems very heuristic-dependent, there is actually a simple method to discover one of many possible sequences. We do this with the following strategy:

1. Start at the final state
2. Move backwards one step
    - Choose one of the following:
        (a) Remove a tetromino from the playing field
        (b) Add a line to any row on the playing field
3. Repeat the above step until the board is empty

The tetromino removal and line addition should aim for the following heuristics, but are not always limited by these:

- Remove tetrominos with higher tiles before lower ones
- Removal of tetrominos should not result in floating tiles
    - Prevent difficult situations.
- Remove pieces only if they have completely empty columns above the tetromino.
    - This emulates the tetromino placement in reverse.
- Only add lines if no tetromino removal is possible.
    - This prevents excessive line addition, which would unnecessarily lengthen the sequence.
- Remove the most tetrominos possible before a line addition is required.
    - Attempt to clear out the currently existing tiles as much as possible.

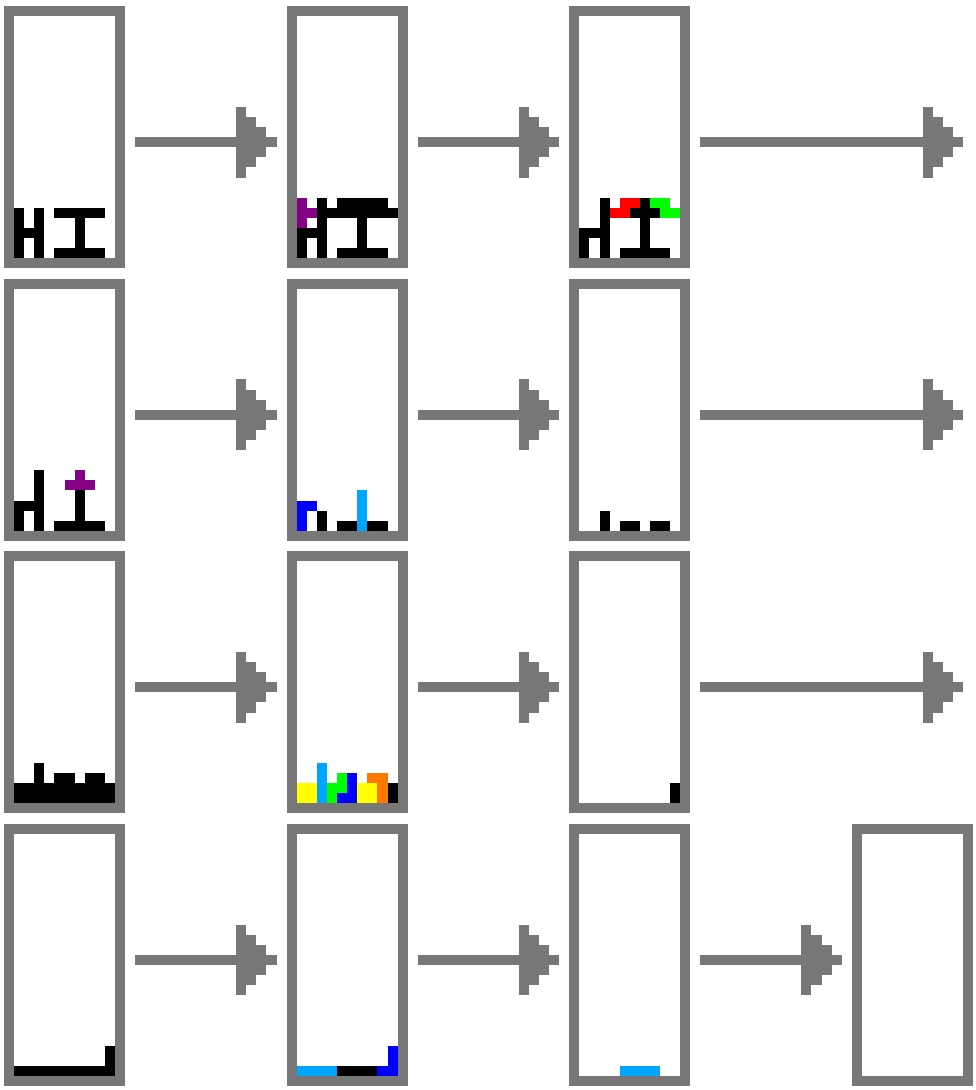


Figure 1: Reverse tetromino sequence example for the "HI" final state executing either piece removal or line addition

In Figure 1 we can see an example sequence that solves the gamestate-reachability problem on a 2D-matrix spelling out the word "HI". Note that the given example is only one of infinitely many sequences that would result in the same outcome. This can be explained with simple pattern-solving examples.

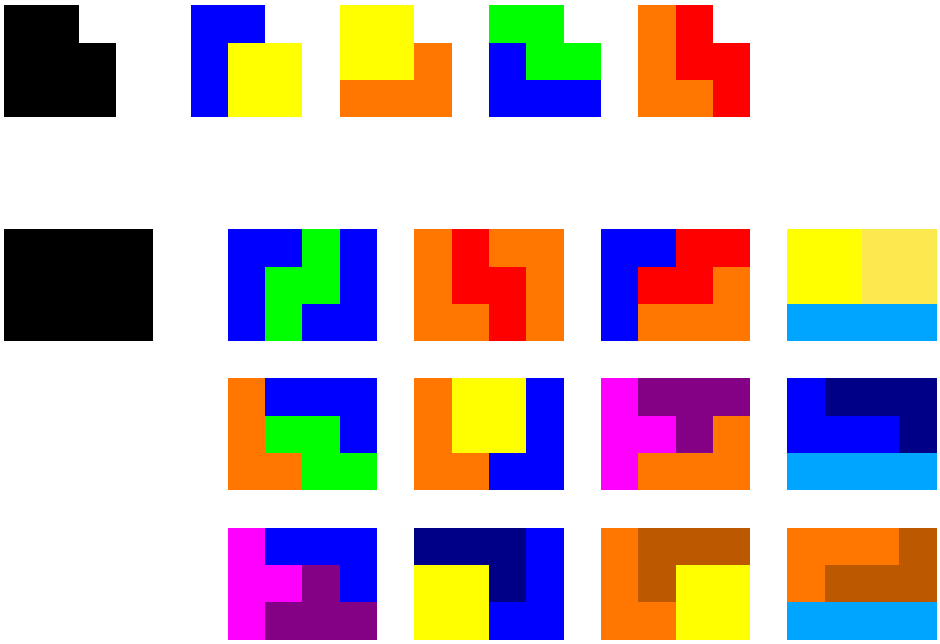


Figure 2: Two examples of methods how patterns can be filled in with tetrominos

As Figure 2 shows, certain patterns can be solved in many different ways. As these patterns become larger, the number of possible solutions increases in complexity. This means that for the given example sequence in Figure 1 certain patterns could be replaced with the ones given in Figure 2 and it would make no difference in the final outcome.

1.4 Using MCTS to solve the gamestate-reachability problem

There are many different frameworks for approaching the automatization of finding a valid solution to this problem. Some of the more prominent frameworks are as follows:

- Reinforcement Learning [8]
  - Deep Q-Learning [20, 12]
  - Policy-based RL [1]
- Monte Carlo Tree Search [2]
- Manual heuristical methods

There already exist different Reinforcement Learning approaches to Tetris [16, 10], though these both optimize in the regular high-score-based game. Reinforcement Learning has the potential to solve the gamestate-reachability problem, but the choice was made to work with MCTS, as at first glance the problem seems very solvable because of the example in Figure **??**. And with MCTS, the parameter interpretability is better compared to Reinforcement Learning, where hyper parameter tuning is mostly a trial-and-error process in its experimental setup.
For the scope of this paper, we are more interested in the process of automating the sequence finding, so we disregard the manual heuristical methods. However, it is still an interesting problem for the mathematical and pattern recognition fields.
The research questions this paper will address are defined as follows:

RQ1. How can an MCTS framework be set up for the gamestate-reachability problem?

RQ2. How can the gamestate-reachability problem be solved using an MCTS framework?

