A Vo

Professor Wong

Artificial Intelligence (CS 4100)

10 December 2021

Project Report – Tetris

Abstract

This paper presents the results of experiments carried out with the purpose of applying various machine learning techniques (of reinforcement learning) to the game of Tetris. We use reinforcement learning techniques to train weights used to calculate the importance of features in primarily Dellacherie's Algorithm, an optimal one-piece (no look-ahead) heuristic algorithm for the game of Tetris. Using said algorithm with trained weights, the reinforcement agent is able to select the best possible next course of action given its current state to maximize its reward. That said, we also attempted to implement other RL-learning algorithms to compare their performance.

Introduction to the Project

This project was proposed roughly a month ago as a Capstone Project and final project for the course CS4100 – Artificial Intelligence at Northeastern University. While the primary motivation for this endeavor is to apply and showcase our understanding of machine learning and artificial intelligence concepts, research into Tetris algorithms and strategies has been an interesting application of classroom-taught techniques.

Introduction to the Game

Tetris is regarded as a popular game that most people understand how to play or can understand relatively quickly. Given a rectangular board that is 10 cells wide and 20 cells high, the player must organize falling pieces (S, Z, L, J, I, and O pieces) into specific locations to prevent the user from toppling over the top of the board. To avoid such, the main gameplay mechanic revolves around creating full rows with the falling pieces in order to "clear" (delete) said row. With careful placement strategies and a sequence of Tetris pieces that are not impossible to solve (for example, an S-Z repeating sequence will eventually topple over unless the board dimensions are a multiple of 4), a game of Tetris could run indefinitely.

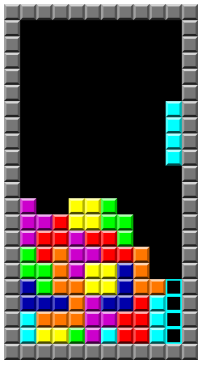


Figure 2 - Tetris Game Board

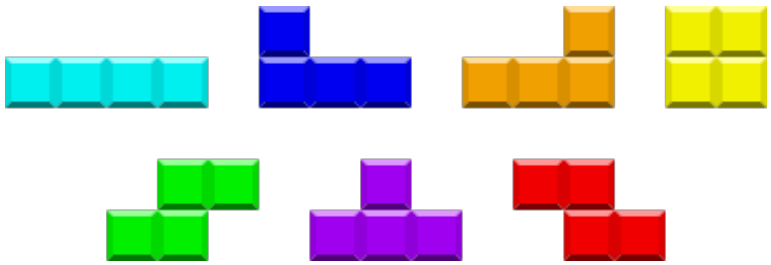


Figure 1 - Tetris Pieces

Introduction to the Game – Variations of Tetris

From a high level, the proposed problem is to formulate a Tetris agent that is capable of playing indefinitely. It is important to note here the distinction between different versions of Tetris: Tetris Guideline (Modern) and classical Tetris.

The Modern Tetris Guideline is the standard for Tetris used in virtually every Tetris game created in the past two decades (2001 and later). The main features (aside from a new rotation system) that are relevant to this project is enhanced look-ahead, hold, and the bag system.

Look-ahead allows the user to view the next one to many pieces which can be used to set up various Tetris stacking techniques such as T-Spin factories and 4-Wide setups. On the other hand, classical Tetris only allows for zero to one piece look-ahead which limits the amount of planning ahead for both a human and a computer agent and introduces some probability (expectimax) into the system.

The hold queue allows a player to swap the current falling piece with the piece currently the hold queue, acting as a storage and crutch for players to better stack pieces. For example, if the current falling piece can only create holes in the game board, then it might be better to swap to the hold piece which could be placed more optimally. Modern Guideline Tetris has a hold queue whereas classical does not.

The final significant difference is the bag system, which determines the order at which pieces enter the board. A bag system enforces that every piece is seen at least “x” times before a piece may re-enter the queue. For example, a “7-bag” requires every piece to be seen at least once before allowing for a piece to show up again. This, combined with the hold queue, should allow the player to play indefinitely, which is why it is a major component of Modern Guideline. On the other hand, classical Tetris uses a random piece generator, meaning that there is no promise of any order. As a result, it can be common for “droughts” to show up (usually the I piece) which can inevitably cause a game to end.

From a technical perspective, classical Tetris should be easier to develop for as it has less features to take account of. For example, the agent will not have to analyze a decision tree if their hold piece or predict the next couple of pieces given the past seven pieces. However, not only are these all major components of Modern Guideline, but these are the ways a well-trained Tetris player will view a Tetris board. Thus, it may become an interesting challenge to take on in the future. However, I have decided to take on a variation of classical Tetris.

