Term Project: Learning to Play Tetris with Big Data

by Project Group 06
Fan Weiguang(A0148031R), Wu Zefeng(A0147971U),
Xie Jihui (A0147969E), Yang Zhuohan(A0147995H)

Abstract

The aim of this project is to design an artificial intelligence agent to learn to play Tetris. Our team tried both Particle Swarm Optimization (PSO) and Genetic Algorithm (GA) and used the better one for training the agent. In this report, the idea of algorithms design will be discussed, which is followed by training details. Next, the training results will be illustrated and compared. Based on the results, the report will have a brief discussion on the reflections and how it can be scaled to big data learning.

Design of AI agent

The main idea for an AI to play Tetris is to try all possible placement of a Tetromino (game piece) and evaluate how likely to play further based on the resulting game board. Then, agent pick the placement that has the highest likelihood for next move. Our agent uses a weighted linear sum of different features to represent the likelihood. More formally, use H = $\sum_i^n \omega(i) * \Phi(i)$ as a heuristic function to describe the game board. where n is the number of features of a game board, $\omega(i)$ is the weight of the i-th feature and $\Phi(i)$is the corresponding value. After choosing the features that best describes a game board, we used both PSO and GA to find optimal weights.

Algorithm Details

PSO Algorithm

For PSO, the basic idea is that we randomly initialize a group of particles, each has its initial velocity and position. Then we let each particle to play the game of Tetris to get a fitness value. After that, particles are to update velocities and positions according to the self-best fitness and social-best fitness until all particles reaches the best positions. Specifically:

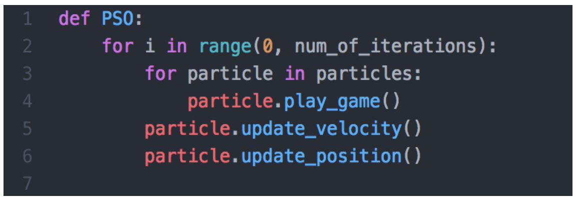
$$x_i(t+1) = x_i(t) + v_i(t+1)$$

$$v_i(t+1) = inertia \cdot v_i(t) + c_1 r_1(t)(y_i(t) - x_i(t)) + c_2 r_2(t)(\hat{y}_i(t) - x_i(t))$$

Where here $x_i(t)$ is the position of particle i at time t. $v_i(t)$ is defined as the velocity for particle i at time t. $inertia$ is a coefficient that describes how important is the intial velocity. c_1 and c_2 are coefficients describing influences from individual and social best position. $r_1(t)$ and $r_2(t)$ are two random numbers between 0 and 1. Then $y_i(t)$is the position of particle i when it meets the best fitness. $\hat{y}_i(t)$is the position of the best neighbouring particle of particle i at time t. At last, $x_i(t)$is the current position of particle i at time t. To evaluate the fitness of a particle, it will play certain number of games and observe the accomplishment. We made use of the evaluation function proposed by Langenhoven (Langenhoven, 2010):

$$fit(p) = L + (\frac{P_{max} - P}{P_{max}} \cdot 500) + (\frac{H_{max} - H}{H_{max}} \cdot 500) + (\frac{R_{max} - R}{R_{max}} \cdot 500) + (\frac{C_{max} - C}{C_{max}} \cdot 500)$$

Suppose k games are played, L is the total number of lines cleared. P_{max} corresponds to the max height in the terminating game-board while P denotes the average over k games. H, R and C are just similar while each of them corresponds to number of holes, row transitions and column transitions (definition can be found in training details). The pseudo code is:



```
def PSO:
    for i in range(0, num_of_iterations):
        for particle in particles:
            particle.play_game()
        particle.update_velocity()
        particle.update_position()
```

GA Algorithm

Genetic algorithm mimics the process of natural selection, it evolves the weights to maximize fitness value, which is basically the total number of lines cleared in 100 game plays with upper limit of 200,000 moves each. A population is initialised with some random weights, from which we assign each set of weights with a fitness value evaluated by the fitness function. After one iteration over each set of weights in the population, a portion (30%) of the population with high fitness values will be selected as parents, from which we generate crossovers by formula $\vec{c} = \vec{p_1} * fitness(\vec{p_1}) + \vec{p_2} * fitness(\vec{p_2})$, where $\vec{c}$ denotes the crossover and $\vec{p_1}$, $\vec{p_2}$ denote a pair of high fitness parents. To maintain and introduce diversity, as well as prevent the algorithm from getting stuck at local maxima, mutation is applied to each child with a small probability by the formula $\vec{c}[i] = \vec{c}[i] + r$, where $\vec{c}[i]$ denotes a random weight selected, r denotes a random value in the allowed range (±0.2 value in one variable). From now on, a portion of the population with lower fitness value will be pruned and replaced by newly produced children. This process will continue until the number of iterations is reached.

Implementation Details in Training and Novel Contributions

In our design of Tetris agent, we decided to take advantage of the following 13 features as described in Langenhoven's paper (Langenhoven, 2010).

- Rows Cleared: the number of rows that are to be cleared after the placement of a Tetromino.
- Pile Height: the height of the highest column in the game-board.
- Hole Number: the number of holes in the game-board. Where a hole is defined as a non-filled cell that has a filled cell above.
- Connected Hole Number: the number of connected holes. A connected hole is formed by merging adjacent holes in the same column.
- Altitude Difference: the difference between the highest column and the lowest column in the game board.
- Max Well Depth: the largest altitude difference between two adjacent columns.
- Well Depth Sum: the sum of all adjacent altitude difference between two adjacent columns.
- Landing Height: the lowest row number on which the incoming Tetromino lands.
- Block Count: the total number of cells occupied in the game-board.
- Weighted Block Count: the sum of occupied cells, which are weighted by their corresponding row numbers.
- Row Transitions: total number of horizontal transitions between a block and a hole. The border is treated as a block.

