T E T R I M A X

Weight-Based Genetic Algorithms for Tetris AI

Mara Hubelbank, Connor Nelson, Tyler Passerine
Northeastern University, May 2022

Overview

The goal of this project is to develop and evaluate a suite of agents which can play the game Tetris asynchronously in a sandbox environment; this relaxation of the game will allow our agents to play optimally without having to make individual keyboard strokes under increasingly strict time constraints. Initially, we aimed to develop an expectimax-based agent using the game-solving strategies covered in the course; however, we pivoted midway to utilizing purely genetic algorithms, as we found the NP-complete problem to be largely unsolvable with deterministic strategies due to the state space explosion. In our final implementation, we present three types of genetic weight-based agents, each of which can successfully play Tetris to the completion of various goals under the aforementioned problem simplifications.

Methodology

Problem Representation

We started with defining the game as a search problem and configuring its data representation.

- The state space representation is a 2-dimensional boolean grid for the active game board, the game score, and the current piece in play.
- A losing terminal game state is reached when the board “tops out”, i.e. a tetromino cannot be placed at least partially within the play zone. Tetris has no true terminal win state, but we will define a score of 999,999 as an agent win.
- Where evaluation functions are applied, they maintain an inverse relationship with the quality of the game state; a terminal loss state is evaluated to negative infinity.
- Grid dimensions are parameterized to allow for experimenting with gameplay on a small board, and also to allow us to evaluate how board size affects agent performance.
- Tetris pieces are represented as enumerated types which, as a behavior, can return a list of their possible rotations, depicted below; these are likewise represented as Boolean grids. In addition to rotation, we define the possible actions on given Tetris pieces of moving left or right, and dropping down on a given game board.

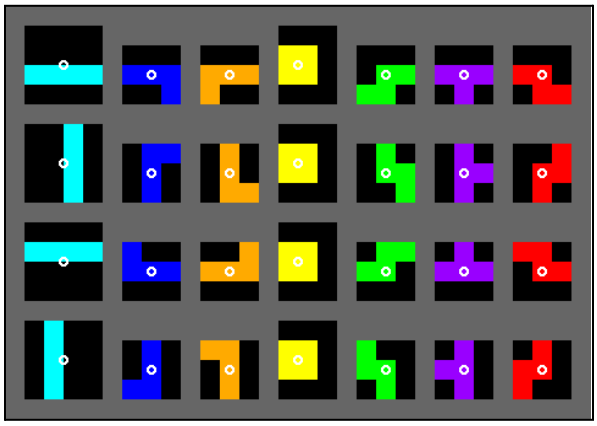

Fig. 1. *Tetris piece rotations*

After defining the game representation, we began to think through designing an algorithm for discovering possible piece placements on a given board. This ended up being far more complicated than we originally thought; simply put, there are far more edge cases than are programmatically calculated and our perception of what is a possible move greatly increased based on our experimentation with various Tetris agents, as shown in the drawing below. In the end, we decided to use breadth-first search as the base algorithm for discovering reachable states; we then adjusted our state space representation based on this discovery.

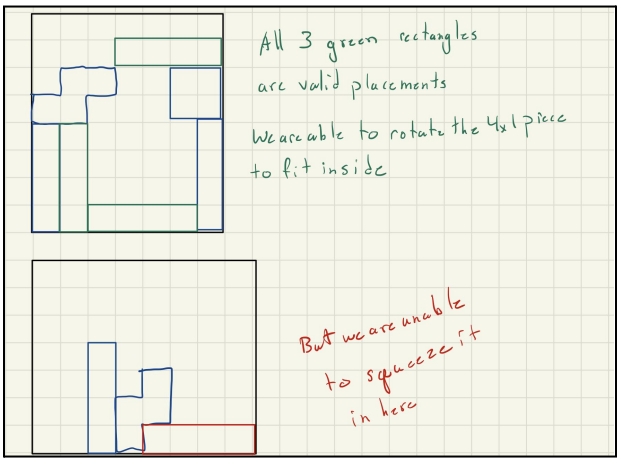

Fig. 2. *Sample edge cases of I-block placement*

Heuristic Design

Next, we designed the goals with which agents would evaluate game states. We began by researching the gameplay strategies used by Tetris world champions, looking at previous implementations of Tetris AI, and playing the game ourselves.

Playing “flat” appears to be a core strategy used by professional players; this is defined as dropping pieces in a way that will create the flattest top row of pieces. This is crucial because a flat playing field allows for more options when placing pieces.

- One rule of thumb for the flat playing strategy is to avoid stacking pieces more than two blocks high, or create a hole more than two blocks deep, because it's impossible to fill a width-one hole of depth greater than 2 with anything other than an I-block.
- Though the above rule is optimal, it is expected that mounds with height greater than two will happen at some point in gameplay, almost inevitably. In such cases, it is generally preferable to build these mounds in the center of the playing field, as this allows for more opportunities to flatten the field.

