TetrAIs

Pranit Brahmbhatt, Sachi Patel
CS 5100: Foundations of Artificial Intelligence, Prof. Chris Amato
Northeastern University
December 12, 2023

Abstract

To play the Tetris game, this project uses five separate progressive AI agents. Using feature-based search at a depth of two, the depth-limited greedy agent searches. To train ideal weights, the reinforcement learning agent employs feature-based Q-learning. These two agents can continue playing the game indefinitely.

1. Introduction

1.1 History and How it works - Informal Definition

The intricate game Tetris is well-liked worldwide. It is played on a block space board of 22 by 10. tetriminoes, or seven sets of blocks, can be used as the figure above suggests. The game will produce a random tetrimino at every step. tetriminoes collapse continuously from top to bottom. On the board, each tetrimino occupies four squares. The player can manipulate each tetrimino's terminal position by using the arrow keys. Every line that has a block in it will be taken off the board and yield rewards. A higher reward can be obtained by clearing multiple lines simultaneously. The player's objective is to survive for the longest amount of time while clearing the lines on the board. The game Tetris goes on for an indefinite amount of time.

The goal of the artificial intelligence models and agents known as "Tetris AI" is to create computer systems that are superhuman at the game of Tetris. Tetris is a difficult game for AI since it necessitates quick decision-making and planning from the agent.

1.2 Objective and Solutions

This project aims to create an autonomous Tetris player in the form of a TetrAIs AI agent. We have used five distinct agents and various AI methods, such as greedy, depth-limited greedy search, and feature-based Q-learning, to accomplish this goal. Every strategy builds on the one before it.

Applicability in the present:

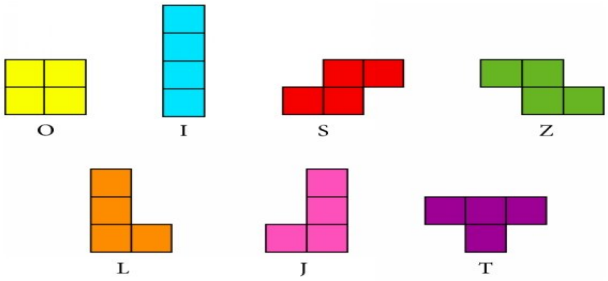

Figure 1: The Standard Set of Tetriminoes

Research on Tetris AI is still important nowadays. With millions of players worldwide, Tetris is a well-liked game, and there's rising interest in employing AI to create fresh, creative games. Tetris AI is a useful resource for academics researching machine learning and AI algorithms.

To prevail in a Tetris match, we must create agents that can:

- Clear lines quickly and effectively
- Refrain from making holes in the game board.
- Generate replicable outcomes (so we can take note of their achievements and mistakes).
- Take a video of them playing so we can review it later.

2. Approach

2.1 Reinforcement Learning:

With Reinforcement Learning, agents learn a (roughly) optimal policy in an environment by using observable rewards. This is accomplished by choosing courses of action that, given the current observed situation, maximize the expected payoff.

2.2 Q-Learning:

Q-Learning is a model-free reinforcement learning technique in which samples are used to update Q values. This is done by first computing a discount for Q value and then updating the Q value. Enough exploration combined with a

low learning rate leads to the convergence of Q values to an optimal policy.

$$sample = R(s, a, s') + \gamma * max_{a'} Q(s', a')$$
$$Q(s, a) <- (1 - \alpha)Q(s, a) + \alpha(sample)$$

2.3 Feature-based Representation:

In situations where the state space is too large for direct representation, feature-based representation is required. Under such circumstances, a vector of characteristics and a corresponding vector of weights, represented as vector w, can be used to characterize states. To get Q values, this feature-based representation is used, and weight adjustments are made every time a new sample is received.

$$Q(s, a) = w * f(s, a)$$
$$Q(s, a) <- Q(s, a) + \alpha(error)$$
$$w_i <- w_i + \alpha(error) * f_i(s, a)$$
$$error = R(s, a, s') + \gamma * max_{a'} Q(s', a') - Q(s, a)$$

3. Related Work

Examining the characteristics of Tetris and exploring its mathematical foundations are important steps in developing the Tetris agent. To clear lines and get to the game's finish is the main goal of Tetris.

3.1 Possible attributes:

Features are important elements of a state that are mapped from states to real numbers (often 0/1) [3]. As such, we list every possible Tetris aspect before developing the algorithms, which are as follows:

- Present Holes: spaces in the active game board where holes indicate locations that cannot be reached by a drop.
- Maximum Height: the game board's highest column as it currently exists.
- Total Height: the total height of the game board, is calculated by adding up each column's height.
- The lines that have been cleared are known as eliminated lines.
- Removed Lines: the lines that have been removed.
- Height Difference: the overall variation in height between two neighboring columns.
- Existence of special tetriminoes: tetriminoes in the form of a square, a line, or a "z".

