Tetris AI

Thawsitt Naing
Stanford University
thawsitt@stanford.edu

Adrien Truong
Stanford University
aqtruong@stanford.edu

Orien Zeng
Stanford University
ozeng@stanford.edu

***Abstract*—This paper explores different approaches to implement an AI agent that maximizes the number of rows cleared in the game of Tetris. We model the game as a Markov Decision Process and explored Deep Q-Learning, Limited Depth Search, and Monte Carlo Tree Search. We found Monte Carlo Tree Search method, using a heuristic function for roullouts, to be the most successful, with the agent achieving superhuman level and clearing over 3500 lines. The features used and the heuristic function greatly impact the performance of our AI agent.**

Keywords*—*Tetris, Game Playing, MDP, Monte Carlo Tree Search, Deep Q-learning, Feature Engineering.

I. Introduction

Game playing has been a significant field in the research and development of artificial intelligence. Our goal for this project is to implement an AI that plays the game of Tetris at a superhuman level.

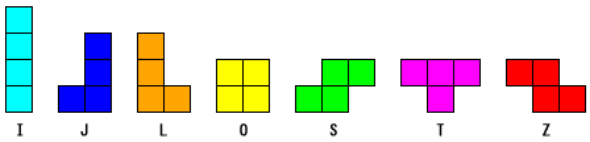

Fig. 1: Seven Possible Tetromino Shapes

Tetris is an incredibly popular single player arcade game. The game pieces come in 7 distinct shapes, each called tetrominoes as they are composed of four square blocks arranged in different positions. The pieces are shown in Figure 1. Players stack the pieces on the grid one at a time to form horizontal lines that get cleared from the grid. Players can choose where to stack each pieces and how to orient each pieces. The game is ultimately lost when the grid overflows with blocks.

II. Task Definition

A. Game Setup

Tetris takes on many varieties and there is no standard version. Therefore, we specify the rules and configuration of the version our AI will be playing:

1) Our Tetris grid size is 23 rows by 10 columns.
2) Our set of shapes will be all seven possible tetrominoes.
3) At each time interval, a player can take one of the following actions: move piece left, move piece right, rotate piece clockwise, rotate piece counterclockwise, do nothing.
4) After each action, the game's time is incremented and the piece moves down 1 space.
5) A player is able to see the next piece. (1-piece look ahead)
6) Each upcoming piece is randomly generated with each tetromino having probability 1/7.
7) Score is equal to the number of rows cleared. There is no bonus for clearing multiple rows at once.

B. Objective

The objective of our AI is to maximize the score which is equivalent to maximizing the number of rows cleared. At any given time, our AI has access to the following set of information:

1) The current board state. More formally, a 23×10 matrix where $B[i, j] = 1$ indicates that a piece occupies location (i, j) on the board.
2) The current piece, its location and rotation on the board.
3) The upcoming piece.

III. Literature Review

In 2013, Yiyuan Lee worked on a Tetris AI which can almost indefinitely clear rows. [1] The goal of the AI was to clear as many lines as possible, and therefore, to make as many moves as possible. The AI decides the best move for a given Tetris piece by trying out all the possible moves (rotation and position) and computing a score for each possible move (together with the look-ahead piece). The author used a Genetic Algorithm to come up with the parameters for the score formula, which accesses four heuristics: aggregate height, complete lines, holes, and bumpiness, each of which the AI will try to either minimize or maximize.

Similar Tetris AI project has also been attempted by Stanford students. One was done for a CS231N project [2], where Matt Stevens and Sabeek Pradhan used deep reinforcement learning to train an AI to play Tetris. For their project, they used a convolutional neural network to estimate a Q function that describes the best action to take at each game state. Their resulting AI demonstrated the basic ability of a neural network to learn to play Tetris.

IV. Setup

A. Infrastructure

To develop our agent, we built our simulator on top of an existing Python implementation of Tetris [3]. It is a barebones terminal-based implementation that omits a GUI. This implementation allows us to play the game without having to output a UI showing the state of the game. For our work, this is ideal because it allows us to quickly simulate games and easily copy and manipulate state.