2 Background information

2.1 Complex problems

As paraphrased by Funke: "Complex problem solving (CPS) emerged in the last 30 years in Europe as a new part of the psychology of thinking and problem solving," and complex cognition is required to do complex problem solving [4]. Creating complex problems allows for new AI challenges to arise and create new insights with the solutions AI can provide. This is not exclusive to Tetris, though Tetris has its benefits when used for complex problems. Tetris has approaches on being solved by AI [16, 10]. The gamestate-reachability problem is a complex approach to the base game of Tetris allowing for different insights into how the game can be played.

2.1.1 Tetris as a framework

The game can be very engaging as the entry level for beginners is very manageable, yet the game can be played in many ways, allowing for advanced players to think of strategies that allow for efficient and consistent board management to obtain the most points.
Another advantage is that, as can be seen from Figure 2, there are many solutions to a single problem, which makes the solution more approachable for our MCTS searching algorithm.

2.2 Monte Carlo Tree Search

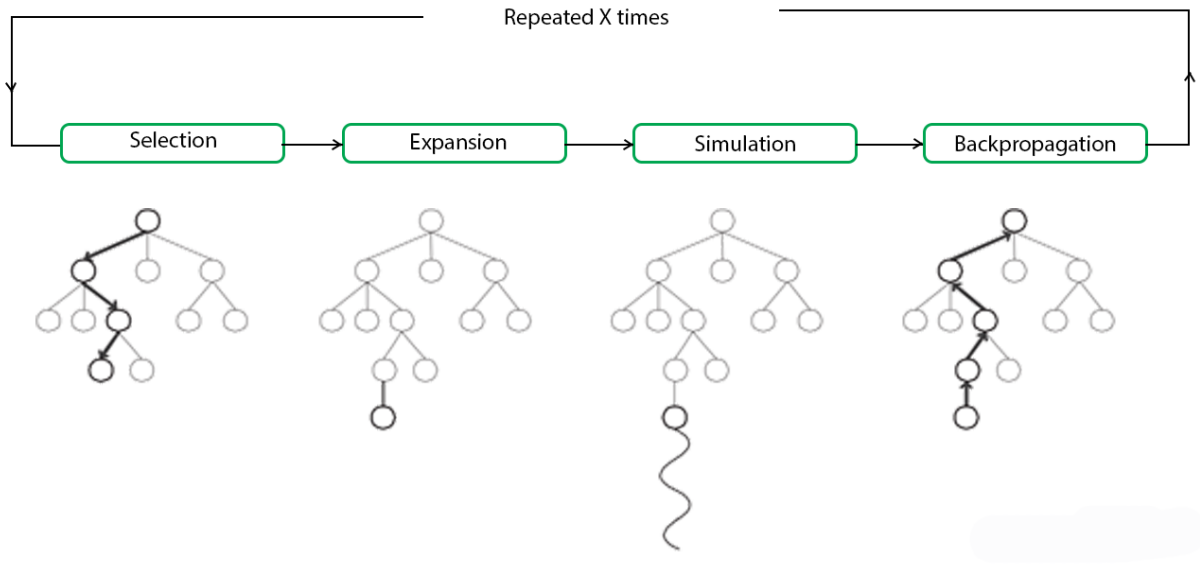


Figure 3: Monte Carlo Tree Search algorithm diagram [5]

MCTS is a widely known binary search tree algorithm [2] used in games to find a winning state by chance during random playouts. To summarize, the MCTS algorithm used in this project is divided into four main steps: selection, expansion, simulation, and backpropagation (see Figure 3).
The selection process is done using the UCT calculation [2], which determines which node should be expanded next. The calculation is as follows:

$$UCT(node) = \frac{wins_{node}}{visits_{node}} + c\sqrt{\frac{\ln visits_{node.parent}}{visits_{node}}}$$

The arbitrary choice was made to set c as $\sqrt{2}$ for this project. As for the priority of selection, experiments were done on both the regular UCT calculation and a combination of UCT and breadth-first selection, where every move from the root must be selected first before being able to select the children of those moves.
The expansion process simply selects a random move from the selected node, a node is created representing that move, the created node becomes connected to the selected node as a child, and the selected node becomes connected as the parent of the created node.
The simulation process takes the state of the expanded node as the starting point and starts to repeatedly do random moves until either the game state is an ending state or the maximum number of possible moves from the root up to that state has been done. Once it has reached either condition, the score of that state is saved, the state is reset to the expanded node, and the process is repeated until a certain number of simulations have been done. After all simulations are finished, all the saved scores are summed up and divided by the number of simulations. This value will be saved as the final score of the node.
The purpose of the model training in the previous step is to determine the score at the end of each simulation. Instead of using a win/lose scoring system, we use a linear scaling method, as explained in Section 3. To distinguish the final state from other states, we alter the scoring system by dividing the score by 100 when it is not a final state. With this changes, a score of 100 represents the final state, 0.99 represents a state 1 move away from the final state, 0.98 represents a state 2 moves away, etc. Because the max_moves for these experiments is set to 20, the score will never exceed 20 when a final state has not been found, so it can easily be distinguished from scores that are 100 or above.
Finally, the backpropagation process takes the score of the current node, and adds this score to the score of every parent up until the root of the whole tree. Also, for every node it encounters in this process, it adds one to the number of visits of those nodes. Essentially, an expanded node always has its visits set to 1, the number of visits in the root is the number of simulations done, and for every node, the number of visits entails the number of leaves that node subtree has including itself and its children.

2.3 Convolutional Neural Network

The general idea behind a Convolutional Neural Network (CNN) [9] is that it takes an n-dimensional input and outputs either one or multiple values. When looking at Figure 4, there is an input layer on the left, multiple hidden layers in the middle, and an output layer on the right. Every layer contains nodes, which take a certain number of inputs and can output a singular value to nodes in the next layer. To not go into too much depth, the output depends on its weights, biases, and activation functions. The nodes of the CNN layers are initialized with random values. The CNN can be trained by inputting training samples in the input layer,

comparing the output layer values with the actual ground truth values of the sample, and adjusting the weights in each node in the CNN layers.

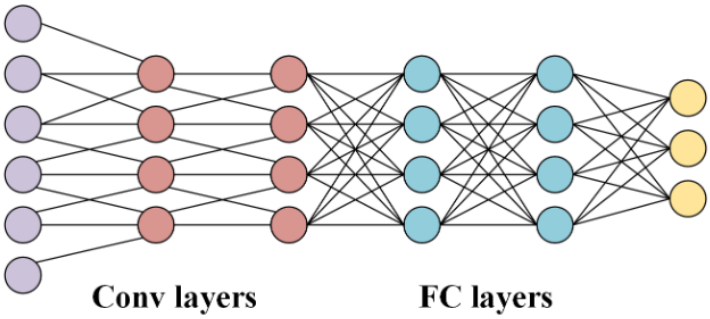


Figure 4: Diagram of CNN layers and FC layers, from [9]

In this project, we use a binary 2D matrix as input, and thus Figure 5 gives a better idea of the actual implementation. Our CNN implementation converges into one singular value as output, but it still roughly summarizes the functionality.

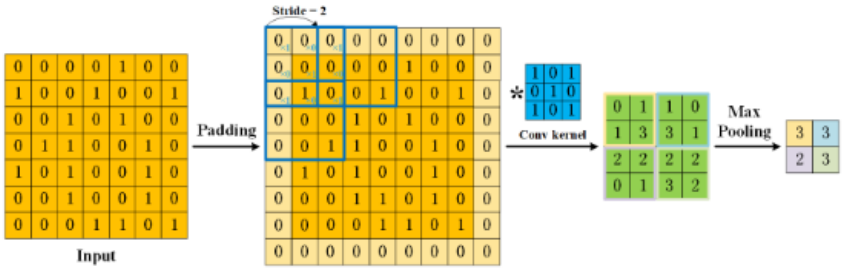


Figure 5: Procedure of a 2D CNN, from [9]

3 Approach

In this section, we discuss the framework of the project, outlining specific key design choices and the reasoning behind them.

