Reinforcement Learning for Tetris

Michael Kayser and Mark Paluta

Abstract

This report details the implementation of a Sarsa reinforcement learning algorithm for the game Tetris. Tetris has an extremely large state space, which we handle by featurizing into elements strategic to gameplay decision making. The agent enumerates all possible sequences of actions on an active piece and determines the best action sequence according to its current policy. We evaluate performance by computing a trailing average of recent rewards and compare to baseline methods of random actions and lowest center of gravity block placement. We found that the most successful agent used reinforcement learning that was first "taught" with the lowest center of gravity baseline.

Introduction

Tetris is a game played played on a 20 row, 10 column grid with each grid space a binary value of occupied or unoccupied. An active tetromino "falls" down the rows of the grid, and the player can horizontally translate the piece or rotate it until it cannot fall any further without taking up an already occupied space or moving beyond the bottom row of the grid. At this point, a new piece is randomly generated to be the active piece and the process continues. Completing an entire row of the gameboard with a piece placement will positively reward the player, return all the cells in that row to an unoccupied state, and cause any blocks above to translate down one row. This is known as "clearing" a row. Multiple rows can be cleared with a single tetromino placement. Additionally, a negative penalty is assigned if the blocks stack up to hit the ceiling of the game board (the active piece would be placed partially or fully above row one). In order to facilitate reinforcement learning, a variable game board environment was created to allow non-standard grid sizes, pieces smaller or larger than tetrominoes, and variable scoring. The game board can be visualized to observe the agent's decisions online.

Problem formulation

The problem of playing Tetris can be formulated in at least two ways, depending on the level of granularity chosen. At a fine level of granularity, the agent chooses among five actions, Left, Right, Rotate, Down, and NoAction. We chose a simpler formulation in which the agent chooses among all possible legal *placements* of the current active shape (thus, the list of candidate actions is dynamically generated based on the current world state and active shape). Once this selection is made, the placement occurs and the next block is generated. This simplifies our scenario because state transitions are simply the transitions among block configurations. As a result, choosing the best action reduces to choosing the best block configuration. This makes Tetris a game with "turn-based" mechanics, somewhat similar to chess.

Algorithms

Baselines

We wrote the code for our project in Python; it is available on github (**https://github.com/mkayser/tetris_rl**). After implementing basic game mechanics, we wrote a "random" agent which performs actions randomly. This agent was very ineffective as was expected. As a more competitive baseline, we implemented a lowest center of gravity (LCOG) agent. LCOG looks at all possible orientations and placements of the active piece and chooses the one that would give it the lowest resulting center of gravity. This was a strong baseline, as will be discussed in the results section.

Reinforcement learning

We chose *Sarsa-λ learning with global approximation* for the reinforcement learning (RL) algorithm. Sarsa-λ is a model-free method of RL using the Bellman equation to update expected utilities given state-action pairings. In small problems, this works in a tabular format with each pairing representing a cell of the table, but for larger problems such as this one, generalization is needed to assess unseen states. We used eligibility traces in an attempt to facilitate convergence given the relatively rare rewards and penalties. For complete information on the hyperparameters used, please see Table 2 at the end of the paper.

Linear approximation

Linear approximation is a standard method for approximating a function—in our case, the Q-function. We predicted that linearization without basis expansion might prove a poor method for approximating the Q-function. For example, one of our features, *maximum column height*, can be good to a certain extent because the existence of blocks allows the player to clear rows and score points. However, too high of a maximum column height means the player is nearing the ceiling and may soon receive a large negative reward. With only a linear weighting of the feature, the algorithm was predicted to struggle to assign a single weight to meet these contradictory goals.

Discretization and basis expansion

One method to handle the above problem is to *discretize* the features. Since the overarching state space of Tetris is discrete by nature, discretization of certain features can be straightforward. An example would be maximum column height. The height of any column can take on integer values from 0 to 20 in the standard tetris environment, so using these values as bins can capture the nonlinearities. The disadvantage to this method is that the algorithm does not have inherent knowledge that nearby bins should elicit similar behavior from the agent (i.e. that bin 15 is similar in nature to bin 16), so it has approximately four times as many independent parameters to learn as, for example, a fifth degree polynomial, requiring only five independent parameters. As a result, while we experimented with discretization, we focused

attention on *polynomial fitting*, particularly with features normalized to values between 0 and 1. Without normalizing feature values, polynomial fitting was prone to divergence, perhaps because the dynamic range of different features could be vastly different.