Does the Stack Look Like a Skyscraper? (Denoting if the stack is getting close to the game board's ceiling).

3.2 Mathematical Basis and Proof:

In "Can you win at Tetris?", Brzustowski, [1992], and Burgiel [July 1997] present a mathematical proof stating that given a series of tetriminoes, the final tetrimination will always wind up in a well with a width of 2(2n+1), where n might be any rumination.

We will discover that the same well states will repeatedly arise when employing a winning method. This is because well states without complete cells above row 22 are limited in quantity. A successful strategy can have a maximum of $2 * 2 * ... * 2 = 2^{220}$ different states because every one of the 10*22 = 220 cells in or below row 22 is either filled or empty.

	Well Width=2		Well Width=2n, n>1		Well Width=2n+3, n>=0	
Piece	Height	No. of States	Height	No. of States	Height	No. of States
Square	0	1	2	n	(no winning strategy)	
Kink	2	2	4	2n	(no winning strategy)	
Bar	4	2	4	2n	4	2n+3
Elbow	2	2	5	2n	8	4n+6
Tee	2	3	4	3n	7	5n+5

Figure 2: An overview of successful single-piece tactics

4. Tetris NP-Hard Work Analysis

In the work analysis of Tetris NP-Hard, titled "Tetris is Hard, Made it Easy," Tetris is an NP-complete problem, more precisely, a 3-PARTITION problem with R, W, and H as its three partitions.
The Standard tetriminoes Set has seven different tetriminoes, where;

- R represents the amount of space needed for rotating and translating pieces.
- W stands for the game board's breadth, while I is equal to 4s+6.
- H, which is equal to 5T + 18, denotes the height of the bottom portion of the game board that needs to be cleared.

The work of Demaine, Ho-henberger, Liben-Nowell [6] and Breukelaar, Hoogeboom, and Walter [5] reduces instances of the 3-Partition issue to instances of Tetris where the game board cannot be cleared by defining a smaller beginning game board and a less complicated sequence of Tetris pieces.

Since Tetris is an NP-Hard issue, we are essentially limited to solving it with search algorithms like Uniform Cost Search, Depth First Search, and Breadth First Search. However using these algorithms leads to unnecessarily high time and space complexities, which renders them unfeasible in time restrictions. We will also look at comparable situations in other agents created to solve the Tetris puzzle.

5. Our System Architecture

Our system architecture is based on states that uniquely and completely capture the game states. The arrangement of these states is (board, score, block, next Block):

- Board: a two-dimensional list of integers with varying values representing different blocks and 0 signifying space.

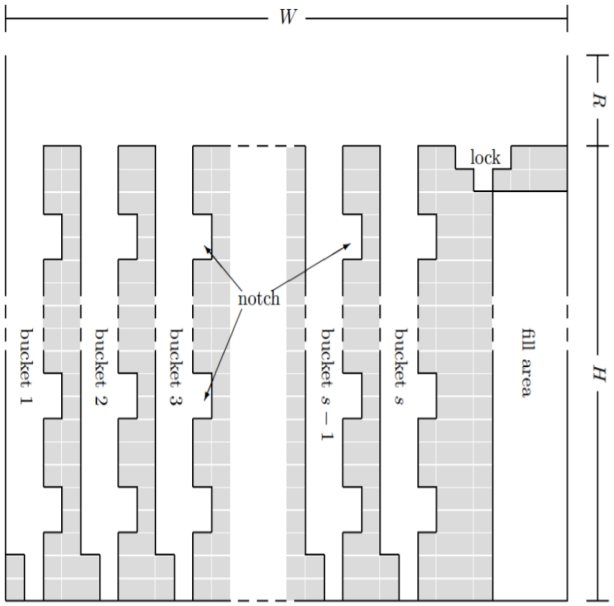

Figure 3: The initial state of a Tetris Game Board

- Score: an integer representing the game's total score as it is currently played.
- Block: a two-dimensional list of integers at the top center that does not rotate, and that represents the block that is now provided.
- Next Block: will be supplied in the next move, but it will resemble the current block.

5.1 Actions

The game's practical movements are limited to moving blocks one square at a time, but using these operations in our applications can take longer. We streamline our program actions as follows to address this: (rotateN, x):

- rotateN: An integer in 0, 1, 2, 3 that represents the block's count of rotations in a clockwise direction.
- x: The x-position for inserting the block is represented by a number in [0...maxNumCols].

5.2 Data Structures

A class implementation of our state includes a function to produce possible successor states. Different algorithms are designed to find an optimal move given the present state, acting as sub-classes of the state.

6. Implemented Agents

The five agents that we have created are the Random Agent, Minimum Height Agent, Greedy Agent, Depth-limited Greedy Agent, and Q-learning Agent.