3.1 Tetris environment

To optimize the time efficiency for this project, we use an existing Tetris framework, namely the Tetris-AI repository by Nuno-faria [3]. The Tetris-AI repository has a working DQN model that can achieve a theoretically infinite score with the method it plays the game. Because the gamestate-reachability problem results in board evaluations different from those of conventional Tetris based on high scores, the DQN will unfortunately be discarded. We do use the framework from the game, as it follows the rules of the game and states and actions can be accessed easily.

3.2 MCTS vs Reinforcement Learning

In Section1.4, we discussed different approach methods, where the manual heuristics were disregarded. As for the choice between Reinforcement Learning (RL) and Monte Carlo Tree Search (MCTS), there are advantages and disadvantages to both. RL has the highest potential of learning the objective with high consistency, while MCTS is much simpler in its framework and has less complexity in the learning process. However, the main drawback of using RL over MCTS is the interpretability of the parameters. MCTS has parameters that have much more clarity over their respective functions. RL on the other hand, which has many parameters that have no direct feedback on the performance of the model, is much harder to fine-tune to the task and would require much more investment through the process of hyperparameter tuning and learning curves.

3.3 MCTS CNN reward function

When initialising MCTS, it is provided with a specified desired final game state. Whilst running MCTS, it is continuously given game states in which the actual testing is taking place. These game states are then scored based on the number of moves the current game state is away from the desired final game state.
Even though manually calculating the mathematical formula has many benefits once it is successful, the scope of this paper lies more within the training of models rather than mathematical derivations, and therefore a CNN was trained and used for this project.
Besides the CNN, we also train a linear regressor, random forest model, and a k-nearest-neighbor model. These are used for comparison to see the main benefits of one superior model compared to the other models.

4 Method

We divide the experiment into subtasks:

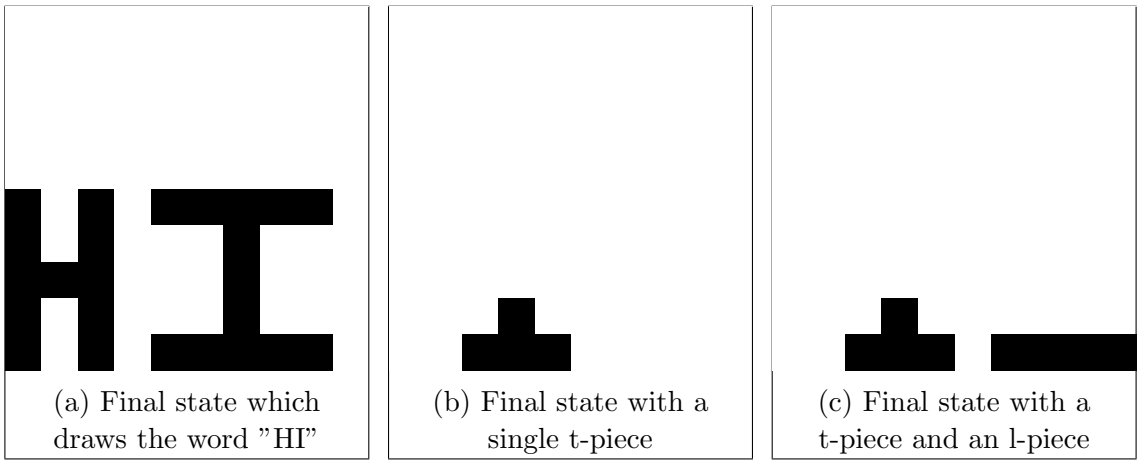
1. CNN model training
    (a) Sample generation
    (b) Training CNN + performance comparison
2. MCTS experimentation

4.1 Preparation step

Before we can run the experiments, we must first determine a final state which we want to generate a tetromino placement sequence toward. We run the experiments with three different final states:

1. A final state which spells out the word "HI" (Figure 6a)
2. A final state with only one t-piece (Figure 6b)
3. A final state with one t-piece and one l-piece (Figure 6c)

The main challenge will be state spelling out "HI". The other two states are to benchmark the performance on a smaller scale. In the case the main experiment fails, we can check if the smaller experiments are successful or if they also fail.



(a) Final state which draws the word "HI"

(b) Final state with a single t-piece

(c) Final state with a t-piece and an l-piece

Figure 6: Final states which the experiments are run with

4.2 CNN model training

4.2.1 Sample generation

The purpose of the CNN model in MCTS is to calculate the reward of any current state based on the predetermined final state. The training of this model requires task-specific samples for our task.
These can simply be generated with the following method:

1. Initialize at the final state
2. Do n backwards moves from the final state, where $n = random(1, \ldots, 20)$, and backwards moves as described in Step 2 of Section 1.3.2
3. Save the current state as a sample with label "sample_$\{m\}$_score_$\{n\}$.png", where m is to index every sample
4. Repeat until satisfied with the number of samples generated

To prevent the state from adding a line for every possible move, we define a probability for adding a line. We generate a dataset with a 20% chance and a dataset with a 40% chance of adding a line for performance comparison.
Before training the models, we experiment with a third dataset, where we introduce the sample generation from the starting state instead of the final state. These are generated as follows:

1. Initialize at the starting state
2. Do p forwards moves from the starting state, where $p = random(1, \ldots, 20)$, and forwards moves being the regular moves in Tetris.

3. Save the current state as a sample with label "sample_$\{m\}$_score_$\{q\}$.png", where m is to index every sample, and $q = l - p$, with l being an arbitrary large number. For this experiment, $l = 100$

When running the sample generation for the third dataset, there is a 50% chance to either generate using the first backwards method, and a 50% chance to generate forward samples with the second method.
Before using the three datasets for the model training experiments, we check the uniformity of each dataset and add duplicate data entries in the cases of skewed datasets.

4.2.2 Model training

We run two model training experiments. The first experiment is the performance of different models on dataset A. We run the first model training on a linear regressor, a random forest regressor (RF), a k-nearest neighbor regressor (KNN), and a convolutional neural network (CNN).
The second experiment is training separate CNN models on each of the three generated datasets and comparing the performance between the three models.
We compare the performance of the models by displaying the results of a 1-fold cross-validation test in a box-plotted graph, where the train-test split is set to 75%-25%. The graph includes a straight line indicating the ground truth. The x-axis displays the truth score of the input game state, and the y-axis shows the predicted value of those game states. The main objective behind this test is to get a grasp of the general performance of each model without tuning the models. The performance could be improved through hyperparameter tuning, but for this project, this is less of importance, as can be concluded from the upcoming sections.

4.3 MCTS experimentation

The pseudocode of the MCTS implementation can be found in Algorithm 1. The used algorithms 2 to 4 can be found in Appendix 9.1

4.3.1 Validation

To validate the MCTS implementation, the algorithm was applied to Tic-Tac-Toe. Because the game has a small gamestate space, it is feasible to manually check the validity. Various existing MCTS Tic-Tac-Toe implementations [19, 6] show a breadth-first approach to the selection process. This brought up the idea of experimenting with different selection processes.