Our agent interacts with the game via one function, $get\_action()$ which takes in inputs described in Task Definition. The agent returns a single action or list of actions to play next in this function.

B. Model

We modeled the problem as an MDP; the state is completely observable, and additionally the transition probabilities are known. The game is deterministic except after a piece is placed, where there is an uniform chance for each of the 7 possible next pieces to be chosen. In the final evaluation, there is no discounting; however, methods using the bellman equation use a discount factor of 0.95 for training. Although the problem can be stated as an mdp, simple approaches such as value iteration face challenges due to the size of the state space; with no simplification, the number of states is $2^{10*23} * 7 * 4 * 7 * 10 * 23$, which makes value iteration intractable. We summarize our MDP below:

s_{start}: (empty board, current piece, upcoming piece)

$Actions(s)$: left, right, rotate clockwise, rotate counterclockwise, drop, do nothing

$T(s, a, s')$: Actions occur with probability 1. When placing a piece in its final location, upcoming piece becomes current piece and upcoming piece takes on a new value randomly with each tetromino having probability 1/7.

$Reward(s, a, s')$: Actions have 0 reward unless they result in rows cleared, in which case the reward is number of the rows cleared.

$Discount factor$: Rows cleared now are equal to rows cleared later. Therefore, we use a discount factor of 1.

V. Baseline and Oracle

A. Baseline

To provide a point of comparison for our work, we started by developing a simple baseline. Our baseline is a greedy algorithm that searches for the locally optimal move. It evaluates all possible final locations of a piece by comparing the top y value of the piece when it is placed. We define the locally optimal position as the one that minimizes the top y value of a piece. We then generate a list of moves to place the piece at the locally optimal position. Over

50 Tetris games, this baseline cleared an average of 5.06 rows.

B. Oracle

For our oracle, we looked at previous implementations. We ultimately borrowed an algorithm that utilized a depth 2 limited search with a linear predictor as a function approximator. The linear predictor uses the following features:

1) Number of holes: $\sum_{y,x}[\exists a, a > 0 \wedge Board(y + a, x) = 1]$
2) Aggregate height: $\sum_{x} Height(x)$
3) Bumpiness: $\sum_{x} |Height(x) - Height(x-1)|$
4) Number of rows cleared

The weights were learned using a genetic algorithm. In terms of performance, the oracle averages a score of 1246 which is orders of magnitude better than the baseline and, arguably, at the level of superhuman performance. In following sections, we describe alternate approaches to try to close the gap between our baseline and oracle.

VI. Approaches

A. Depth Limited Search

Our first starting point was extending our baseline to optimize globally instead of just locally. To do this, we augmented our evaluation of the board state after a move. In addition to just the top y value of the piece we just placed, we added the following heuristics:

- number of holes created
- number of rows cleared
- bottom y value of the piece at its final location
- y value of the highest piece on the board

We weighted each of these factors to generate a score and manually tuned each weight by hand using trial and error. Now, instead of just evaluating a final location for a piece based on just a single heuristic, the top y value of the piece, the agent evaluated a location based on multiple heuristics each weighted a different amount. With these new heuristics, we yielded an average score of 50 rows cleared over 50 runs which is a 10x improvement from our original baseline.

The next observation we made is that the AI has access to the upcoming piece but does not utilize this information. To incorporate this information, we increased the depth of our search to 2. With this modification, we again significantly improved the performance of our agent.

B. Deep Q-learning

The state space for tetris is too large to do pure q-learning, so we use Q-value approximation using a neural network. The state representation of this problem is the same as it is for the baseline: two dimensional board, piece position and rotation, etc. The output of the neural network will be a score for each action (Left, Right, Down (all the way down), Nothing (down by one), RotateCounterclockwise, RotateClockwise), where that score represents the Q-value for the state, action pair given a discount factor (note that the discount factor is only used for training; all agents are compared using the same evaluation function of the number of rows cleared). The model performed Q-learning updates initially over state, actions pairs chosen by the greedy algorithm, and once the neural network is partially trained the model would use choose state, action pairs based on its own score function. At test time the algorithm would greedily pick the action with the highest score.