Problem Formulation

Given the algorithms I will be using, it would be best for formulate Tetris as an MDP.

The set of all possible states is the set of all possible board combinations formed by the Tetrimino pieces, which is roughly 7^{200} possibilities.

The set of all possible actions would be a clockwise or counter-clockwise rotation, left and right piece movement, as well as fast drop (though, for the sake of our program we will most likely not need to account for this).

The transition function, given a state, will determine the probability of the next state. This will be affected by our learning algorithms in choosing an optimal move.

The reward function would be a score based on how many pieces have been placed in the game so far. In addition, the reward function will yield a larger reward for double and triple line clears as well as Tetrises. Thus, the agent will be rewarded based on how well they perform.

Methods – Optimizations to the Problem Formulation

This problem formulation is quite unwieldy and may be quite slow in regards to calculating based on raw state data that represents the board as a whole. In addition, if each action simply moves the piece, then there will be many states between each successful placement of a piece which is also a significant waste of resources.

From my past month of research, I have found two promising articles and projects that I plan on utilizing and implementing to create my own Tetris agent. The first optimization I can make is to use features (generalizations) to represent a state. From the article "Reinforcement Learning and Neural Networks for Tetris" by Lundgaard and McKee[1], they propose a higher-level representation of individual Tetris states. Specifically, we can use the following seven features:

Feature	Description
Current Piece	The shape of the current falling piece.
Next Piece	The shape of the piece in the next queue.
Board Height	Average height of the board (0 to 20)
Board Level	Boolean – checks if height levels are even
Has Single Width Valley	Boolean – is there a spot for an I piece
Has Multiple Valleys	Boolean – multiple I piece spots (suboptimal because could build a tower)
Number of Buried Holes	Number of empty cells covered by filled cells above in a column

The new state-space would be: $7^2 * 20 * 2^3 * 5 \approx 40000$ states, far better than 7^{200} now that we only track necessary data.

[1] https://mcgovern-fagg.org/amy_html/courses/cs5033_fall2007/Lundgaard_McKee.pdf

In my implementation, I am using a pre-built package of the gym titled "gym-simplifiedtetris"[2] which aims to reduce the amount of states through a simplification of the game's core mechanics. Rather than manually moving a piece to a spot, this OpenAI Gym environment chooses where to place a piece and instantly places it there, ignoring all the states in between that are useless to process and additional overload to calculate.

Methods and Algorithms

Currently, I have three implemented learning algorithms. One working method is not so much a method and more just a test to see if the environment works as this agent places pieces randomly.

The second method I am using is called Dellacherie's Algorithm[3]. This is a one-piece no look-ahead algorithm (meaning no next queue is necessary, fully functional with classical Tetris). It operates purely on the features of a state:

Feature	Description
Landing Height	The height where the latest piece is put.
Eroded Pieces/Rows Eliminated	(# rows eliminated) * (# cells the piece contributed to eliminating the rows)
Board Row Transitions	The total number of row transitions. A row transition occurs when an empty cell is adjacent to a filled cell on the same row and vice versa.
Board Column Transitions	The total number of column transitions. A column transition occurs when an empty cell is adjacent to a filled cell on the same column and vice versa.
Board Buried Holes	Number of empty cells below at least one filled cell in the same column.
Board Wells	Amount of empty cells in succession where their left and right cells are filled.

which follows the formula (where each variable w corresponds to a weight):

$$w1(landingHeight) + w2(rowsElim) + w3(rowTrans) + w4(colTrans) + w5(holes) + w6(wells)$$

From this algorithm, I will be able to assign a value to each state, and call argmax to select the most optimal state. In the event of two or more states having identical weights, I have created a prioritization function that tie-breaks, opting for pieces further away from the center as well as unrotated. The logic behind this is that since pieces spawn in the center, it is safer to build on the sides. At the same time, it is faster to place pieces if they are not rotated.

[2] https://github.com/OliverOverend/gym-simplifiedtetris
[3] https://mcgovern-fagg.org/amy_html/courses/cs5033_fall2007/Lundgaard_McKee.pdf

One issue with this method Dellacherie's algorithm (and any linear approximation function) is that I need to calculate weights and calculating weights on six features is quite the computational hurdle.