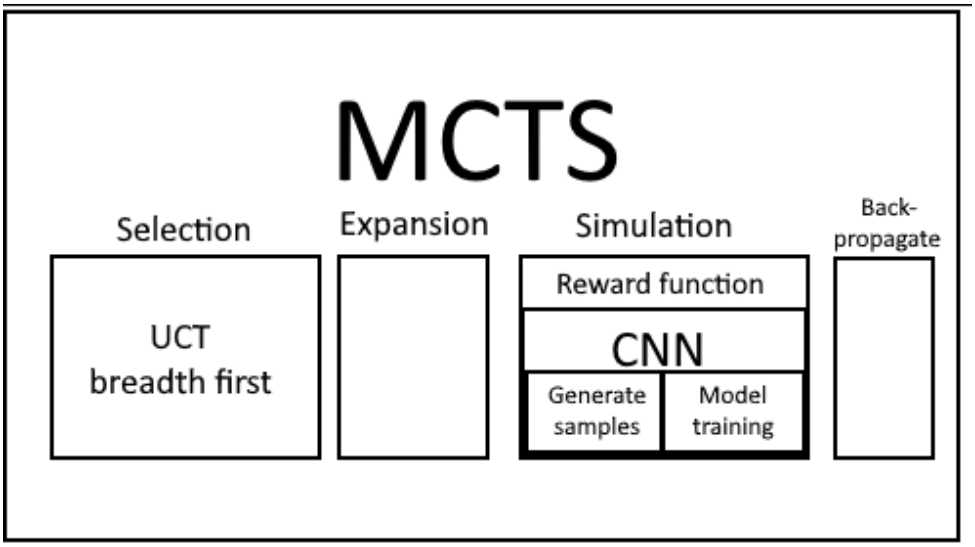


Figure 7: Monte Carlo Tree Search project overview

Algorithm 1 MCTS Tetris implementation

```
procedure MCTS_Tetris
    n ← 0
    max_steps ← 20
    mcts_iterations ← 10000
    root ← Node
    current_node ← root
    while game not ended and n < max_steps do
            ▷ Run MCTS until final state reached or no tetromino placements possible
        i ← 0
        while i < mcts_iterations do
            selected_node ← select(current_node)
                                                          ▷ See Algorithm 2
            if selected_node exists then
                if selected_node has next states then
                    expanded_node, value ← expand_and_simulate(selected_node)
                                                          ▷ See Algorithm 3
                    backpropagate(expanded_node, value)   ▷ See Algorithm 4
                else
                    Return root
                end if
            end if
            i ← i + 1
        end while
        best_move ← Node from current_node.children with the highest wins/visit
        score
        current_node ← best_move                  ▷ Repeat MCTS with a new "root"
        n ← n + 1
    end while
end procedure
```

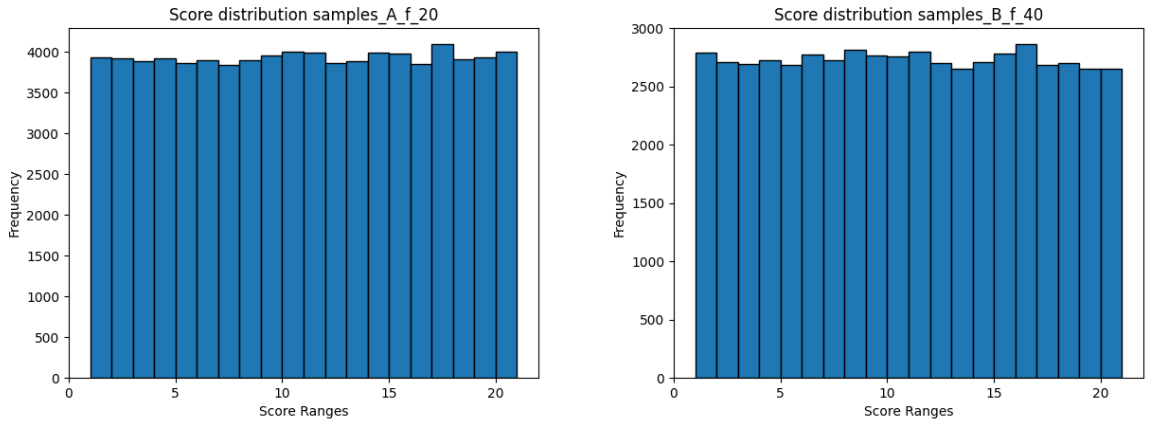
5 Experiments & Results

All hardware and software specifications can be found in Appendix sections 9.2 and 9.3. The source code is available on Github[1]

5.1 Sample generation

The three datasets that we generated for training are the following: A. Backwards generated samples (20% chance), B. Backwards generated samples (40% chance), and C. Forwards + backwards generated samples (20% chance). To check if the datasets are skewed, we generate a histogram, where the x-axis displays the score of the samples and the y-axis displays the sample count of that score.
As for dataset C, because the scores range from 0-40 and 65-99, we split the data projection into two histograms to increase clarity.
With the distribution fix in the case of skewed data, we duplicate data up to a margin of 90% of the largest count of a single score in the dataset.

5.1.1 Results



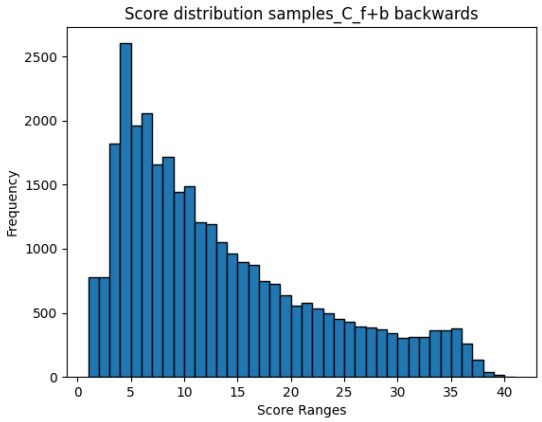
(a) Histogram of dataset A (backwards samples, 20% line chance)

(b) Histogram of dataset B (backwards samples, 40% line chance)

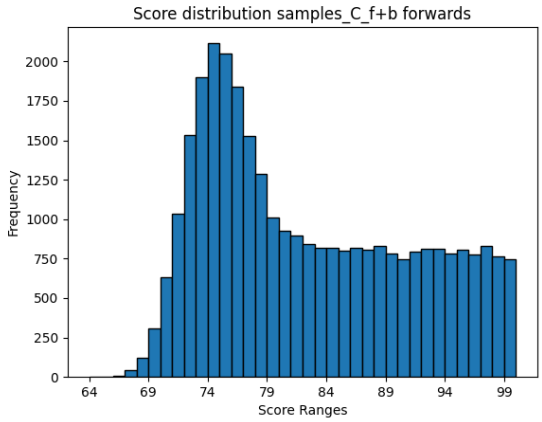
Figure 8: Histograms of the sample generation of datasets A & B

Datasets A and B display a very even distribution in Figures 8a and 8b. However, dataset C ended up very skewed towards the lower values, as can be seen in Figures 17a and 17b, so the distribution of dataset C is fixed with the method explained earlier.

[1] `https://github.com/davidlinye/TetrisAI`

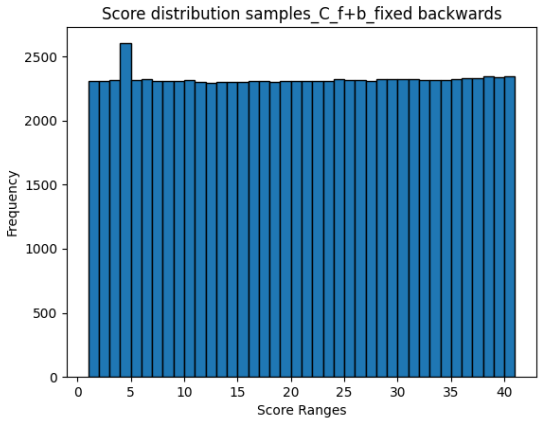


(a) Histogram of dataset C backwards (forwards + backwards samples, 20% line chance)

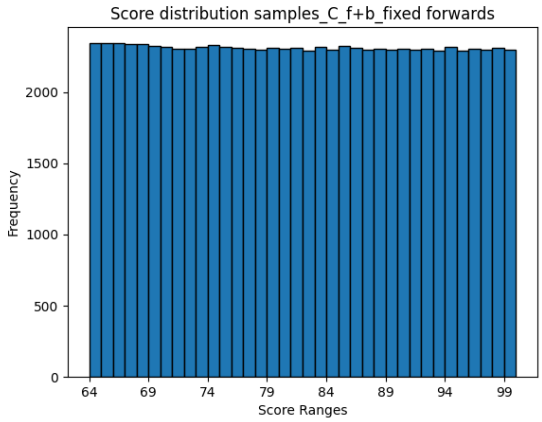


(b) Histogram of dataset C forwards (forwards + backwards samples, 40% line chance)

Figure 9: Histograms of the sample generation of dataset C



(a) Histogram of dataset C backwards (forwards + backwards samples, 20% line chance)



(b) Histogram of dataset C forwards (forwards + backwards samples, 40% line chance)

Figure 10: Histograms of the sample generation with fixed distribution of dataset C

5.2 Model training

5.2.1 Different models experiment

The first experiment is the training and performance comparison of the four different models trained on dataset A.