In addition to vertical spacing, it is effective to optimize horizontal spacing based on the flat strategy as well. In the case of an extremely suboptimal playing board, it is likely that the agent failed to create sufficient O-, S-, and Z-shaped gaps, as these are the least versatile pieces. Therefore, it is important to accommodate for these pieces by creating multiple gaps at least two spaces across.

Implementation

Initially, we aimed to develop a Tetris AI derived from the expectimax algorithm, from which the title of the project is derived. Our preliminary implementations of this design failed miserably due to the innately massive state space: with eight different columns for placement, seven unique piece shapes, and each piece having four rotations, we promptly exhausted our computing resources with models which traversed the Tetris game tree. We experimented with limiting the depth, but this resulted in our agents relying too heavily on their evaluation functions.

Genetic Factory

Pivoting to the genetic algorithm approach, we explored a variety of ways to implement Tetris agent generation; these include manmade agents, a homemade genetic algorithm, and utilizing the NEAT-Python module. Each agent behaves by generating all possible moves and resulting boards for the current piece, weighing each board according to an evaluation function, and taking the move corresponding to the highest weight. This interface behavior design allows us to easily generate many agents which operate in the same way, but have different evaluation tactics. Moreover, most of the agents take in a feature generation function as part of their initialization; thus agents are defined by both their behavior and how they featurize a given board.

The genetic algorithm factory process operates very similar to natural selection. In each generation, we produce agents at random, evaluate each one by assigning it a fitness score, then select the best ones to reproduce. In this context, we define the agents strictly as a list of weights. Weights are started off as randomly generated from a uniform distribution; in order to breed two agents, we choose each weight randomly from the two parents, and add in random gaussian weight.

Agents

There are three main agents that showed promise during performance testing; the first two such agents were tuned with a manual genetic algorithm, and the third trained using NEAT-Python:

1. We initially developed a linear agent, containing as many weights as features. The output of a given board state is given by $x_1 w_1 + x_2 w_2 + \ldots + x_n w_n$.
2. We then experimented constructing an agent with a fixed-size neural network. This network was constructed with two hidden layers both with a number of perceptrons equal to the number of features.
3. Finally, we trained an agent with the NEAT (NeuroEvolution of Augmenting Topologies) using the NEAT-Python module. NEAT is a framework for developing neural networks that can grow over time. Agents start as the simplest possible network, and can evolve by creating new layers and connections the same way that the other agents mutate their weights.

Experimentation

The agents' performance was tested using our evaluation engine module, which takes in an agent instance and simulates an entire game of Tetris. Using tunable parameters, it operates by allowing agents to place the current piece in any valid position. Unlike a human player, the agent is not moving the piece itself: as all of our agents act instantly and asynchronously, it is unnecessary to force the agents to learn to perform complex, time-sensitive input techniques. As such, the only agential task is determining where to put a piece on the given board.

Each agent was evaluated by two metrics: survival time and Tetris rate. To maximize score, Tetris players aim to clear four lines at once, as shown in the standard Tetris scoring algorithm depicted below; this uses a 1-3-5-8 weight, and in experimentation we tested using this breakdown as well as 1-2-3-4, 1-2-3-16, and 1-2-4-8. Moreover, note that we relaxed the problem to ignore level, and have adjusted the scoring mechanism accordingly.

# of Lines Cleared	**Points Awarded**
1 line	100 points x level
2 lines	300 points x level
3 lines	500 points x level
4 lines (a "Tetris")	800 points x level

Fig. 3. *Standard Tetris Scoring.*

Results & Analysis

Agents were trained under two main goals: to survive forever and to score the most points in a fixed-length game. We found that each agent's goal vastly impacted the way in which the agent behaved. For example, agents whose goal was simply to survive as long as possible took no risk and cleared lines as fast as possible; their game boards rarely grew higher than the 3rd or 4th row. From our testing, these agents would never lose. They would easily clear thousands of lines and never come close to the top of the board. On the other hand, agents who were given a goal of scoring the most points often intentionally did not clear rows with the intent to score a "Tetris" (four rows cleared at once) at some point in the future. On average, our agents scored at roughly a 30% Tetris rate, which is lower than the percentage that top human players are able to achieve.

There are two reasons why we believe that our agents are unable to play perfectly (or at least to the level of professional Tetris players). Firstly, the agents do not have access to perfect information; at the moment, they are learning solely based on a feature representation of the board. This means that they don't even get to see the whole board state at once. Our agents are learning based on what we think they should find important, but we anticipate that allowing them to learn and act independently would produce interesting results as well. Secondly, the agents do not have access to information about previously played pieces, lacking any concept of memory. In contrast, proficient human players keep track of the piece count aggressively so that they know how risky they should be playing in the moment based on what they can expect in the future. Notably, this means that unlike humans, an agent cannot know or predict the maximum time until a Hero (4x1) piece, and thus cannot optimally prepare for scoring Tetrises.