Teaching agent

Finally, we experimented with using a *teaching agent* to aid in the initial stages of training. In this scheme, the Sarsa learner spends a fixed number of timesteps (say, 10-30 thousand game clock ticks) updating weights according to observations, but not choosing actions: the actions are instead chosen by a *teacher*. After the fixed timestep limit, the Sarsa learner begins choosing actions in the usual way. We chose the LCOG baseline as the teaching algorithm.

Exploration is also built into the algorithm to ensure that many states are visited in the learning process. We use epsilon-greedy action selection: every so often the agent is forced to place a block randomly.

Featurization

Because the standard game board has 200 grid spaces of binary possibility, with the constraint that no row can be completely filled, $\left(2^{10}-1\right)^{20}$ possible states exist for the game board [Bodoia]. To reduce the state size, we featurize the Q-function. We included the following features in the final configuration for experimentation purposes because they seemed to include most aspects of good Tetris strategy:

- **Mean height:** The average height of all blocks.
- **Maximum Height:** The height of the highest block.
- **Square Types:** A vector of counts, one for each square type. Valid square types are: filled, unfilled, and *unfilled-but-trapped-by-K-blocks* for K from 1 to 14 (thus, there are 16 total features). A square is trapped by K blocks if there exist exactly K filled squares somewhere above it. These counts are normalized so the vector sums to 1.
- **Trapped Squares:** The percentage of trapped squares. This is simply the sum of the above-defined *unfilled-but-trapped-by-K* percentages for all values of K. Intuitively, trapped squares are to be avoided.
- **Row State Histogram:** A vector of frequencies for each *row state type*. A row state is a triple of integers containing the number of filled, unfilled, and *unfilled but trapped* squares in a row. Intuitively, row states depicting empty or nearly complete rows are good, while row states depicting rows with trapped squares are bad.

Other features investigated included:

- **Height Variance-** The variance of the highest block each column.

- **Row Transitions-** The number of occupied/unoccupied transitions iterating across rows. Left of the first column and right of the final column are considered occupied.
- **Column Transitions-** The number of occupied/unoccupied transitions iterating down columns. Above the ceiling is considered unoccupied and below the floor is considered occupied.
- **Compactness-** The total number of placed blocks / (mean stack height * width of game board).
- **Maximum Well Depth-** The depth of the deepest well. A well is an uncovered vertical group of unoccupied cells surrounded on either side by occupied cells.
- **Covers-** The total number of covers. A cover is defined as an occupied cell with at least one unoccupied cell below it.

Experiments

We analyzed performance in two ways. First, to diagnose convergence we tracked δ, the difference between predicted and observed Q values. Due to the stochasticity of Tetris's random piece generation, the value will never converge exactly to zero. We expected (and observed) it to oscillate above and below zero with equal frequency, indicating equiprobable over and under-estimations of the Q value.

Our second and more important metric was the trailing average of the past 30000 rewards.

For experimentation purposes, we chose a slightly smaller, 15 by 10 arena, which significantly improved runtime. Additionally, similar to some implementations of Tetris, we chose the rewards for clearing multiple rows at once to nonlinearly increase with number cleared. Specifically, the agent receives 1, 8, 27, and 64 points for simultaneously clearing one, two, three, or four rows respectively (The penalty for filling the grid and thus restarting the game is set to -100). This made the required strategy inherently more interesting as an agent would have the option to plan ahead to build up a large well before clearing several rows with a single piece. It should be noted that this reward schedule limits the effectiveness of the LCOG baseline because that agent rarely builds up large wells.

With this environment established, we ran tests using the following agents:

- Random baseline
- LCOG baseline
- Five-feature "standard configuration" with varying degrees of polynomial used to weight features (linear up to quintic).
- Four, three, two, and one-feature configurations, created by removing one additional feature at a time from the standard configuration. Quintic weighting of features.
- Teacher algorithm. LCOG for either 10K or 30K game clock ticks, followed by standard five-feature configuration with quintic fitting.

feat5 (main)	feat4	feat3	feat2	feat1
MeanHeight MaxHeight SquareTypes TrappedSquares RowState	MeanHeight MaxHeight SquareTypes TrappedSquares	MeanHeight MaxHeight SquareTypes	MeanHeight MaxHeight	MeanHeight

Table 1. Features used in different feature ablation configurations. See Figure 2.