The initial neural net architecture consists of a convolutional layers over the board using 5 x 5 filters with depth 16 and stride 1, followed by 4 x 4 filters with stride 2 and depth 32. The output of the convolutional layers is flattened and concatenated to miscellaneous features such as the position and rotation of the current piece. This new feature vector is fed into two affine layers, with output width 256, and 6 parameters, respectively. The final output is interpreted as the Q-values for each action. The network is trained using gradient descent updates using the Bellman equation.

Based on the methods of the DQN paper [4], we used the following methods to increase stability during training:

- We used a replay memory of size 100,000 consisting of state, action, reward, state' pairs generated by interactions with the environment. The replay memory would assist in reducing correlations between consecutive training data while performing stochastic gradient descent. In each training iteration, the agent (either the greedy baseline or the DQN agent) would interact with the environment to generate a SARS' tuple, which is added to the replay memory. Then, a minibatch of size 32 would be sampled randomly from the replay memory to perform a gradient descent update.
- We froze the target model so that our model would have a stable environment to train against. The target model is a previous iteration of the model weights, and the target would refresh every 30,000 mini-batch updates.

C. Heuristic Function

For our heuristic function, we added more features as follows.

1) Maximum height
2) Minimum height
3) Average height
4) Squared difference between the maximum and minimum height
5) Number of holes
6) Bumps: how "bumpy" the pieces are; we calculate by summing of the absolute values of the differences between each column height
7) Rows cleared so far

The weights of each feature are manually tuned through trial and error. Our heuristic function performs relatively well and plays an important role in the "rollout" stage of our next approach, Monte Carlo Tree Search.

D. Monte Carlo Tree Search

Next, we approach Tetris using Monte Carlo Tree Search [6]. We define a class to represent the nodes in the game tree. Each game node holds the following information.

- $Move$: the move that got us to this node; None for the root node
- $Depth$: the depth of the current node in Monte Carlo Tree
- $ParentNode$: None for the root node
- $ChildNodes$: an array of containing child nodes that have been explored
- $RunningScore$: the total rows cleared before reaching this node
- $TotalScore$: total score obtained from all the simulations
- $Visits$: the number of times this node has been visited.
- $UntriedMoves$: an array containing future child nodes to be explored
- $ActionList$: our MCTS algorithm returns a sequence of moves to take

Dividing $TotalScore$ by $Visits$ gives us the average score of a child node which is the average score of taking a particular move.

After we have defined the root node as described above, we conduct a Upper Confidence bound for Trees (UCT) search for a maximum number of iterations. UCT combines Monte Carlo Tree Search (MCTS) with Upper Confidence Bound (UCB) method. Our algorithm performs the following steps:

1) **Selection**: Starting at root node R, recursively select optimal child nodes (explained below) until a leaf node L is reached.
2) **Expansion**: If L is a not a terminal node (i.e. it does not end the game) then create one or more child nodes and select one C.
3) **Rollout**: Estimate the score of a potential move using the heuristic function.

4) **Backpropagation**: Update the current move sequence with the estimated result.

We use Upper Confidence Bound (UCB) formula to select the optimal child nodes during the exploration [7]. This provides a good balance between the exploitation of known rewards and the exploration of relatively unvisited nodes. In the rollout stage, doing the actual Monte Carlo simulation to estimate the score of a child node is impractical for our project as it takes too much time for each move. So, instead of simulating the game, we use our heuristic function to calculate the estimated score for a given game state, which is then used to update the Monte Carlo Tree.

VII. Results

A. Performance

TABLE I: Average scores of agents

Agent	Number of Rows Cleared
Baseline	4
Oracle	1246
Depth 1 Limited Search	134
Depth 2 Limited Search	1103
MCTS (500 iterations)	150
MCTS (1600 iterations)	1784
MCTS (3200 iterations)	3548
DQN	0

Looking at our results, we can see that depth 2 limited search greatly outperforms depth 1 limited search. It might seem surprising at first that an increase in depth of just 1 drastically affects performance. However, this can be explained by the fact that depth 2 limited search is able to utilize the information about the upcoming piece. In addition, since our oracle is identical to our depth 2 limited search, just with a better heuristic function, we see that a machine learned heuristic function actually does not drastically improve performance.

