AI Agents for Playing Tetris

Sang Goo Kang and Viet Vo
Stanford University
sanggookang@stanford.edu
vtvo@stanford.edu

***Abstract*— Game playing has played a crucial role in the development and research of artificial intelligence. One such result of game play research is in the development of AI agents that have sophisticated deduction, reasoning, and problem solving skills. Our goal for this project is to create AI agents that can effectively solve the game of Tetris, and determine which algorithm performs the best. We performed a greedy-search and a depth-2 search combined with a genetic algorithm and Nelder-Mead optimization to solve the Tetris game. On average, we discovered that the depth-2 search optimized with the nelder-mead algorithm performed the best, being able to clear up to 4000-5000 lines.**

I. INTRODUCTION

Our interest for this project is to create an AI agent to effectively play the game of Tetris. The game consists of a 20 row by 10 column board, and seven different tetrominoes. These tetrominoes consists of the S-shape, Z-shape, T-shape, L-shape, Line-shape, MirroredL-shape, and the Square-shape, all of which are composed of 4 blocks. The goal of the game is to rotate and place the randomly-generated falling tetrominoes in such a way as to achieve as many filled rows of blocks on the board as much as possible. Every time a row is filled, the row will be removed and the leftover blocks above the row will be shifted downwards. The player loses the game once a new tetromino can no longer fit on the board the moment it is generated.

II. TASK DEFINITION

The primary task is to develop AI agents that can determine the best set of rotations and translations of the tetrominoes to achieve as many filled rows on the board as possible. In order to accomplish this task, it is crucial to engineer features that describe the problem well, and optimize the weights for each feature. *Table 1* displays the state variables of each Tetris game state. The variables describe the current layout of the board, game score, and total number of lines cleared so far. The performance of our AI was evaluated based on the average number of lines that it clears before losing.

III. PREVIOUS WORK

Solving Tetris can be accomplished through several techniques. Search algorithms and learning algorithms are both popular methods. A paper written in the University of Oklahoma looks into the use of both deep learning and reinforcement learning to develop an AI for Tetris. They utilized neural networks and Q-learning to train their AI agents [2]. In contrast to our methods, this paper used neural networks in conjunction with reinforcement learning for an unsupervised task. The neural network was used to summarize the state-action policy as well as the expected rewards. Another paper treated the Tetris game as a Markov Decision Process and used fitted value iteration to deal with the large state space [4]. This paper discovered that the MDP did not work very well and found that using search algorithms with parameter learning performed significantly better. We plan to explore search algorithms and find our parameter weights using optimization instead of learning.

IV. SETUP

The tetris program was a pygame program that was adapted from an open source github repository [3]. We utilized python to code the majority of our algorithms. Whenever a new tetris block is generated, the current state is generated and put in a thread-safe queue for the AI to access. The AI returns a sequence of moves on another thread-safe queue for the tetris program to access. The AI program was developed to run on a separate

TABLE I

TETRIS STATES

State Variables	Description
curr_board	A grid of binary numbers where 1 indicates a filled block occupied by a tetromino, and 0 otherwise
score	The current score of the game
next_stone	The next tetromino to appear on the board
total_lines	The number of lines cleared so far.
stone_pos	The x-y coordinate of the current tetromino on the board
level	The level of the game increases throughout the game, further increasing the drop speed of the block
is_end	Indicates whether or not we have reached the game over state

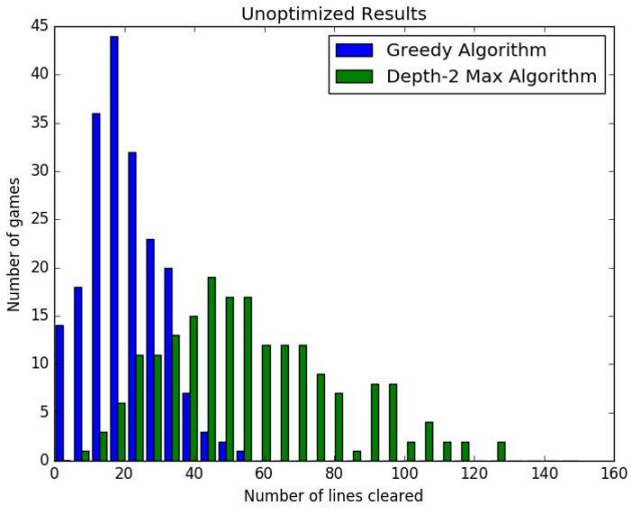

Fig. 1. Unoptimized Results using greedy and depth-2 search.