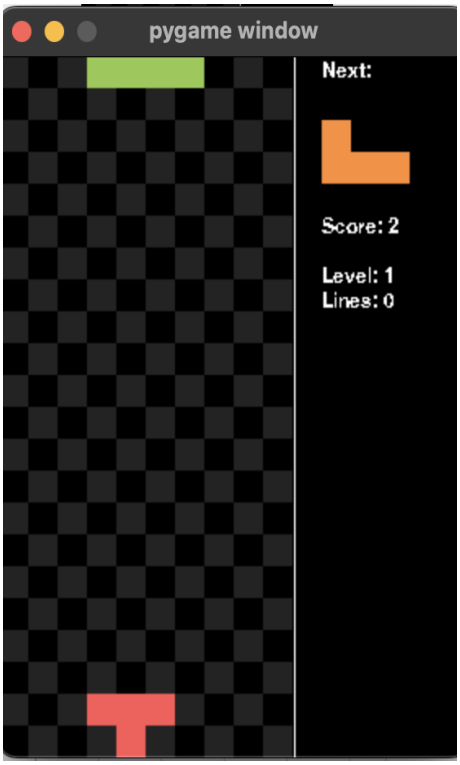

Figure 4: The initial state of a Tetris Game Board - Simplified

6.1 Random Agent

The goal is to determine the Tetris game's minimal performance standard. In our baseline, actions are selected randomly, with little thought given to the best game-play experience. The goal is to evaluate Tetris's ability to handle random tetriminoes and actions, such as random translations and rotations, given an initial game board (22 in height and 10 in width) and a series of Tetris pieces. Over 10 runs, the Random Strategy's average score is assessed. As was previously said, the random method performs poorly in Tetris because the game is NP Hard.

Assessment of the Baseline:
The average score after 10 runs shows that the baseline does not achieve the required performance, even if it aims to maximize the number of lines removed.

Average score: 24.756
Normalize variance: 0.09662975

6.2 Minimum Height Agent:

The Minimum Height Agent's main goal is to strategically install tetriminoes at the lowest area on the game board. As opposed to using a random agent, this method improves Tetris performance. To find empty spots for tetriminoes, the algorithm repeatedly scans each row from top to bottom and from left to right inside each row.

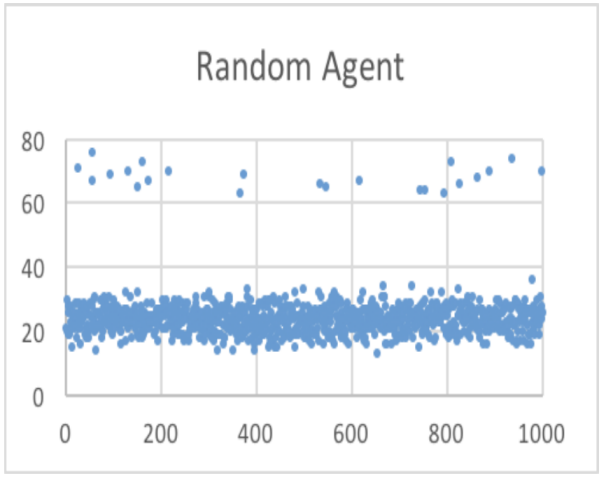

Figure 5: Score Distribution for Random Agent

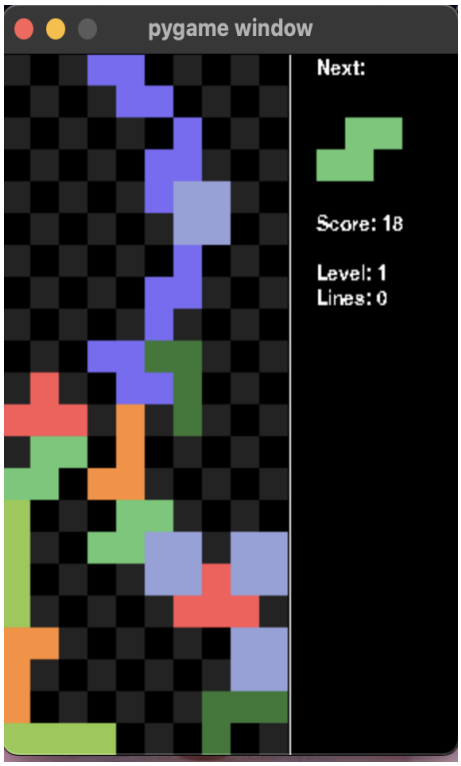

Figure 6: Random Agent Playing Tetris

First, the first tetrimino appears at the left bottom of the board according to the algorithm. Then, as shown in Figure 6.3, which depicts the typical condition of Minimum Height, it places the second tetrimino in the middle until the first line is full. This technique guarantees that every tetrimino is positioned as low as feasible, without taking into account movement modifications after reaching the board's end to prevent misplacing the tetrimino.