To calculate these weights, I have implemented a genetic learning algorithm. This learning system assigns a randomized weight vector (6 elements) and adjusts future weights based on the success of their parents. Specifically, for each generation, the top 20-percent of parents are used to produce children, where the weight is a 50/50 choice from either parent. On top of that, there is a 20-percent chance of a weight being mutated which would add a random value onto the weight.

Unfortunately, this method is either very taxing or slow on computer hardware. This is because the algorithm needs many iterations over multiple generations to truly get at a proper weight vector that makes sense. When running this algorithm on my computer, I tested this with two Tetris board configurations: 10x10 and 20x10. Running 25 epochs at where each epoch has a population of 25, it takes the 10x10 roughly 30 to 40 minutes to finish running whereas I could only run through 5 epochs on 20x10 before my computer crashed (and even that took about 2 hours).

In addition, I found that the genetic learning algorithm did not produce the results that I was satisfied with. I believe the main issue is that the algorithm seems to settle for less, and once there is a generation where the top 20-percent of scores are "sub-optimal" (less than 300 points), the weights will no longer change as much, resulting in similar scores. Thus, the only way to break out of this herd majority would be through the 20-percent chance of mutation, but in 25 epochs was not enough time for anything substantial to happen.

The third method I implemented is titled "Building Controllers for Tetris[4]", or the BCTS algorithm. This algorithm has eight features, where the first six are the same as Dellacherie's algorithm:

[4] https://hal.archives-ouvertes.fr/hal-00397045v2/document

Feature	Description
Landing Height	The height where the latest piece is put.
Eroded Pieces/Rows Eliminated	(# rows eliminated) * (# cells the piece contributed to eliminating the rows)
Board Row Transitions	The total number of row transitions. A row transition occurs when an empty cell is adjacent to a filled cell on the same row and vice versa.
Board Column Transitions	The total number of column transitions. A column transition occurs when an empty cell is adjacent to a filled cell on the same column and vice versa.
Board Buried Holes	Number of empty cells below at least one filled cell in the same column.
Board Wells	Amount of empty cells in succession where their left and right cells are filled.
Hole Depth	The total number of filled cells on top of each hole
Row Holes	The total number of rows with at least one hole

The reason for implementing this is because it is claimed that the BCTS algorithm performs better than Dellacherie's algorithm[5].

For the future, I found one more algorithm that represents a state using four features:

Feature	Description
Aggregate Height	Sum of heights of each column.
Rows Eliminated	Number of columns eliminated so far.
Board Buried Holes	Number of empty cells below at least one filled cell in the same column.
Bumpiness	The variation in column heights. Computed by summing the absolute difference between all two adjacent columns.

which follows the following formula:

$$w1(aggHeight) + w2(rowsElim) + w3(holes) + w4(bumpiness)$$

With two less weights to train for, this may be computationally more effective for training speed. However, I am very certain that Dellacherie's Algorithm is better (in regards to survivability when given sub-optimal piece scenarios) given the data I have seen online. While I do not have time to implement and train this algorithm at the moment, it is something I will try implementing in the future.

[5] https://arxiv.org/pdf/1905.01652.pdf#cite.Boumaza2009 (page 5)

Concrete Technical Problem Statement

For each agent to function, it must take in the grid (state of the board). Some may also require the action space, but I noticed that the action space is always the same for an _ by 10 size board, 34 possible actions (one for each column multiplied by the four orientations minus overlap). The output of an agent is always the same, the optimal action to take.

References to Other Works

Aside from reading many other online blog posts and research papers, I am utilizing a GitHub repo from OliverOverend titled “gym-simplfiedtetris”. This package aims to reduce the complexity of the game by instantly placing the pieces where they should go rather than calculating the intermediary steps between placement. By itself, it provides a Tetris environment built on OpenAI Gym that I can use to train. The author also included some code to several learning agent formulas, but I will be writing my own implementation of these (as that is the entire purpose of this project).

My main concern with this repo was the lack of state information. The method info() returns a one-element dict that contains the number of rows cleared so far which is just a part of each algorithm. To circumvent this, I created property (getter) methods from the Tetris engine in (./gym_simplifiedtetris/envs/simplified_tetris_engine.py and ./gym_simplifiedtetris/envs/simplified_tetris_base_env.py) which I can directly access and pass to my learning agents. With this information, I can easily calculate the properties and values of the features proposed above.