thread. This was done in order to develop an AI that plays the game in real time without having the game pause for the AI to make its move.

Our input into our AI program is the current state of the Tetris game, which consists of the variables shown in *Table 1*. We modeled possible successor states as all the possible actions a tetromino can take, where an action is defined by the number of rotations and horizontal shifts it performs. The end state is defined by the state where a tetromino can no longer fit on the board.

V. BASELINE AND ORACLE

In the search algorithm, the action was defined as the number of rotations and the horizontal translation of the piece, assuming that the piece will be dropped from that point to get to the next state. For our baseline, we implemented a greedy search algorithm with a simple evaluation metric to set a lower bound for game performance. For each new input stone, we exhaustively search every x-position on the board to find the location that has the maximum amount of drop distance to the bottom of the current board landscape. The AI also tries to minimize the number of holes it will create once placed. We gave the minimization of holes a lower priority than drop height for our baseline. The baseline does not predict future blocks, or minimize holes effectively, and is shown to clear on an average of 10-20 lines, as shown in the blue distribution in *Figure 1*.

The oracle would essentially be a perfect AI that can play indefinitely with perfect placement of tetrominoes and operate at fast speeds. The oracle would be able to ensure every shape that has been dropped gets cleared. The number of lines cleared in the long run would approximate to $L = n * (4/10)$, where L is the number of lines cleared, n is the number of shapes dropped, 10 being the width of the board, and 4 being the number of blocks in a tetromino. The gap between the baseline and the oracle is tremendous when regarding the effectiveness of clearing lines on the board, and speed since the baseline is extremely limited in terms of its behavior whereas the oracle can theoretically never lose at the game.

VI. FEATURES

The initial features in the baseline simply included the number of holes in the board as well as the drop height of the immediate piece that has been dropped. To improve the baseline, the feature extractor was modified to have thirteen features, some of which were derived from previous work [5]. The feature extractor was designed so that the state of the board is used to calculate the features. The ordering of the features was hard-coded in a separate utility file, so that the features can be represented as a simple number array. The features along with their descriptions can be seen in *Table II*.

TABLE II
FEATURES USED

Feature	Description
totalHeight	The sum of the heights of each column.
maxHoleHeight	The height of the highest column that contains a hole.
numHoles	The number of open spaces with a fill space somewhere directly above it.
playableRow	The lowest row that can be cleared without clearing any row above it.
numHoleRows	Number of rows that contain at least one hole.
numHoleCols	Number of columns that contain at least one hole.
numBlockades	Total number of filled spaces that have a hole somewhere directly beneath it.
clearedRows	Number of lines cleared in the last move.
bumpiness	The sum of the absolute value of the slope of column heights.
concavity	The sum of the absolute value of the concavity of the column heights.
maxHeight	The height of the tallest column.
score	The current score in the game.
numLines	The total number of lines cleared in the game.

VII. ALGORITHMS

A. SEARCH ALGORITHMS

The two primary search algorithms used include the greedy search algorithm as well as the depth-2 max algorithm. The greedy search algorithm searches through every possible rotation and horizontal position for the current stone to be dropped, to find the best possible way to drop it. This results in about 40 different combinations since each tetromino can be rotated 4 times and translated to 10 different positions. The depth-2 search algorithm performs this search behavior on the current tetromino as well as the next piece by performing a look ahead. This will allow the AI agent to drop the pieces in the most optimal positions while taking the next state into account. This results in an upper bound of 40^2 evaluations for each action.