The boxplots displayed in Figures figs. 11 to 14 show on the x-axis the true value of a game state, and on the y-axis is the value of the game state predicted by the model. The red regression line is the ground truth, as the predicted value matches the truth values on that line, and the boxplot should show values as close to the regression line as possible.

Because the problem is not linear, the linear regressor is expected to break as can be seen in Figure 11, and thus, we disregard this model immediately.

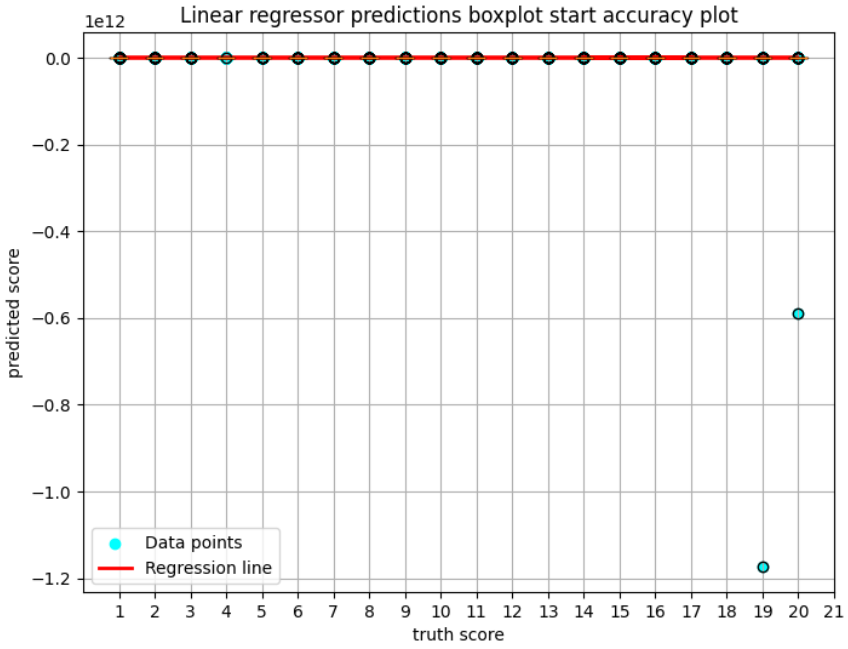


Figure 11: Boxplot linear regressor trained on dataset A (20% line chance backwards generated samples) run with 1-fold cross-validation 75%-25% train-test split

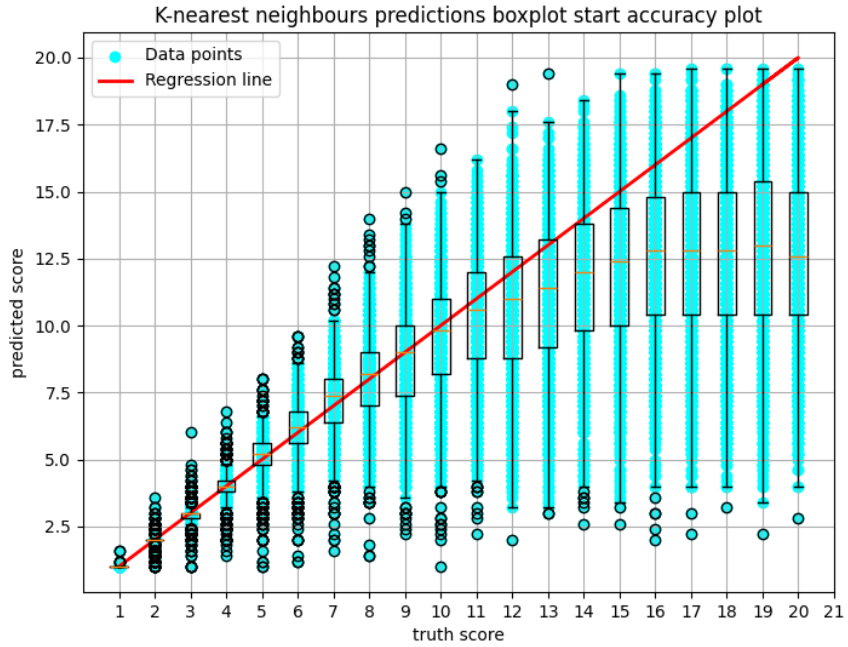


Figure 12: Boxplot k-nearest neighbor model trained on dataset A (20% line chance backwards generated samples) run with 1-fold cross-validation 75%-25% train-test split

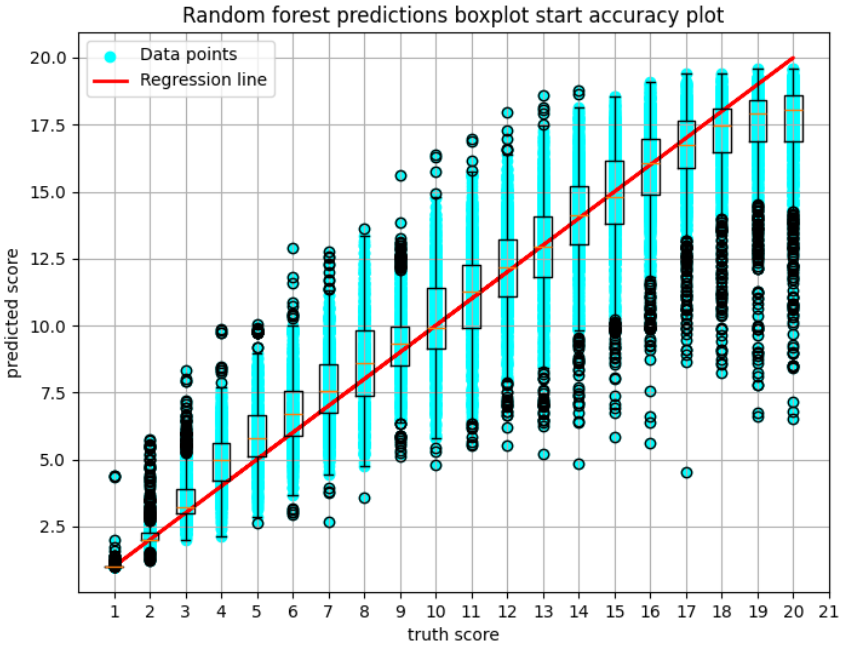


Figure 13: Boxplot random forest model trained on dataset A (20% line chance backwards generated samples) run with 1-fold cross-validation 75%-25% train-test split

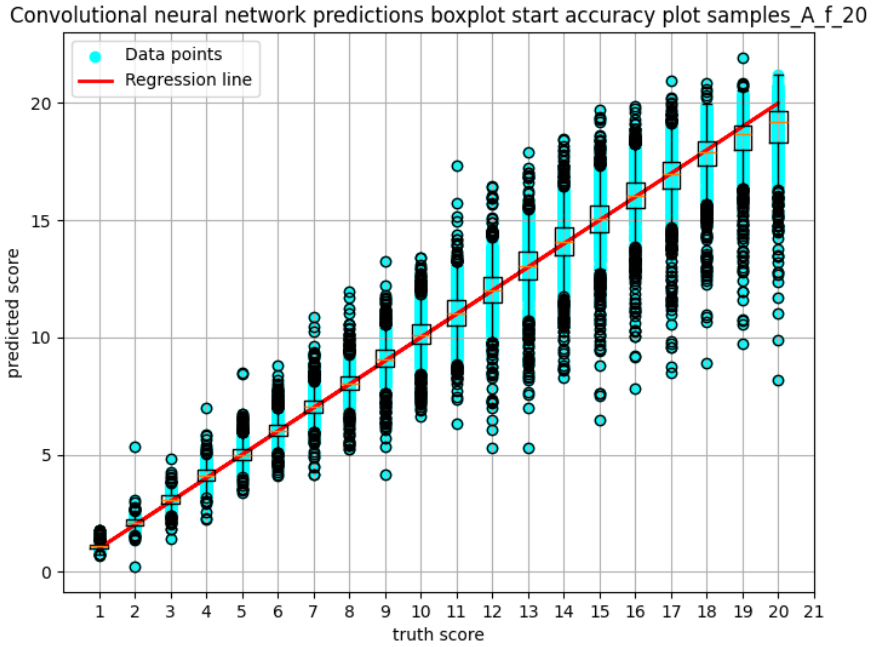


Figure 14: Boxplot convolutional neural network trained on dataset A (20% line chance backwards generated samples) run with 1-fold cross-validation 75%-25% train-test split