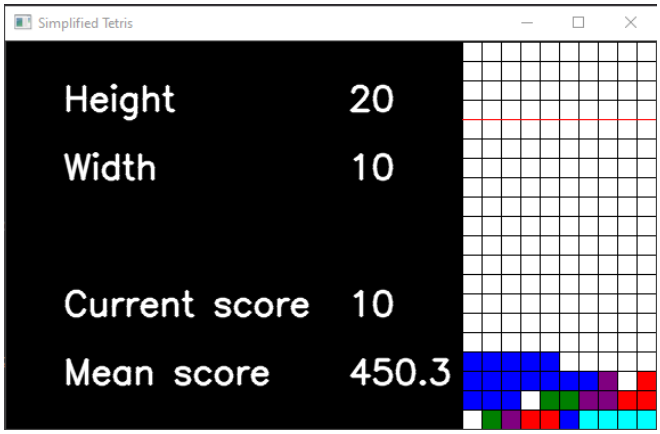
The genetic learning algorithm is based on a “Towards Data Science” post called “Beating the world record in Tetris (GB) with genetics algorithm”, which implements a genetic learning algorithm on a Game Boy version of Tetris. Much of the logic is the same as the algorithm takes purely the score as input and outputs a vector of weights (in the form of a NumPy array) which can be passed to an agent.

Results

For this project, I have tried approaching this in two ways: genetic learning results and pure score results from the learning algorithms.

For score results, I let the BCTS and Dellacherie algorithms run for 10 iterations:

The BCTS algorithm yielded a mean score of 450.3.

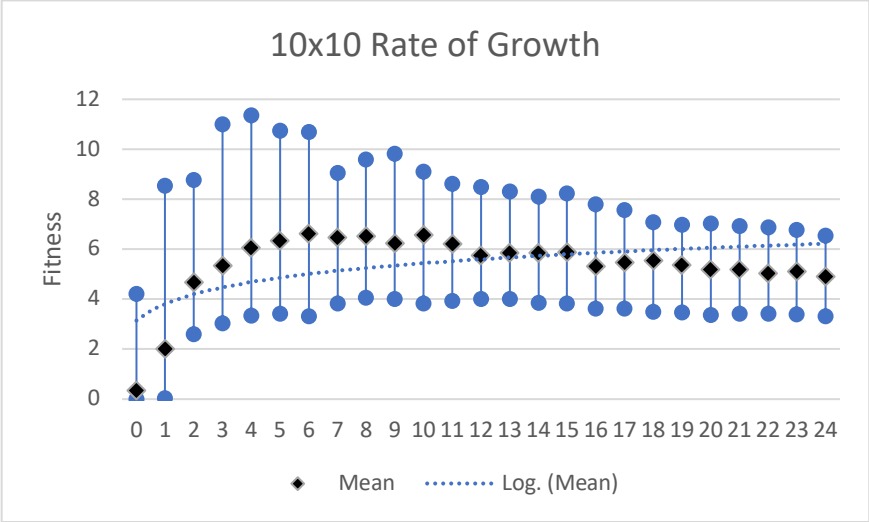


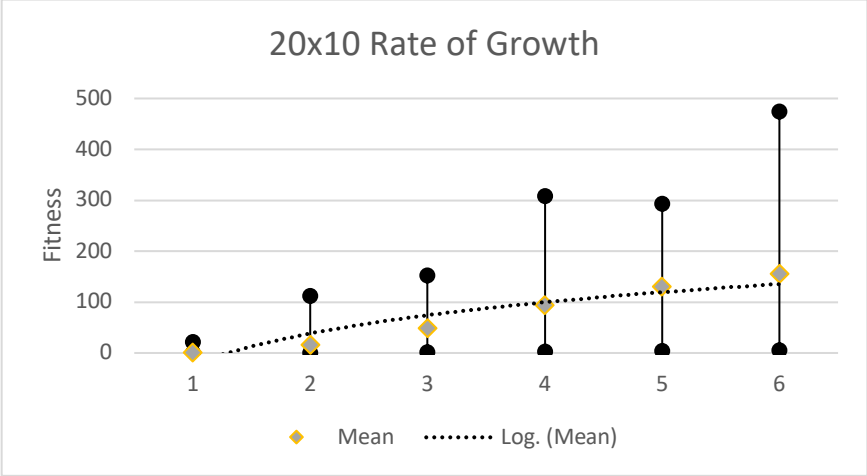
Dellacherie's algorithm yielded a mean score of 844.3.

```
Final Results:
Games run: 10
Mean/avg score: 844.3
Standard Dev: 961.4917628352308
```

I believe this discrepancy is mainly influenced by the fact that my tie-breaking function does not always seem to produce the most optimal action. As I mentioned, the tie-breaker prioritizes piece placement that is further away from the center and unrotated. However, I have noticed that this can lead to disastrous scenarios. Since gym-simplifiedtetris is not 7-bag, the next piece is completely random. Even random sequences can be deadly as there are lethal/non-usable piece sequences that exist. In addition, sequences of Z and S pieces typically place on top of each other, resulting in tall towers that are hard to build around.