Looking at Monte Carlo Tree Search, we see that it performs even better than a depth limited search and actually, with enough iterations, outperforms our oracle. We can attribute this to MCTS's ability to avoid exploring branches that are likely bad. In the same number of iterations, it is able to explore promising branches more deeply. We can also see that MCTS enables us to arbitrarily scale how hard we want to work to compute an optimal action. With depth limited search, increasing the depth does not scale linearly with computation time. From depth 2 to depth 3, we go from just 1,600 states to explore to 64,000. Depth 2 may be tractable but depth 3 isn't and there's no way to interpolate between the two.

Another thing to note is both depth limited search and MCTS have high variance. There appear to be certain game scenarios which our agents do not perform well on. They score well below the average, averaging only a couple hundred rows. The oracle exhibits the same behavior. Considering the oracle has machine learned weights, this may indicate that the features we used cannot fully express the value of a state.

B. DQN Results

Initial results of the neural network Q-learning were unsuccessful due to the Q-learning training based on state, action, reward pairs generated by a random agent. The reward would almost always be 0 because the agent is unable to clear a single row. So, the scoring output of the neural net would simply converge to 0 for each action, especially when there is greater discounting. This was improved with the introduction of the greedy baseline, which produced rewards more often than the random agent does and therefore allowed the Q-learning to train at a reasonable rate. However, upon training on the greedy algorithm we found that the neural network would give all actions a similar Q-value and would often pick the same action over and over, so the network approach is currently not very effective.

This behavior may be explained by little variation in estimated q-value compared to the bias on the action. For example, two example score outputs are [0.50888866 0.50817966 0.50957996 0.22736108 0.52800912 0.53410703] and [0.50888866 0.50817966 0.50957996 0.22736108 0.52800912 0.53410703]. The first two elements of each vector correspond to LEFT and RIGHT, respectively. So, we see here that the network was not successful at using the board to differentiate between actions.

Another issue is that sometimes the board weights and loss would roll out of control to the

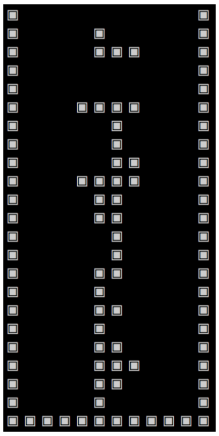

Fig. 2: The deep-q-learning agent picked drop over and over, causing 0 rows to be cleared

point where all scores are NaN. The introduction of model freezing and experience replay eliminated this problem, allowing the model to be trained for hundreds of thousands of iterations.

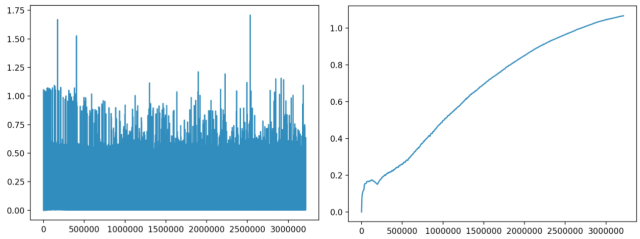

Fig. 3: Loss vs time graph and initial state score graph

On the left side of figure 3, we plot loss vs time, which is very noisy. The spikes are likely associated with losing moves, which have a large negative reward. In the DQN paper, the loss function was also very noisy, but the Q estimate of some predefined states would monotonically increase, implying the model is learning to generate better scores. The plot on the right shows the score assigned by the network to the initial state, which is similar to the one in DQN. However, this does not imply the model is learning because our model is training on SARS' tuples chosen by a different agent, so an increase in Q value of the initial state is not directly associated with the reward that the DQN policy would generate.

After more training, the model picked different actions based on different board states; however, the agent was still not successful at making intelligent decisions on where to place each piece.