An analysis of space complexity
When tetriminoes reach the bottom of the board, ideal outcomes are taken into consideration as part of the investigation of space complexity in coding. Even with its seeming benefits, this kind of coding has drawbacks. Checking tetrimino conflicts with the existing board after every move is necessary to reach the minimal height. Nevertheless, taking into account every scenario for every tetrimino leads to a time and space complexity that is too great for quick solutions.

Tetriminoes are dropped in a straight line with a single spin in a simplified method, which results in $O(n^2)$ time and space complexities. Investigating each rotation at every place increases the complexity over $O(2^220)$, which is not feasible for our algorithm. The complexity problem is clear when one considers the seven different tetrimino kinds, all of which require a minimum of two rotations, and the $22 * 10$ position game board.

Average score: 58.705
Normalize variance: 0.334299294

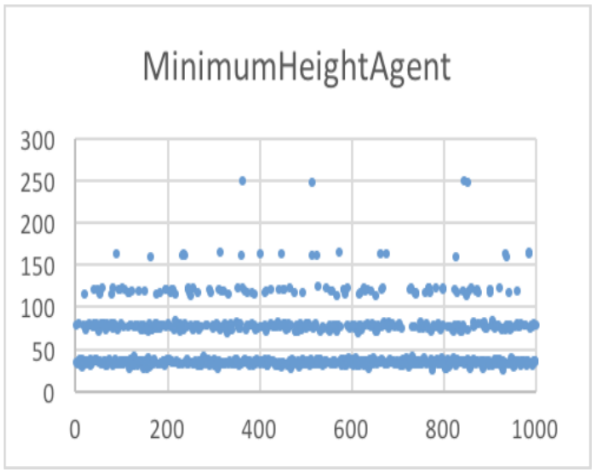

Figure 7: Score Distribution for Min. Height Agent

Analyzing the Random Agent to the Minimum Height Agent
The Random Agent is constantly beaten by the Minimum Height Agent after ten runs. Trying to maximize the number of lines cleared before the game ends, however, reduces its effectiveness.

6.3 Greedy Agent

Using a linear function to determine the value for each state, the Greedy Agent uses the optimal value in the subsequent

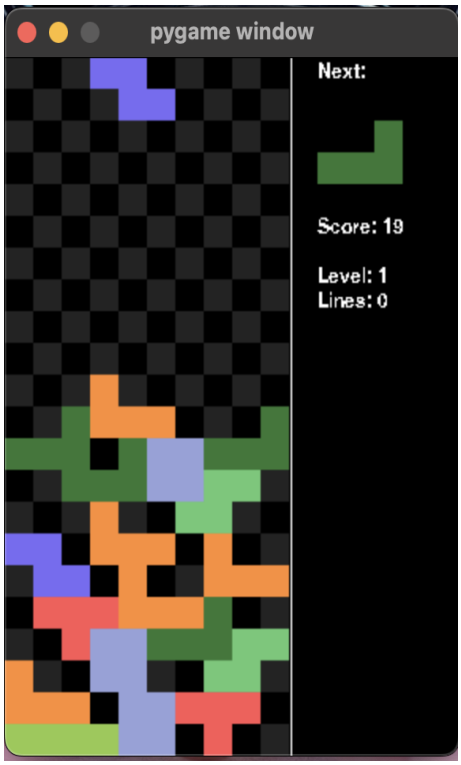

Figure 8: Min. Height Agent Playing Tetris

phase to choose the best course of action.

Important inquiries:

- What makes a move the best?
- Which characteristics are essential to determining the state's value?
- Which features have what weights attached to them?

Optimal Movement Considerations:

- Make clearing lines a priority, especially when doing so for several lines at once.
- Filling blocks above vacant squares should be avoided if line clearance is not possible.
- Reduce oscillations to prevent hills from forming.

We incorporate the following features to meet these standards.

Feature Selection:

Maximum Height:
The height of each column is indicated by the highest filled square in that column. Reduced maximum height is a feature because it suggests an easier game condition.

Total Height:
For a thorough evaluation, the total height—which is the sum of the heights of all columns—is computed beyond the maximum height.

Holes:
Because they are difficult to clear, empty squares with full squares above are seen as undesirable in a state.

Cleared Lines:
Removing a line results in a score (40 for a single line, 100, 300, 1200 for 2, 3, and 4 lines), which affects the state's advantage.

Delta:
When tetriminoes build up in a single column, the total of the absolute delta values between the heights of the surrounding columns helps.

The delta in this instance can be calculated using the formula:
$|2-2|+|2-4|+|4-4|+|4-5|+|5-4|+|4-4|+|4-4|+|4-3|+|3-0|=8$.

The agent will attempt to insert tetriminoes in columns with lower heights when the delta is lower.

Weight Adjustment:
When reinforcement learning is not used, weights are manually assigned.