Results

First, we compared the two baseline agents to the standard configuration as well as two teaching algorithms, one transitioning at time step 10,000 and one at time step 30,000. Figure 1 shows the trailing average reward over time of these agents. Note that the LCOG baseline outperforms the RL algorithm *unless* the RL algorithm is "taught" first. It is also important to note that the untrained RL algorithm does appear to be trending upward, so in time, it might catch up to LCOG as well.

Figure 2 shows the difference in performance with varying numbers of features. The best combination of features attempted was Maximum Height, Mean Height, and Square Types. Adding further features beyond this actually hurt the performance. This could be because the added features do not add much information but create a more challenging (high-variance) learning problem, hindering convergence.

Finally, we wanted to see whether higher order polynomial basis expansion improved significantly upon linear weighting. In fact, linear weighting performed the best, significantly above the other four higher-order methods. This difference is shown in Figure 3.

Conclusions

The most effective agent was that which used reinforcement learning, when "taught" with the LCOG algorithm. Without first teaching, RL performed worse than LCOG at up to 75,000 time steps. Additionally, polynomial basis expansion actually worsened the RL algorithm's performance. Finally, including several features enabled the algorithm to have proper strategy, but too many features worsened the algorithm's performance.

Sources

Bodoia and Puranik. Applying Reinforcement Learning to Competitive Tetris.
http://cs229.stanford.edu/proj2012/BodoiaPuranik-ApplyingReinforcementLearningToCompetitiveTetris.pdf
Bohm, Kokai, and Mandl. An Evolutionary Approach to Tetris.
https://www2.cs.fau.de/EN/publication/download/mic.pdf

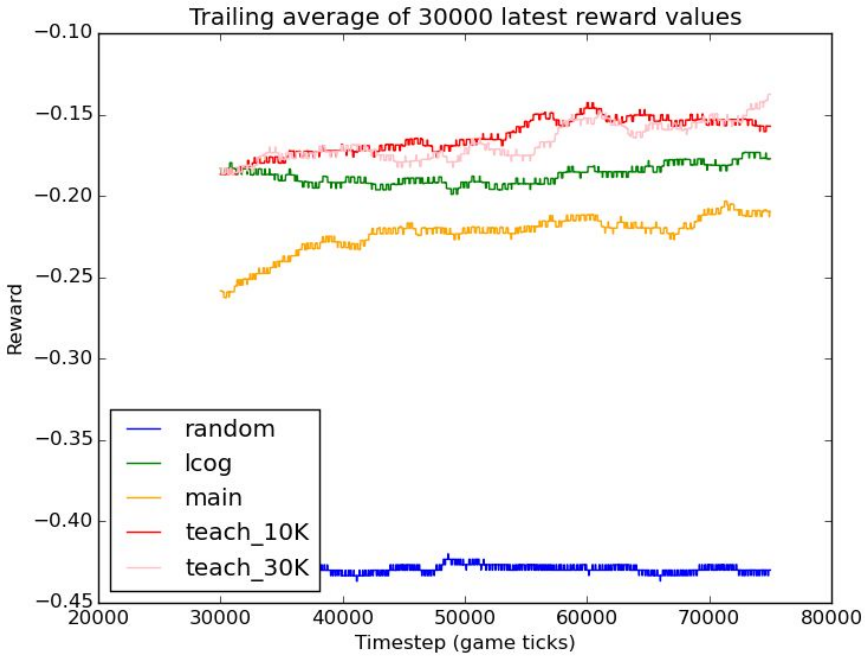

Figure 1. Performance of different system types. Observe that the "taught" RL systems are better than the "lcog" (Lowest Center of Gravity) baseline, that the regular "untaught" RL system (main) is worse than the baseline, and that the randomly-acting system is far worse than others.