As for the other three models, they all follow a linear increase in value predictions as the truth score increases. The random forest model and CNN show a clear difference in accuracy compared to the KNN, as the KNN starts with consistent predictions around the truth score, but then as the truth score of the game states increases quickly spreads out the predicted values over a large range.
In contrast, the random forest model and CNN both display a consistent trend over the entire graph. Noticeable is that the CNN is much denser around the regression line compared to the random forest model. Even though the random forest model is utilized more as a control group, it seems to handle the problem surprisingly well. Though, the CNN clearly performs much better, even though the CNN has not been altered and was left on default parameters given by the scikit-learn documentation [11].

5.2.2 Different datasets experiment

The results of the second experiments, where the CNN is used to do a 1-fold cross-validation on the three datasets, can be seen in Figures figs. 15 to 17.
Figures figs. 15 and 16 show a very similar result. There is almost no distinction between the two, meaning that the line chance does not impact the actual CNN performance.
In contrast, the datapoints in Figure 17 are very sparsely divided, and there are too many outliers to be used in an actual MCTS experiment.

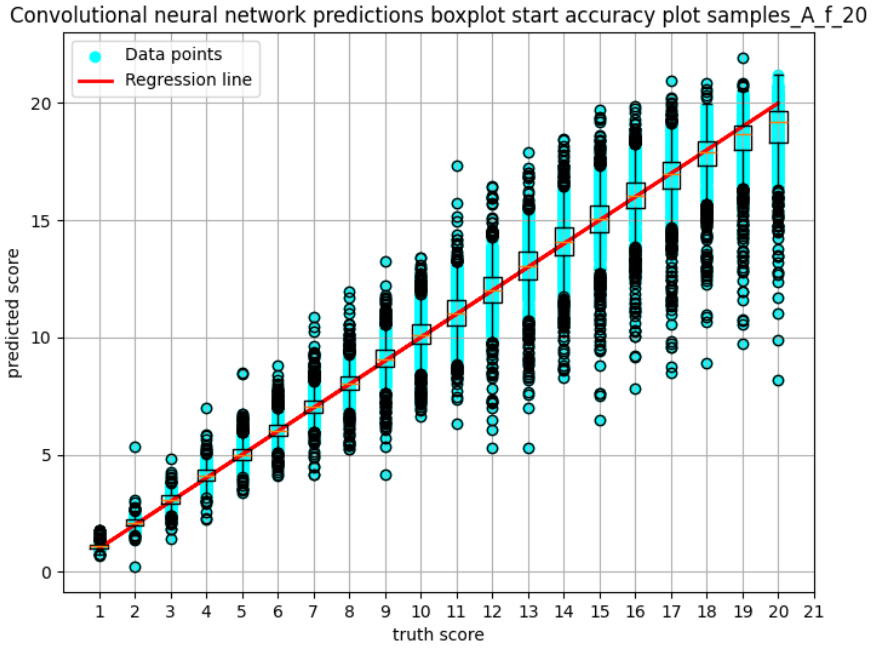


Figure 15: Boxplot CNN trained on dataset A (20% line chance backwards generated samples) run with 1-fold cross-validation 75%-25% train-test split

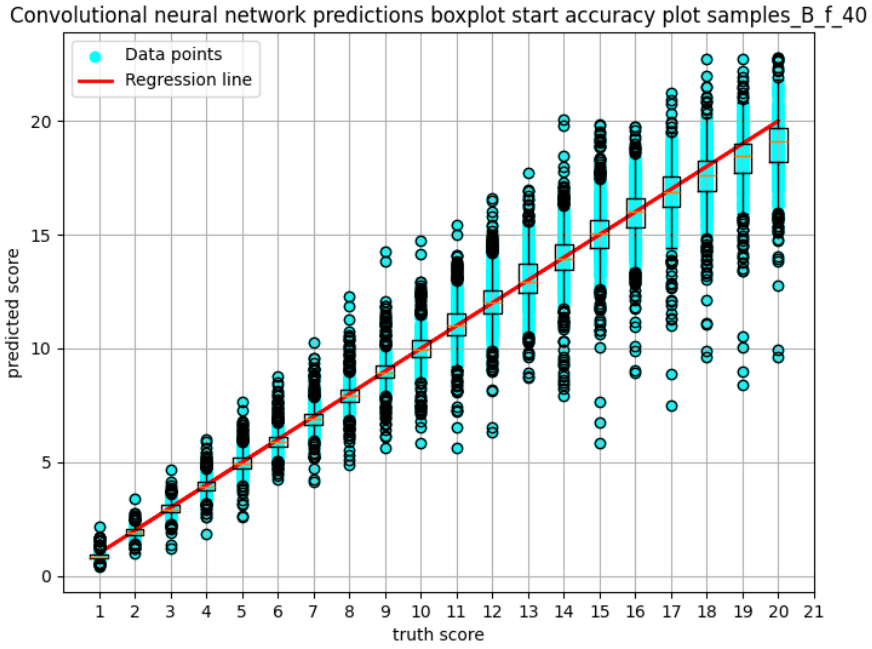
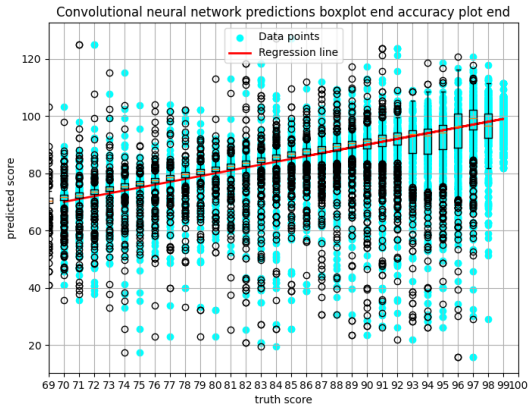
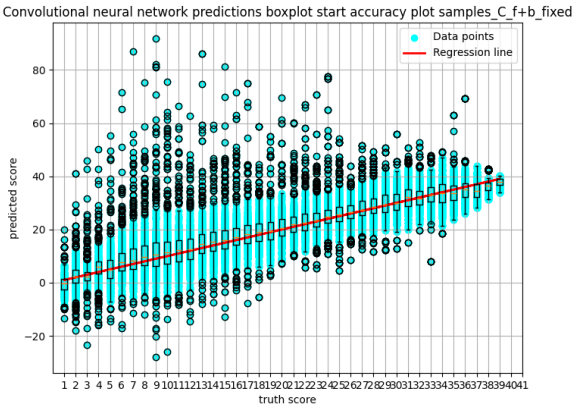


Figure 16: Boxplot CNN trained on dataset B (40% line chance backwards generated samples) run with 1-fold cross-validation 75%-25% train-test split



(a) Boxplot forwards generated samples



(b) Boxplot backwards generated samples

Figure 17: Boxplots CNN trained on dataset C (20% line chance forwards + backwards generated samples) run with 1-fold cross-validation 75%-25% train-test split

5.3 MCTS experiment

The MCTS algorithm was run with the following parameters:

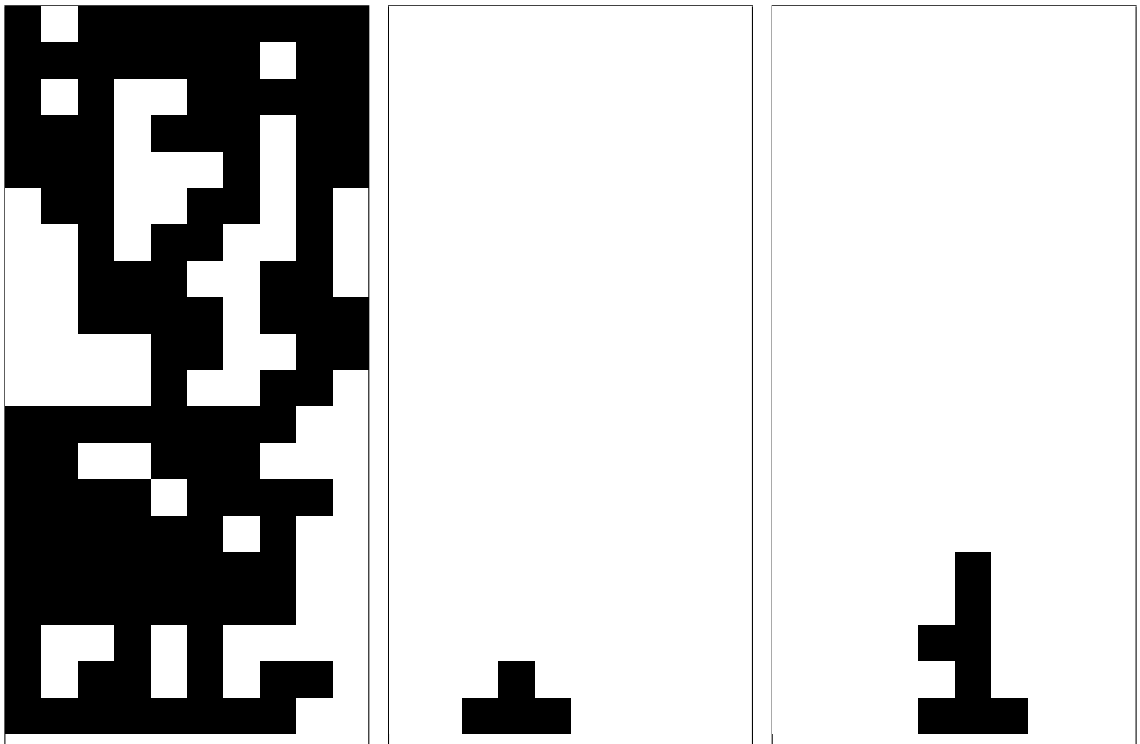
```
10000 iterations, 50 playouts
1000 iterations, 500 playouts
```

Both were run with a linear scoring scale without/with distinguished win vs. loss and UCT

selection without/with breadth-first approach.

5.3.1 Results

The algorithm saves the whole sequence of states as separate images for potential analysis, but for the sake of the results we only show the end state.
Unfortunately, the MCTS experiment in Figure 18a resulted in an ending state which does not resemble the spelled out "HI", regardless of the iterations, playouts, distinguished win/loss and breadth-first selection. Something noticeable is that the algorithm roughly attempted to do more moves by vaguely spreading out the pieces, but other than that, not much information can be obtained from this figure.
This experiment actually shows the single t-piece consistently over repeated runs, as can be seen in Figure 18b. However, this most likely has to do with the fact that for every future state, it is compared to the final state, and then considered winning if there is a direct match, and less with the MCTS implementation.
The experiment with two pieces as shown in Figure 18c shows that the MCTS implementation does not work. This could have multiple reasons, which will be discussed in the next section.



(a) End state of MCTS experiment with "HI" as final state (failed)

(b) End state of MCTS experiment with a single t-piece as final state (passed)

(c) End state of MCTS experiment with t- and l-piece as final state (failed)

Figure 18: Results of the three MCTS experiments

6 Discussion

The approach for generating labeled samples for the training process seemed feasible at first, but this approach might have caused a high bias towards the states near the final state, as it is more likely to create more duplicates of states with less reverse moves than ones that need many more moves, especially when generating thousands of samples.
The forwards generated samples were disregarded because of the results shown in the boxplots, but intuitively it was a bad decision to consider that doing more moves would end up with a better score as it would be closer to the final state that way.
The performance comparison was based on a 1-fold cross-validation test with a train-test split of 75%:25%. However, the samples used for the testing portion were very biased towards the method of sample generation, rather than samples that were more targeted towards the actual problem MCTS wanted to solve. This caused misleading results with the boxplots, as it seemed to perform very well on biased samples.
The previous section showed that the model training displayed very promising boxplots with the CNN boxplot being very stable. For this task, the model should have been sufficient enough, as MCTS only requires that values remain relative to eachother, so if all scores were shifted by the same factor, the algorithm would still be able to function. Of course there is much room for improvement in the training process with hyperparameter tuning, which could make the model predict more accurately.
As for the MCTS experiments, the general performance is not feasible yet. There are many factors that could have caused the undesired end result. The experiment size could have been too small, as the number of MCTS iterations or playouts might have needed to be much higher. Another flaw could have been the scoring system, where the linear scaling system might work for training the model, but in practice the predictions could have a bad influence on the difference in losing/winning states. The problem could have been too complex for MCTS to be solved, and the smaller problems were only considered at a late stage of the project. Starting off with the smaller problems could have given more insight into which parts of MCTS functioned properly, as the "HI" experiments displayed no information on any disfunctional parts of MCTS. Reinforcement learning might have solved this problem with less complexity. Regardless, we do have many improvements that can be applied for future research. In fact, we propose a different approach which could eliminate the bias, the bad forwards approach and the entire training process in Section 7.

7 Future work

7.1 Sample generation methods

The current approach for generating samples could be changed as to prevent bias towards states closer to the final state, and be more targeted towards the actual problem. Also, there could be experiments with the different chances of adding lines, as the 20% and 40% were arbitrary choices.

7.2 MCTS tuning

The current selection process was based on the basic MCTS UCT-calculation without changes. The UCT-calculation for the selection could be tuned. Also, heuristics and pattern matching

were not considered in this project, as this would have added much more complexity, but it definitely has potential for a better functioning MCTS algorithm. Finally, the reward calculation could be reworked, as the current implementation might not be feasible for MCTS to function properly.

7.3 Problem scaling

The project started on the difficult "HI" problem, and only at the end of the project were the small problems regarded. Starting the project with a small problem could give more insight in which parts of MCTS have certain causes to certain problems.

7.4 Other approaches

7.4.1 Playing MCTS without complex reward function

Playing MCTS without a reward function would change the entire experimental setup. The game can then either be played forwards or even backwards. If playing forwards, the game starts with an empty board, and the final state is considered a win. Playing backwards, the game would start at the final state and the game is then played backwards. In both cases, any state that is not the winning state is a losing one. This would increase the complexity by a lot, but it would prevent any problems that the linearly scaling scoring system could have caused. It would also save a lot of experimental effort, as the sample generation and model training are no longer required for the MCTS experiment. This approach could even be combined with heuristics and pattern recognition algorithms to simplify problems into subproblems. A rough overview can be seen in Figure 19

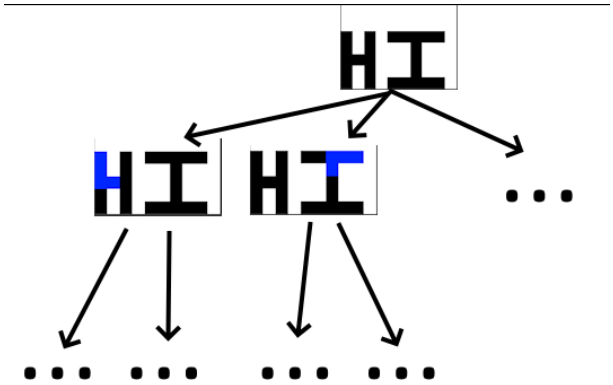


Figure 19: A new approach to MCTS: playing backwards

7.4.2 Reinforcement Learning

We have not touched the premise of Reinforcement Learning in this project, but there is much potential to solve the gamestate-reachability problem. A disadvantage of the MCTS method is that the trained CNN is very task specific, and if another final state were to be introduced, it would require the CNN to be trained on newly generated samples. Reinforcement Learning could circumvent this process through the use of unsupervised learning, which allows for flexibility around the provided final state and the training process as a whole.