- Maximum Height: Started at -1 with a reduced maximum height priority.
- Total Height: Starts at -1, highlighting a shorter height overall.
- Holes: Originally assigned a score of -2, emphasizing the significance of reducing holes.
- Lines Cleared: Starts at 2, recognizing its substantial effect on the score.
- Delta: Commences at -1, signifying its inverse relationship to the score.

Analysis of Space Complexity
We just take one step ahead of the existing greedy agent. As seen in Figure 1: We have seven distinct tetriminoes in the Standard tetriminoes Set [4], and component 4 takes R, W, and H into consideration. Additionally, the height and length of the game board are significant to these variables.

Each Tetrimino has four distinct rotations in the program, and ten columns total, based on the width of the board game. Consequently, both the space complexity and the temporal complexity are $O(n^2)$.

When using a greedy agent, such as a Minimum Height Agent, we additionally need to verify that the given tokens still correspond to the existing board after every move. For the same reason, we don't take into account every tetriminoes scenario.

Analyzing the Greedy agent
After 1000 runs, the Greedy Agent shows improvement over baseline agents, but it still can't solve the Tetris problem well enough, especially when it comes to maximizing the number of lines cleared.

Average score: 6883.101
Normalize variance: 0.317827668

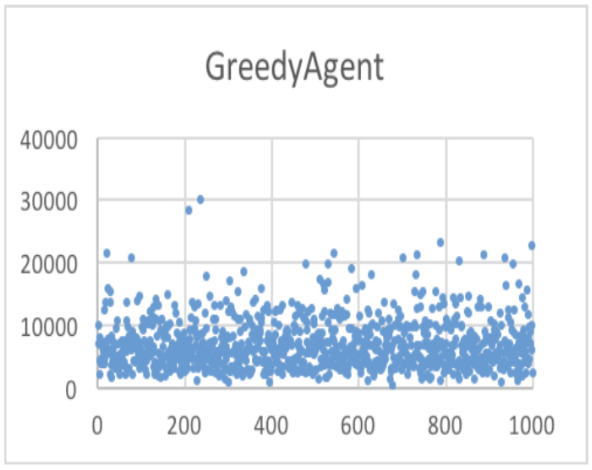

Figure 9: Score Distribution for Greedy Agent

6.4 Depth-limited Greedy Agent:

To make the Greedy Agent more efficient, we decided to use the information about the Tetris pieces that come after this one.

Given: A finite series of Tetris pieces and a first game board that is 10 units wide and 22 units high, the following questions are asked:

- How can we determine the best move in light of the next Tetris piece sequence?
- How far ahead of time can we project our analysis?
- How does this agent compare in performance to the Greedy Agent?

The following procedures must be followed to determine the best move given the arrangement of the remaining Tetris

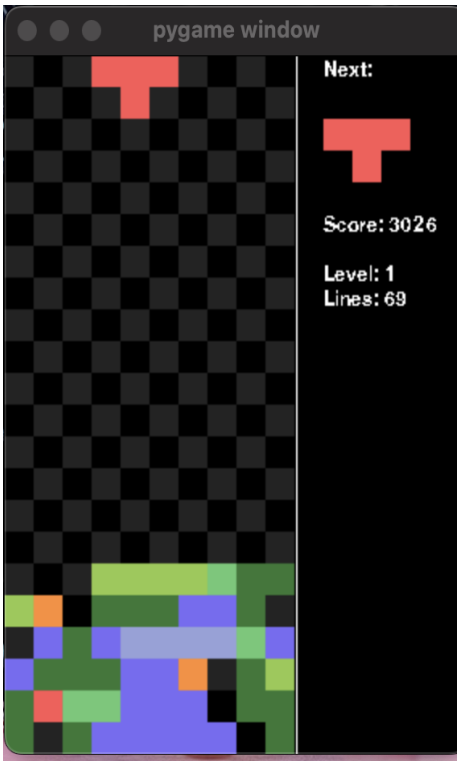

Figure 10: Greedy Agent Playing Tetris

pieces:

- Applying the Greedy Agent's feature evaluation method.
- List all possible states after the present piece and the next N pieces in the sequence are placed.
- Evaluating every possible state with the same characteristics as the Greedy Agent.
- Carrying out the plan that would result in the state receiving the highest evaluation score.

To project future steps, an action is a tuple (rotateN, x), which results in 40 different actions for every state (4 rotations and 10 horizontal locations), as explained in the System Architecture. Thus, the time complexity for a search with depth N is $O(K * 40^N)$, where K is the expected time complexity of evaluating the states (220). Limiting N to a maximum of three actions, and completing one action in less than a second is essential for effective action selection.

To evaluate this agent's effectiveness in comparison to the Greedy Agent, we created a depth-limited search agent with a depth of 2. We demonstrate that our depth-2-limited search agent can play continuously without any time constraints using the same state assessment features and weights.