B. EVALUATION FUNCTION

Two evaluation functions were tested. One is a simple linear classifier, where the score can be represented as:

$$score = \theta^T \phi(x) \tag{1}$$

Another evaluation function that was used was a neural network, in which the neural network was used simply as a way of generating non-linear relationships between the features. The output for each node in the neural network was set to be a linear function of its inputs. In this case, θ is represented as a list of matrices.

$$score = \theta_l^T(\theta_{l-1}^T(...\theta_1^T\phi(x))) \tag{2}$$

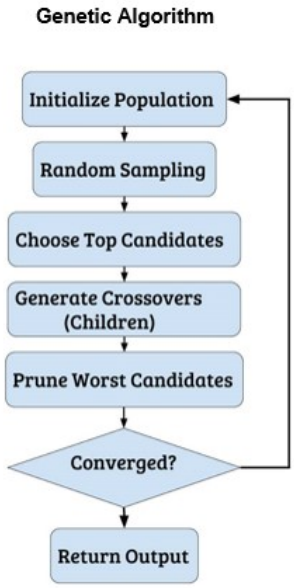

Fig. 2. Genetic Algorithm

C. OPTIMIZATION ALGORITHMS

In our problem definition, we can define this problem as an input/output formula, where the input is an array of numbers corresponding to weights of features, and the output is the average number of lines cleared when playing tetris with those weights. With this, we can run non-linear optimization algorithms in order to solve for the most optimal weights given a specific feature set.

The two primary optimization algorithms we explored included the Nelder-Mead optimization algorithm as well as the genetic algorithm. The Nelder-Mead method is an optimization algorithm that attempts to minimize or maximize a multi-dimensional optimization problem. It does so by using a combination of $n+1$ vertices in a problem

TABLE III
GENETIC ALGORITHM HYPERPARAMETERS

Hyperparameters	Values
Population Size	100 Sets of Weights
Number of Children	25 New Children/Gen.
Mutation Rate	0.20
Mutation Delta	± 0.25

where $x \in \mathbb{R}^n$. The $n+1$ points is represented as a simplex which is a n-dimensional polytope. For each iteration, the polytope moves one vertex at a time until it eventually converges at a local minimum/maximum [7].

The genetic algorithm mimics evolution in order to minimize/maximize an optimization problem. The population is initialized with sets of random weights or "genes", from which a fitness function is used to determine the most "fit" examples. In order to limit the possible values, the random weights were scaled in order to fit in a unit hyper-sphere. This is done due to that fact that for the evaluation function, the relative weights dictates the behavior, instead of the absolute weights. For our fitness function, we simply assigned each set of weights a fitness equal to the average number of lines that gets cleared by using its parameters. Subsequently, crossovers (children) are generated by using Equation (3) where c denotes a new child, while p_1 and p_2 represents the two parents. The children also include a small chance of mutations in order to prevent the function from being stuck at local minima/maxima. From this, the least fit of the population are pruned. This process is repeated until the number of iterations is hit or the population converged to a specific fitness, after which the population is returned. This process can be seen in *Figure 3*, while the hyperparameters can be seen in *Table III*.

$$\vec{c} = \vec{p_1} * fitness(p_1) + \vec{p_2} * fitness(p_2) \quad (3)$$

VIII. RESULTS

A. OPTIMIZATION

The optimization algorithms were run with the greedy algorithm in order to prevent needlessly long computation times. This is because an ideal evaluation function for a greedy algorithm is equivalent to the ideal evaluation function for a depth-2 algorithm. The Genetic Algorithm was initially run with all features with both the linear and neural network classifier. The neural network classifier was shown to clear 90-100 lines after 17 generations, as shown in *Figure 3*. Following previous work, the feature set was then limited to simply four features: totalHeight, clearedRows, numHoles, and bumpiness from *Table II* [6]. When the genetic algorithm was run on this example using a simple linear evaluation, the algorithm converges at an average of 140 lines, as shown in *Figure 4*.