8 Conclusion

We introduced a new approach to Tetris, where we create a new problem based on the reachability of a specified game state, and called it the gamestate-reachability problem. To automatize the process of finding a tetromino placement sequence to a specified game state we used Monte Carlo Tree Search (MCTS), with a Convolutional Neural Network (CNN) as the reward function, to solve this problem.
RQ1: How can an MCTS framework be set up for the gamestate-reachability problem?
We generated labeled samples by doing randomized backwards moves from the final state, and used these samples to train four different models. The linear regressor and k-nearest-neighbor (KNN) models showed either broken or poor results. The random forest model served as a baseline and the boxplot showed a consistent, but still slightly sparsely divided predictions. The CNN had the most dense and consistent performance on the dataset.
RQ2: How can the gamestate-reachability problem be solved using a MCTS framework?
Regardless of the performance of the CNN, the MCTS experiment displayed undesired results, as the generated sequences would lead to a seemingly random final state if the final state contains patterns that cannot be solved with one tetromino placement.
The approach of MCTS can be improved by either changing certain parameters within the experimental setup, or change the experimental setup to an approach where MCTS is played in a way where the scoring system is disregarded to prevent potential problems it could cause.

References

[1] Kai Arulkumaran et al. "Deep reinforcement learning: A brief survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38.

[2] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[3] Nuno Faria. *tetris-ai*. https://github.com/nuno-faria/tetris-ai. Accessed: 2024-04-23. 2019.

[4] Joachim Funke. "Complex problem solving: a case for complex cognition?" en. In: *Cogn. Process.* 11.2 (May 2010), pp. 133–142.

[5] GeeksforGeeks. *ML — Monte Carlo Tree Search (MCTS)*. May 2023. URL: https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/.

[6] hayoung-kim. *mcts-tic-tac-toe*. https://github.com/hayoung-kim/mcts-tic-tac-toe. Accessed: 2024-12-17. 2018.

[7] Jezevec10. *Jstris*. Online game. Accessed: 2025-01-20. 2017. URL: https://jstris.jezevec10.com/.

[8] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[9] Zewen Li et al. "A survey of convolutional neural networks: analysis, applications, and prospects". In: *IEEE transactions on neural networks and learning systems* 33.12 (2021), pp. 6999–7019.

[10] Nicholas Lundgaard and Brian McKee. "Reinforcement learning and neural networks for tetris". In: *Technical report, Technical Report, University of Oklahoma* (2006).

[11] *MLPRegressor*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html.

[12] Ian Osband et al. "Deep exploration via bootstrapped DQN". In: *Advances in neural information processing systems* 29 (2016).

[13] osk. *TETR.IO*. Online game. Accessed: 2025-01-20. 2019. URL: https://tetr.io/.

[14] Alexey Pajitnov. *Tetris*. Video game. 1985.

[15] Guinness World Records. Nov. 2021. URL: https://www.guinnessworldrecords.com/world-records/most-ported-computer-game (visited on 12/02/2025).

[16] Matt Stevens and Sabeek Pradhan. "Playing tetris with deep reinforcement learning". In: *Convolutional Neural Networks for Visual Recognition CS23, Stanford Univ., Stanford, CA, USA, Tech. Rep* (2016).

[17] *Tetris DS*. Video game. Nintendo DS. 2006.

[18] *Tetris/Rotation systems*. Online resource. Accessed: 2025-03-27. 2022. URL: https://strategywiki.org/wiki/Tetris/Rotation_systems.

[19] vgarciasc. *mcts-viz*. https://github.com/vgarciasc/mcts-viz. Accessed: 2024-12-17. 2020.

[20] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

9 Appendix

9.1 MCTS algorithms

Algorithm 2 Selection implementation

```
procedure SELECT(root_node)
    current_node ← root_node
    parent ← root_node
    iterated ← True
    while current_node has children and iterated do ▷ Iterative while through the tree
        iterated ← False
        best_uct = uct(current_node, parent.visits)        ▷ parent visits required for
        best_node = current_node                           ▷ calculation
        for child in current_node.children do
            child_uct = uct(child, current_node.visits)
            if child_uct > best_uct then
                best_uct = child_uct
                best_node = child
                iterated ← True
            end if
        end for
        parent = current_node
        current_node = best_node
    end while
    Return best_node
end procedure
```

Algorithm 3 Expansion and simulation implementation

```
procedure EXPAND_AND_SIMULATE(parent_node)
    playouts ← 50
    tetrominos ← list of all 7 tetrominos in random order
    for piece in tetrominos do    ▷ attempt first tetromino, if no options, try next etc.
        next_states ← all states of piece placements that have not been done
                                                    ▷ parent node can already have children
        if next_states not empty then
            action ← random state from next_states
            child_node ← board properties from action
            parent_node.children.append(child_node)
            child_node.parent ← parent_node

            procedure DO_PLAYOUTS(child_node)
                total_score ← 0
                copy_node ← copy(child_node)   ▷ deep copy to preserve child node state
                n ← 0
                while n < playouts do
                    tetrominos2 ← list of all 7 tetrominos in random order
                    for piece2 in tetrominos2 do          ▷ attempt first tetromino etc.
                        if piece2 can be placed then
                            add piece2 to copy_node board
                            break out of the For-loop
                        end if
                        if piece2 has not been placed then          ▷ end of game reached
                            score ← CNN.predict_score(copy_node)
                            total_score ← total_score + score
                            n ← n + 1
                            copy_node = copy(child_node)          ▷ restart with fresh copy
                        end if
                    end for
                end while
                if total_value < 0 then                  ▷ round values to losing/winning
                    Return −1/playouts
                else
                    Return 1/playouts
                end if
            end procedure

            value = do_playouts(child_node)
            Return child_node, value
        end if
    end for
end procedure
```

Algorithm 4 Backpropagation implementation

procedure BACKPROPAGATE($expanded\_node, value$)
  $expanded\_node.wins \leftarrow expanded\_node.wins + value$
  $expanded\_node.visits \leftarrow expanded\_node.visits + 1$
  while $expanded\_node.parent$ exists **do** ▷ Iterative while to the root
    $expanded\_node \leftarrow expanded\_node.parent$
    $expanded\_node.wins \leftarrow expanded\_node.wins + value$
    $expanded\_node.visits \leftarrow expanded\_node.visits + 1$
  end while
end procedure

9.2 Hardware specifications

The experiments were run on an Intel i5-8600K processor and an NVIDIA GTX-1070ti.

9.3 Software specifications

The experiments were run on Windows 10 in a Powershell virtual environment with the following pip packages:

Listing 1: requirements.txt

```
contourpy==1.2.1
cycler==0.12.1
descartes==1.1.0
desdeo-problem==1.5.0
desdeo-tools==1.8.0
diversipy==0.8
fonttools==4.51.0
hvwfg==1.0.2
joblib==1.4.2
kiwisolver==1.4.5
llvmlite==0.39.1
matplotlib==3.9.0
ml-dtypes==0.2.0
numba==0.56.4
numpy==1.23.5
optproblems==1.3
packaging==24.0
pandas==1.5.3
pillow==10.3.0
protobuf==4.23.4
pyparsing==3.1.2
python-dateutil==2.9.0.post0
pytz==2024.1
scikit-learn==1.5.0
scipy==1.13.0
Shapely==1.8.5.post1
six==1.16.0
```

```
tensorboard==2.15.1
tensorflow==2.15.0
tensorflow-estimator==2.15.0
tensorflow-intel==2.15.0
threadpoolctl==3.5.0
wrapt==1.14.1
```