6.5 Q-Learning Agent:

Earlier algorithms computed scores based on the attributes of a given state by using explicitly provided weights. To

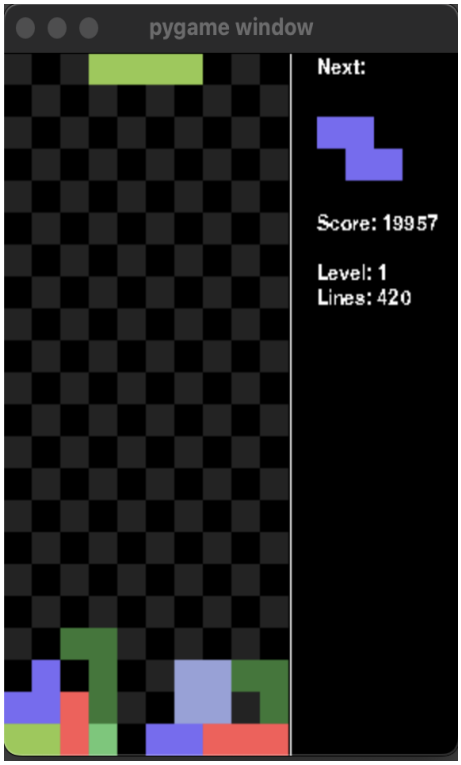

Figure 11: Depth-limited Greedy Agent Playing Tetris

further our methodology and enhance game-play, we have made the switch to using Q-learning to train the weights corresponding to particular features.

In our training program, Q(s, a) is the Q-value obtained from a linear function that incorporates characteristics at the next state, and R(s, a, s') is the number of lines cleared from state s to s'.

$$Q(s,a) = w * f(s,a)$$
$$w_i <- w_i + \alpha(error) * f_i(s,a)$$
$$error = R(s,a,s') + \gamma * max_{a'} Q(s',a') - Q(s,a)$$

Initial Training Outcomes:
The results were not up to par even after using the identical characteristics and feeding manually adjusted weights into Q-learning. The Q-learning agent's performance was insufficient to solve the Tetris puzzle successfully. After some tetriminoes were arranged in a tower-like pattern without clearing any lines, the underlying problems were critically examined.

Examining the update process in detail revealed that the weights were being updated far too quickly. Some weights had already increased to 1.0E+10 after only three episodes. Features like total heights, which have the potential to produce incredibly high values and have a disproportionate impact on later assessments, were blamed for this phenomenon.

Feature Refinement:
To tackle these issues, a thorough reevaluation of every feature was carried out, which resulted in the subsequent modifications:

- Maximum altitude $->$ Delta Maximum Height.
- Delta $->$ Difference in the deltas of two states.
- Holes $->$ Delta Holes.
- The entire height metric is eliminated.

	Lines Cleared	Total Height	MaxHeight	Holes	Deltas
Initial	4	-2	-1	-1	-1
episode 1	3.881490037	5.149721621	6.587757221	-35.66024158	0.635713852
episode 5	1.05E+59	1.21E+120	-1.30E+121	4.38E+121	-5.46E+122

Figure 12: Q-Learning Analysis

7. Data Analysis

We gather information and thoroughly analyze the statistics to evaluate the performance of different algorithms.
Normalized Variance: We first compute the variance after normalizing each data point by dividing it by the corresponding average.

What We Expect:

- Random Agent: It should have a distinct one-line structure and be extremely stable, never exceeding a score of 40.
- Minimum height agent: Very steady but unlikely to get above 100.
- Greedy agent: A greedy agent is likely to show notable variations and may even receive high ratings at times.
- Depth-limited greedy agent: Probably going to do better than the greedy agent, with steady performance.
- Q-learning agent: With steady variance, expected to outperform the greedy search.

Actual Results:

- Random Agent: It shows a low average score but is stable, as expected. Minimum height agent: Significantly deviates from expectations. Notably, more than five lines can be cleared in some unique instances.
- Minimum height agent: Shows a significant deviation from expectations. Notably, more than five lines can be cleared in some exceptional instances.

- Greedy agent: The results show an acceptable normalized variance and, surprisingly, deviate from expectations. Unexpectedly, one outstanding sample received a score of 30000.
- Depth-limited greedy agent: This agent surprisingly outperforms predictions, exhibiting almost limitless survival duration when given the right weights.
- Q-learning agent: Achieves satisfying limitless game-play by optimizing weights.

Analysis:
The agent randomly assigns tetriminoes to the baseline, making it nearly impossible to clear a line. The constant results of the algorithm are explained by this behavior.

Significant progress has been made when evaluating the minimum height agent, although a high variance indicates that luck is still quite important. The hunt for a more reliable algorithm was prompted by the data, which displays a score range of 20 to 300.

With its weighted decision-making, the greedy agent shows a significant increase in average and variance. The algorithm produces scores between 20,000 and 30,000, indicating that it is competent in the majority of scenarios but not so much in others.