The behavior of the DQN model was not useful; given similar looking board states, the model would pick the same action. The column of pieces on the left in the above were all stacked together in a row when the agent picked LEFT over and over. After forming that column, the agent switched to moving pieces to the right. This suggests that the agent is using the board to inform action choices; however, the agent needs to have a much finer understanding of the board before it can clear any rows.

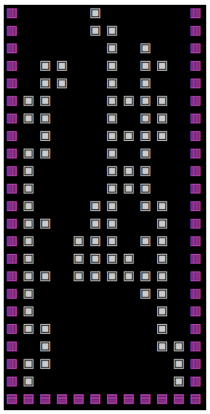

Fig. 4: Despite showing improvements, the DQN model was not able to clear any rows.

VIII. Future Work

A. *Deep Q-Learning*

The deep q-learning approach produces poor results and future work would simply involve debugging the implementation. Prior work in tetris proposed a network architecture with a column-based representation [2], because of the intrinsic meaning they have in the game. Overall, deep q-learning is difficult to train and needs investigation as to how to make it work.

B. *Monte Carlo Tree Search*

The results for monte carlo tree search indicate that increasing the number of iterations greatly increases the performance of the agent. Therefore, if

we can speed up monte carlo tree search, we can squeeze in more iterations in the same time. Speeding up monte carlo tree search falls into mostly two categories.

The first method would be a benchmark of the code and analysis of bottlenecks. For example, almost 50% of program time is spent copying tetris board states while creating a new leaf node. The leaves eagerly generate their successor nodes, meaning that many nodes are generated that are never visited. Lazily generating nodes would drastically increase the speed of montecarlo.

The second method involves parallelization of the monte carlo updates. Parallel Monte Carlo Tree Search [5] details parallelization methods used in monte carlo tree search. In many monte carlo applications, the algorithm is bottlenecked by the speed of rollouts. In that situation, it would be possible to achieve multithreading using a global mutex on the state tree, since time spent navigating the tree is a small portion of overall time spent. However, in our case, we dont perform rollouts and performance is bottlenecked by the node expansion process and node copying.

Another method is to use multiple rollouts, and use a thread for each rollout. This allows the future utility estimate of each node to be less noisy, because rollouts are random. However, the heuristic function we use is deterministic, so multithreading would not be useful for utility estimation.

Therefore, the most promising parallelization method would be local mutexes, which only lock the portion of the game tree that the thread is using. While this comes at the cost of performing many mutex lock/unlock operations, it allows multiple threads to perform selection, expansion, and backpropagation in parallel.

IX. Conclusion

In the end, we learned that there are many challenges in training deep Q-learning. We also discovered that Monte Carlo Tree Search is very effective at playing Tetris given a high number of iterations. In fact, with 1600 iterations, MCTS can outperform our oracle and clear 1784 rows on average. One of the challenges of Monte Carlo Tree Search approach was the ”rollout” stage which can take extremely long if traditional Monte Carlo simulation is done. We solved this timeout problem using our own heuristic function to get a quick estimate. A good heuristic function can drastically improve the performance. The features used in the heuristic function also impact the performance of the algorithm. After experimentation, we were able to engineer features and weights that provide optimal results.

References

[1] L. Yiyuan. Tetris AI The (Near) Perfect Bot
https://codemyroad.wordpress.com/2013/04/14/
tetris-ai-the-near-perfect-player/

[2] S. Matt, P. Sabeek. Playing Tetris with Deep
Reinforcement Learning
http://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf

[3] F. Recanatini, ”termotetris” Tetris Game Source Code
https://github.com/feychou/termotetris

[4] V. Mnih et al. Playing Atari with Deep Reinforcement Learning
https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[5] G. Chaslot et al. Parallel Monte Carlo Tree Search
https://link.springer.com/chapter/10.1007/978-3-540-87608-3_6

[6] P. Cowling, E. Powley, D. Whitehouse. MCTS.ai Python Code
http://mcts.ai/code/python.html

[7] MCTS.ai. What is Monte Carlo Tree Search?
http://mcts.ai/about/index.html