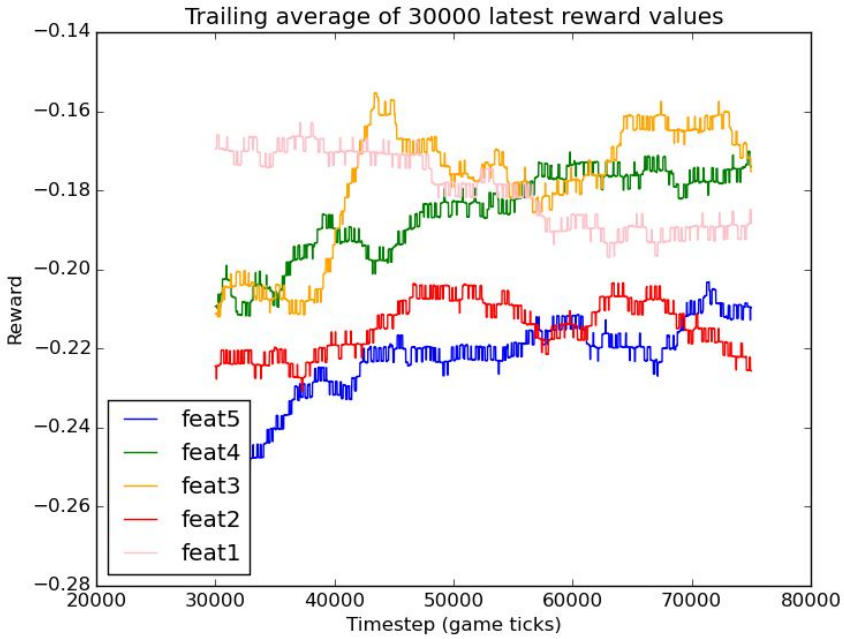

Figure 2. Performance of different feature sets. Observe that using 3 features (specifically, MaxHeight, MeanHeight, and SquareTypes) works best. See Table 1 for system definitions.

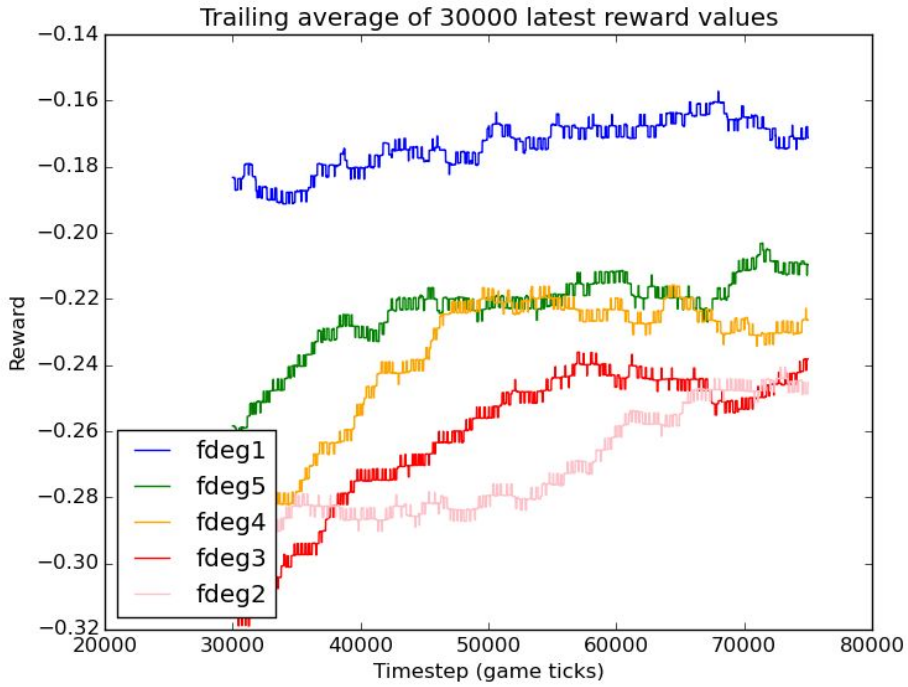

Figure 3. Performance of different degrees of basis expansion using all 5 feature types. For example, "fdeg3" means that for each feature type, the first, second, and third powers are included as features. Observe that using linear features (no basis expansion) works better than all other configurations.

Parameter	**Meaning**	**Value (all experiments)**
λ	Eligibility trace discount	0.9
H	Length of history (number of immediately preceding states to which we apply eligibility trace updates)	40
ɣ	Discount factor	0.9
α	Learning rate (starting value)	0.1
κ	Learning rate annealing factor (multiplicatively applied to learning rate at each game clock tick)	0.9994

Table 2. Hyperparameters used for all configurations of reinforcement learning experiments.