The same four element feature set was used

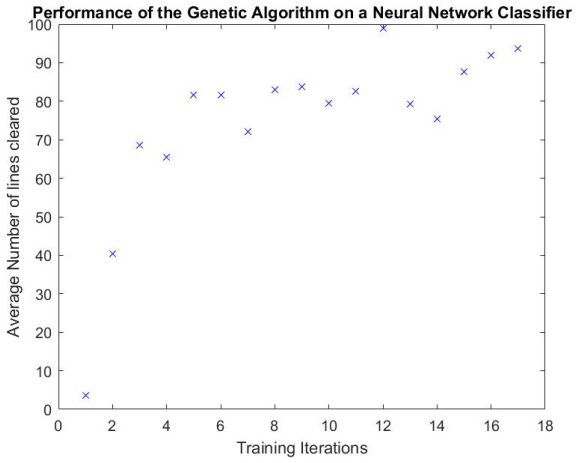

Fig. 3. Genetic Algorithm Results with all features on a neural-network evaluation using the greedy search

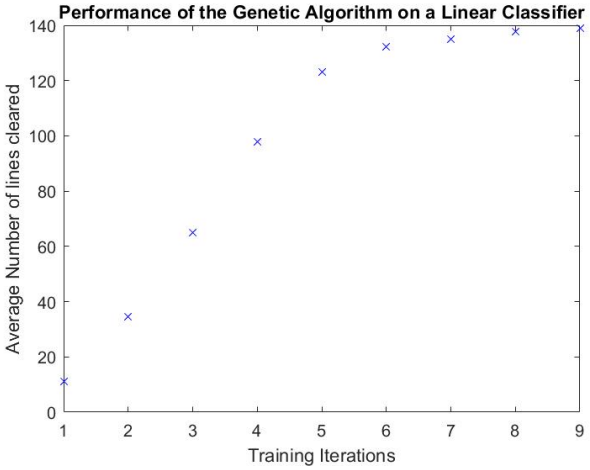

Fig. 4. Genetic Algorithm Results with a four element feature set on a linear evaluation using the greedy search

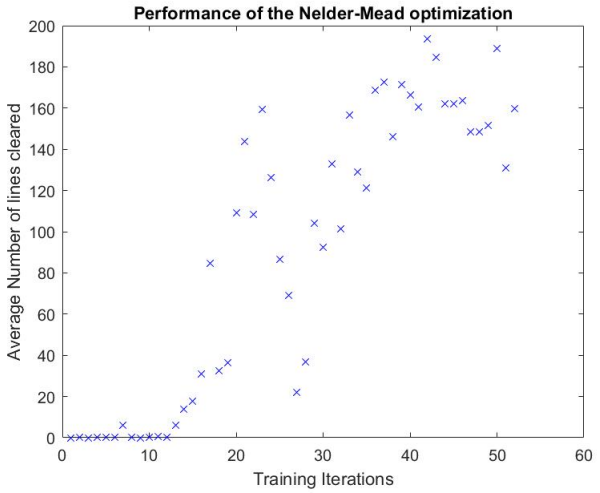

Fig. 5. Nelder-Mead Results with a four element feature set on a linear evaluation using the greedy search

TABLE IV
FINAL WEIGHTS

Feature	Weight
totalHeight	-.0006533
clearedRows	.000600263
bumpiness	-.0002195
numHoles	-.0004049

with the nelder-mead optimization with zero initial weights as well. The results of the nelder-mead optimization can be seen in *Figure 5*. From the graph, it can be noted that the optimization algorithm doesn't have a strictly increasing curve due to it converging using a 5 point polytope. The algorithm is able to reach 180-200 average lines cleared by the end of 52 iterations.

B. FINAL OUTPUT

The final weights can be seen in *Table IV*. These weights were tested in a depth-2 max search environment. During testing, it was necessary to limit the number of stones per game to 500, as the depth-2 algorithm took a very long time on average to lose a single game. It can be noted that the depth 2 results perform as well as our proposed oracle. This is because with 500 pieces, the theoretical maximum number of lines that can be cleared is 400 lines. On an unlimited stone setting, the game was able to clear up to 4000-5000 lines before losing the game.