Subsequently, the depth-limited greedy agent exhibits extended survival and yields satisfactory results.

On the other hand, the depth-limited greedy agent eventually dies if it is run for a long time. Nonetheless, the training result surpasses 10,000 lines, almost resembling an endless game. This is a noteworthy accomplishment that emphasizes how much optimized weights affect survival duration.

Our developmental model is consistent with expectations when all data and video results are taken into account. The diagram shows a steady increase in the agent's performance, starting from the baseline and progressing through variants.

Average Score	Random Agent	Minimum Height Agent	Greedy Agent
10	24	51.6	7143.3
100	25.93	56.92	7473.21
1000	2.48E+01	5.87E+01	6.88E+03

Figure 13: The Agents' Average Score

Average Score	Random Agent	Minimum Height Agent	Greedy Agent
10	0.25347222	0.163316507	0.108576377
100	0.142430249	0.463788158	0.336619846
1000	9.66E-02	3.34E-01	3.18E-01

Figure 14: The Agents' Normalized Variance

8. Conclusion

Finally, as a result of our research, we have successfully constructed and compared a number of AI agents for Tetris, specifically:

- Random Agent
- Minimum Height Agent
- Greedy Agent
- Depth-limited Greedy Agent
- Q-learning Agent

Our analysis of these different agents' Tetris games yields some interesting conclusions. Even with a few restrictions, the Depth-Limited Search Agent performs admirably overall, scoring highly and exhibiting extended gaming capabilities. It is noteworthy because it demonstrates an exceptional capacity to play the game effectively for prolonged periods of time.

The Depth-Limited Greedy Agent efficiently arranges the best course of action given the available information in the Tetris dynamics setting, where the next tetriminoes are known while addressing the current ones. Remarkably, this agent performs well even at a depth of two, completing Tetris puzzles with skill. We've eliminated more than ten thousand lines with the Depth-Limited Greedy Agent, which is our best achievement to yet, without sacrificing game-play.

Moving on to another notable agent, Q-Learning Agent seems as a strong candidate. After enough practice, it achieves competitive game-play and high scores. However, it is imperative to recognize that preliminary poor decisions could impair its functionality because of inadequate information. Even though it faces some early difficulties, the Q-Learning Agent soon overcomes them and demonstrates its capacity for reliable and skillful games.

Furthermore, the Q-Learning Agent consistently performs at a high level for prolonged periods. It's also critical to remember that the weights given to the various features have a significant impact on how effective the Greedy Agent is.

Like a simple Reflex agent, the Minimum Height Agent detects the current board's lowest height and puts tetriminoes strategically in accordance. As a result, it performs better than the standard Random Agent.

To sum up, our Depth-Limited Greedy Agent, which plans ahead strategically, and the Q-Learning Agent, which can produce impressive performance after training, are excellent options for attaining excellent game-play in the Tetris setting.

10. Work for the Future

In the later stages of our project, we don't change the Tetris game's difficulty settings. The tetrimino descent speed increases proportionately with increasing difficulty. The tetrimino travels down one block every 1000 milliseconds at the lowest level and every 100 milliseconds at the highest. Remarkably, every agent exhibits the highest degree of competence—with the exception of our Depth-Limited Greedy Agent. But in order to execute in a reasonable amount of time, the Depth-Limited Greedy Agent requires a more sophisticated Central Process Unity. It is necessary, though a very big task, to address deficiencies in the framework architecture.

Apart from the agents we describe in our study, a lot of work has gone into creating a Genetic Agent specifically for Tetris. For best results, many test cases just require more computational cycles than others. Because of this, to make effective use of computing time, programs need to dynamically distribute cycles among test cases [7]. By utilizing existing features, the Genetic Agent outperforms our Depth-Limited Greedy Agent in terms of efficiency. Most notably, it significantly reduces the running time, which is an essential feature for extended operation.

The most powerful motivator is the attraction of curiosity. One can feel a sense of success from seeing the Tetris AI agent work, but creating a graphical user interface (GUI) to watch two agents fight strategically is an even more entertaining method. In addition, the Instant Drop function for tetriminoes is the only thing that our present system takes into account. Raising the subject of whether Soft Drop considerations could result in the development of a perfect AI agent piques real curiosity.

The following activities are included in the upcoming improvements to our Tetris project, to summarize:

- Enhancing the structure and fixing the underlying structural flaws.
- The Tetris-specific Genetic Agent has been completed.
- The Tetris Agent's ideal features are identified.
- Investigating whether or not the Tetris Agent could benefit from Soft Drop considerations.