For genetic learning results, I attempted to train Dellacherie's algorithm on two board sizes: a 10x10 square board and a traditional 20x10 board.





Ignoring the rate of growth, learning on a 20x10 game board led to much higher scores in general versus a 10x10 board. I believe this is due to the random nature of upcoming pieces. On a smaller board such as a 10x10, a shorter sequence of awkward pieces could instantly end a run. The original intuition was that a 10x10 board was smaller, so would be faster to train as each run would be shorter overall. However, there is far more leeway on a 20x10 board than a 10x10 board which ultimately gimped any form of training on such a small field. This is easy to observe when spectating the agent playing on a 20x10 board. At random moments, it may decide to build very high in order to place pieces more optimally which would be disastrous on a 10x10 board.

Regarding learning time, the 20x10 graph demonstrates how the varability in potential fitness can lead to absurdly long training times. The original plan was to train each game board on 25 epochs. However, the sixth epoch on 20x10 alone took almost three hours while also slowing my computer to a halt. I believe that this would be more realistic if the feature calculations were optimized (as that would speed up playing the game) and I had access to stronger hardware.

Analysis of Agent Gameplay

One key issue I noticed with each of the learning algorithms I implemented was that they did not prioritize downstacking. In Tetris, downstacking is the concept of keeping your field as low as possible, clearing single, double, and triple lines to ensure board safety over scoring points. Often times, this would open access to wells and holes in a Tetris stack, or prevent an opponent from toppling you over in competition.

The features in Dellacherie's and BCTS algorithms do not seem to work around this. Specifically, I noticed that when there was a chance to downstack to access a well, the agent would rather build at the height it was already at and clear single lines to get points. While effectively yielding the same reward, this is a much more dangerous approach. Thus, in the future I will look into seeing if I can increase the value of specific feature weights to prioritize this.

Another issue I noticed was the agent's dependency on single-line clears. Anyone who plays Tetris knows that in order to score larger amounts of points, it is imperative to clear four lines at a time (known as a Tetris). Single-line clears are far safer however as they are the backbone of downstacking. However, that would mean the agent is taking zero risk. I believe this can be solved through an additional feature and would be a great addition for a competitive Tetris agent.

Personal Motivations

Personally, I chose to undertake a Tetris-based project because Tetris is one of my favorite games. I love the fact that you can easily hop into a game, play for a couple rounds (whether it be competitive, sprint, or marathon), and leave whenever you want. Nowadays, most video games do not let you pause and are very time-consuming (referencing MOBAs like League of Legends and tactical shooters like Valorant). Tetris is an outlier compared to all of these and it provides a great way for me to relax and unwind.

Tackling this problem has helped me get a better appreciation of Tetris as a whole. Understanding how the game developed over the years and the various techniques people have formulated to "solve" Tetris have been interesting to read about, especially given how large-scale some of them may be.

Future Goals

For the future of this project, I would love to continue working on this over winter break and into next semester. As I already mentioned, Tetris is one of my favorite games so it is something I would be extremely interested in continuing. Aside from the current algorithms I have planned for this final project submission, I want to continue trying to implement other various learning algorithms such as expectimax and some form of deep learning, and then compare these results in a table. It would be extremely satisfying to see that I have the skills it takes to develop such applications. In addition, I believe it would be a great resume booster, especially if I wanted to go into a career in AI/ML since it would show that I have a good understanding of these topics.

Now that the project is over (for the sake of course submission), there are a few things off the top of my head that I would like to go back over and do. Firstly, I will spend some time fixing the tie-breaker system. I believe this is a crux in the application right now as it improperly builds S and Z pieces into taller towers. Afterwards, I will attempt to implement the 4-feature algorithm mentioned above and see if it is usable and faster overall.

Conclusion

Ultimately, the result of this project was a program that could play Tetris by itself. While not indefinitely, it showcases various Tetris state-based feature algorithms that have been theorized. This project provided me a great opportunity to learn more about genetic learning and its implementation which was something I had always thought was fascinating (from those Flappy Bird and 2048 agents online), but not something that I could realistically implement myself.