TABLE V
SUMMARY OF RESULTS

	Greedy Search	Depth-2 Search
Mean	188.72	366.8
Median	175.5	396
Std	94.283	63.36

C. ERRORS AND LIMITATIONS

We found that although our AI agents performed extremely well after parameter optimization, we noticed that there are a few outlier games that achieve a low number of lines cleared. This result can be seen in *Table V* in which we see that the STD was at 63.36 when the game was limited to running with a maximum of 500 pieces. Some games lost very quickly because there are a few rare combinations of blocks that our AI did not seem to handle very well. Furthermore, we also experimented with how much processing our Tetris AI can do before it fails. Our current AI can solve the game at a lower bound of 50 ms per Tetris block time step (each time step is one tick of movement for the tetromino). We found that the AI could not perform its search algorithms fast enough when we increase the speed below 50 ms.

IX. FUTURE WORK

A. DEEP Q-NETWORK

Another algorithm that was explored was using a neural network to determine the best moves simply by inspecting the pixels of the board. The network used the board along with the next piece as the inputs to the neural network, with the output being a classification among all the possible key presses. The equation for the designed network is as follows, where $\sigma(x)$ is the sigmoid function:

$$key = \arg\max_i \sigma(\theta_l^T(\sigma(\theta_{l-1}^T(...\sigma(\theta_1^T\phi(x))))))_i \tag{4}$$

In this case, each position of the array i corresponds to a specific button press.

Although this algorithm shows promise, it hasn't been explored fully, and only a rudimentary model was developed. Possible future work includes developing a method to optimize and train this Q network, as well as exploring different neural networks for this purpose, such as convolutional

neural networks due to the way direct pixels are being used.

B. MAXIMIZING SCORE

Another possible future work includes attempting to generate a Tetris AI that attempts to maximize the score given a limited number of pieces. This would change the optimal strategy, since in that case it isn't preventing loss, but attempting to set up the board to allow multiple lines being cleared at once.

X. CONCLUSION

In the end, we discovered that the Nelder-Mead algorithm was extremely effective in optimizing the depth-2 max search algorithm, allowing us to solve over 4,000 lines. One of the challenges of this project was the time constraint since optimization can take many hours. Engineering good features was another tough task because we must try to deduce what features allowed the AI to stay alive as long as possible. After experimentation, we were able to conclude that the total height of all the columns, number of cleared rows associated with the action, the bumpiness of the layout, and number of holes on the board were the most influential features.

REFERENCES

[1] Chi-Hsien Yen, Tian-Li, Yu, Wei-Tze Ma, Wei-Tze Tsai, Tetris Artificial Intelligence. University of Illinois. Website: https://web.engr.illinois.edu/ cyen4/pdf/Tetris_AI.pdf.

[2] N. Lundgaard, B. McKee, Reinforcement Learning and Neural Networks for Tetris. University of Oklahoma, Website: http://www.mcgovern-fagg.org/amy/courses/cs5033_fall2007/Lundgaard_McKee.pdf

[3] Kevin Chabowski, Tetris implementation in Python, (2010), GitHub repository, Website: https://gist.github.com/silvasur/565419

[4] Bodoia Max, Puranik Arjun, Applying Reinforcement Learning to Competitive Tetris ,(2012), http://cs229.stanford.edu/proj2012/BodoiaPuranik-ApplyingReinforcementLearningToCompetitiveTetris.pdf

[5] David Rollinson, G. Wager, Tetris AI Generation Using Nelder-Mead and Genetic Algorithms. Website: https://www.andrew.cmu.edu/user/drollins/Dave%20Rollinson/16-899C%20ACRL/DB3B8757-423A -4E88-A495-EC1BB32DEE04_files/tetris_writeup.pdf

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

[7] Fink Curtis, Mathews John, Numerical Methods Using Matlab. Chapter 8, pg 430.