- Column Transitions: total number of vertical transitions between a block and a hole. The border is treated as a block.
- Eroded Piece Count: Sum of products of number of blocks removed from the current piece and number of rows removed after this move.

particle number	iteration number	inertia	c_1	c_2	v_{max}	Games per particle
25	30*2/50*2	0.72	1.42	1.42	0.5	10

PSO algorithm settings

The neighbors in our PSO is defined using Von Neumann topology. Specifically, fill 25 particles into a 5 by 5 grid. Neighbors of particle are 4 adjacent particles in the grid layout. To prevent converging too fast, we limit the max velocity to 0.5.

population size	No. of games in one iteration	portion for parents' selection	children replacement	mutation probability	the max no. of mutation
1000	100	0.1	0.3	0.05	0.2

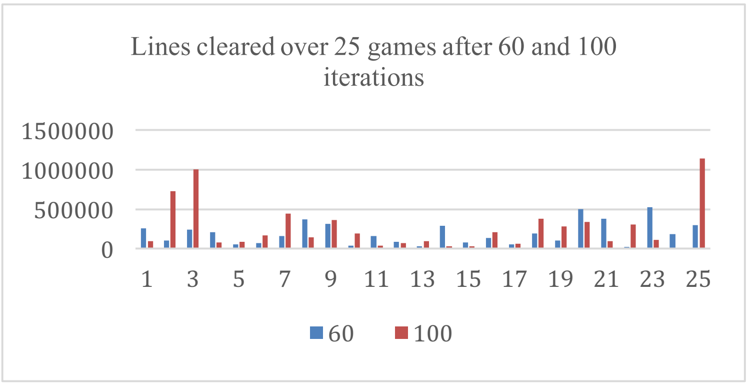
GA algorithm settings

Instead of merely following the algorithms described in paper and other web blogs, we made the following changes.

1. For PSO, when a particle is at local maxima, we add some random factor into is velocity to prevent pre-maturing. Specifically, random factor $r(t+1) = \frac{r(t)}{2}$, $v(t+1) = inertia * v(t) + r(t+1)$
2. As each game of Tetris is independent of each other, we used multi-threading to speed up our process. Particularly, PSO is using 5 threads, each thread runs 5 particles. As for GA, 25 threads are used and each thread runs 40 of population.
3. Instead of running the game to the end, we stop the game at some points and evaluates the stop state.
4. We made use of expectimax strategy when playing the game, considering the influence from one step deeper.

Results and Analysis

1. Without expectimax strategy, the final results from PSO and GA are similar: average of 260 thousand for PSO and 240 thousand for GA. Since the final results for both algorithms do not have much difference, we will take the best result from PSO and show specific performance without expectimax.
   Weights:
   0.6816800738276745, 0.5588151488560493, -3.1031901911592934,
   -4.0657302433854685, -0.9294522666278708, -0.9983263248003504,
   -0.4915038312098661, -3.2275968857355286, 0.008856427717670508,
   -0.03773914898487388, -2.596685029585629, -3.7290379638324547,
   1.645615379270798



Iterations	Average
60	194,458.5
100	259,284.5

Iterations	Best
60	526,993
100	1,139,285

2. PSO converges much faster than GA. After 60 iterations, the best particle in PSO can clear 190 thousand rows. For GA, the best in the population can only achieve 50 thousand rows. We think the main reason is that GA is dealing with larger population and hence converges much slower.
3. Without random factors, PSO and GA converges much faster. However, the best result is not good enough. Weights are likely to be stuck in local maxima. Specifically, the best weights can only clear around 11 thousand rows. Adding random factor helps with preventing pre-maturing.
4. Multi-threading helps significantly in terms of training efficiency. However, the improvements are not linearly distributed. For instance, 25 threads only speed up the process by 4 times.
5. Expectimax helps with playing performance because it considers the influence from one step deeper. After experiment, we found that expectimax can improve result by at least 50 times but the playing is rather time consuming. Due to time constrain, we just have five results from expectimax player: 14’508’002, 8’304’947, 27’393’976, 16’294’063, 20’595’300.

Conclusion and How to Deal with Big Data

After experiment, we are confident to say that PSO is more suitable for dealing with Big Data, since it does not depend on large population for training. When implementing PSO, we should make use of multi-threading on game play part. Properly implemented multi-threading can speed up training by 4 to 5 times. Then, players should terminate after certain number of game terms and do evaluation. From our experience, if we allow player to play to the end, it would take 2 to 3 days to train for 100 iterations. When game terms are limited, 9 hours are sufficient. In addition, instead of train weights 100 iterations directly, we could break it down to multiple batches. After each batch, select best weights for next batch. This helps weights converge faster.

Reference

Andries P. Engelbrecht, Leo Langenhoven and Willem S. van Heerden. “Swarm Tetris: Applying Particle Swarm Optimization to Tetris”. IEEE congress on evolutionary Computation (2010): n. page. Sat, 7 Apr. 2018.

Lee, Y. (2017, June 14). Tetris AI – The (Near) Perfect Bot. Retrieved April 16, 2018, from https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/