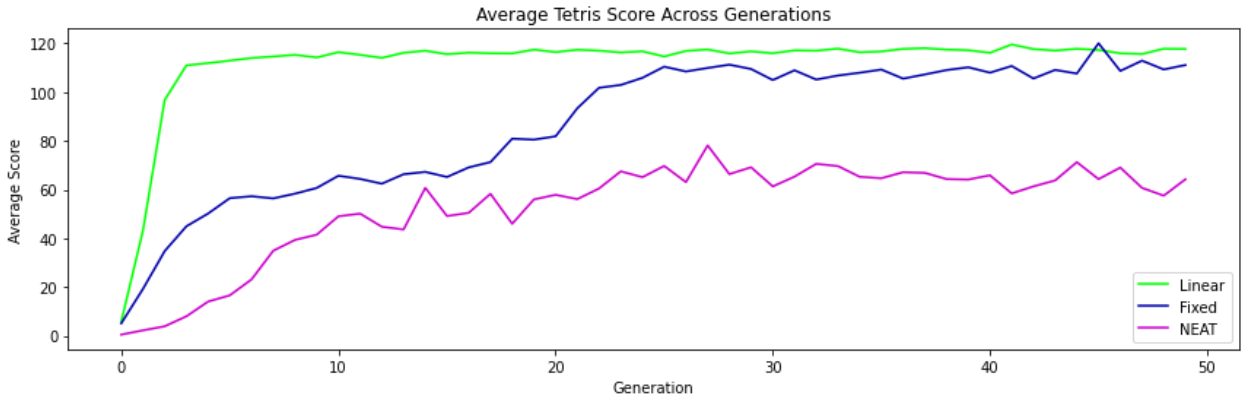

Fig. 4. *Average Tetris score per generation, by agent type*

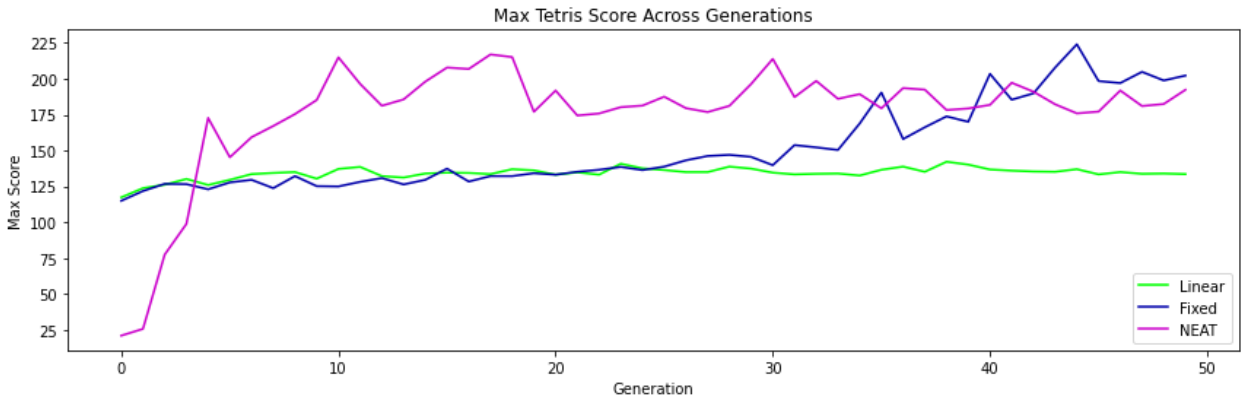

Fig. 5. *Maximum Tetris score per generation, by agent type*

The graphs above visualize the fitness of the best agent from each factory on a generation-by-generation basis. As shown, all of our approaches were effective in generating agents to play Tetris very well, though with notable performance variations. We can see that the more complex agents are able to score a much higher maximum due to the fact that they are able to learn complex behaviors that allow them to score Tetrises. Moreover, we found these agents were able to achieve a Tetris percentage that was relatively high, while the linear agent was unable to have a notably different performance when optimized to score Tetrises. We can also see that the linear agents perform better in the earlier generations, as a result of random weight population, which generally tends to produce at least one agent that performs better by pure chance early on.

Discussion

Given more time, we would like to expand on our project by providing more information to the genetic algorithms. As of right now, we do not provide any information about future pieces and the agents do not retain information about what pieces have recently been played. The latter is especially notable because in all implementations of Tetris, the seven unique pieces are pushed to a randomly ordered set, popped out one-by-one, and played by the player until the set is empty, at which point the set is regenerated. As a result, keen players have an intuition as to how long they will have to wait for a particular piece, such as the Tetris-scoring I-block. Providing these concepts of memory and foresight to the AI would allow our Tetris agents to base their decisions on information which is normally available to and used by the players.

We would also like to expand the model inputs to allow for more features. This extension would give the agents access to a more comprehensive representation of the game board, and thus enable them to make better-informed decisions. Lastly, we relaxed the Tetris state space for this project by only permitting drop moves; this means that particular edge case states, such as that depicted in the top half of Fig. 2, are unreachable in this implementation. We would be interested to see if it is possible to represent more complex piece placements used by professional players, such as the T-spin, and to evaluate the effects of such moves on agent performance.