11. Appendix

11.1 Synopsis and Terminology

:

- Game Board: In our project, the Tetris playing field is a matrix with 22 rows and 10 columns.
- tetriminoes: The essential components of the game that are generated at random and used to facilitate game-play are called pieces. The score is increased by one point for each tetrimino that is played.
- Cleared Line: A row on the game board where tetriminoes have taken up all 10 squares. These lines are quickly erased, which causes the lines above them to descend downward and take up the empty area. The number of lines cleared concurrently determines how many points are awarded for each cleared line.
- Soft Drop: If there are no impediments in the way, players are allowed to move a piece horizontally for a certain number of steps after it has landed on top of the other pieces. Notably, we decided not to apply this criterion to simplify the state transitions in our project.
- Hard Drop: Any more horizontal movement of the component is prohibited after it descends upon the preexisting pieces. This is the regulation that we follow in our operations.

11.2 Detailed Description of our Methods:

'tetris.py'

This file contains the main Tetris game implementation along with a user-controlled Tetris agent.

Constants and Data Structures

- cellSize, cols, rows: Size of a cell, number of columns, and number of rows in the game.
- maxFps: Maximum frames per second for the game.
- colors: RGB values for different block colors.
- tetrisShapes: List of Tetris block shapes.

Functions

- rotateClockwise(shape): Rotates a Tetris block clockwise.
- checkCollision(board, shape, offset): Checks if a block collides with the walls or existing blocks on the board.
- removeRow(board, row): Removes a completed row from the board.
- joinMatrices(mat1, mat2, $mat2_off$): Joins two matrices.
- newBoard: Creates a new Tetris game board.
- TetrisApp: Class representing the Tetris game. Handles user input, game state, and rendering.

‘randomAgent.py‘

This file contains an agent that plays Tetris by making random moves.

- The agent rotates the current stone a random number of times (up to 3 times).
- It then randomly moves the stone to the left or right (up to 5 times).
- Finally, it drops the stone instantly.

MinimumHeightAgent.py

This file contains an agent that plays Tetris by placing blocks at the minimum height on the board.

- The agent looks for the column with the lowest filled cell.
- It then moves the current stone horizontally to align with that column.
- Finally, it drops the stone instantly.

GreedyAgent.py

This file contains a class Greedy that represents a Tetris agent using a greedy strategy based on certain features. The class TetrisGreedy extends TetrisApp and uses the Greedy class to make moves.

- The agent uses a greedy strategy to evaluate potential moves.
- It considers all possible rotations and horizontal positions for the current stone.
- For each combination, it evaluates the resulting state based on various features such as lines cleared, total height, max height, holes, and height differences.
- It selects the move that leads to the highest evaluation score.

Depth-limited Greedy Agent

This file contains a class depthLimitedGreedy representing a Tetris agent with a depth-limited greedy strategy. The agent evaluates the best move by looking ahead at a limited number of moves. The class TetrisGreedy extends TetrisApp and uses the depthLimitedGreedy class to make moves.

- Similar to the Greedy Agent, but it only looks one step ahead.
- It evaluates each possible move by considering the immediate next state's evaluation score.
- It then selects the move that leads to the highest score.

Q-Learning Agent

This file contains a class QL representing a Tetris agent using Q-learning. It uses a set of weights to evaluate the features of the current state. The class TetrisRL extends TetrisApp and uses the RL class to make moves.

- The agent uses Q-learning to learn a policy for selecting moves.
- It maintains a set of weights for different features (lines cleared, total height, max height, holes, and height differences).
- During the game, it updates these weights based on the reward obtained from the current move.
- It selects moves either randomly (exploration) or based on the learned Q-values (exploitation).

state.py

This file contains the State class representing the state of the Tetris game. It provides functions for generating the next states based on actions and evaluating the state based on certain features.

- Defines the game state, including the game board, score, current block, and the next block.
- Provides methods for generating possible next states based on different actions (rotations and horizontal movements).
- Implements collision checking, dropping blocks, adding blocks to the board, and scoring.

12. References

1. The official website of Tetris
2. Tetris is hard, made easy
3. Berkeley Pacman Project
4. Siegel, E., and A. D. Chaffee (1996). Genetically Optimizing the Speed of Programs Evolved to Play Tetris. Advances in Genetic Programming, Volume 2.
5. Playing Games with Genetic Algorithms
6. review on genetic algorithm: past, present, and future
7. E.D. Demaine, S. Hohenberger and D. Liben-Nowell, Tetris is Hard, Even to Approximate, Technical Report MIT-LCS-TR-865 (to appear in COCOON 2003), Laboratory of Computer Science, Massachusetts Institute of Technology, 2002.
8. Ron Breukelaar, Hendrik Jan Hoogeboom, and Walter A. Kosters. Tetris is Hard, Made Easy, Liden Institute of Advanced Computer Science, Universiteit Leiden, 2002
9. Search Based Artificial Intelligence Program for Autonomous Tetris Game-play, Computer Science 182, Harvard University 2013
10. John Brzustowski's "Can you win at Tetris?" Master's thesis, University of British, Columbia, 1992.

Link to GitHub repository for the code and demo of this project.