Deep Reinforcement Learning on Tetris

Chen, Xiyu
Gu, Zhengrong
Jiang, Tianxing

Abstract

This project focuses on building an agent that can play Tetris. The team wants to look into deep reinforcement learning and its application in game Tetris. Deep Q-learning was chosen to be used because of its simplicity and fast speed as Tetris requires fast decision making by players. The team finally builds an agent that can play Tetris very well. The agent can easily keep playing Tetris games over 5,000 pieces. Deep Q-learning is a good algorithm to build a Tetris agent. However, there are still a lot of features and decision processes to look into in the future.

1. Introduction

Tetris is a tile-matching game created by Russian software engineer Alexey Pajitnov in 1984. It has been popular for decades and loved by many. In Tetris, players complete lines by moving tetrominoes, which descend onto the playing field. Once lines are completed with pieces, they disappear and grant player points, and players can continue playing until there are no vacated spaces in the playing field. The longer the play lasts and the more the number of lines disappear, the higher the score is.

Games are no longer just entertainment, though. Training an agent to play like a human being or even outperform human players, and to optimize its performance, can teach us how to optimize different processes in a variety of different subfields. With AlphaGo, Google DeepMind beat the strongest Go player and scored a then-unbelievable score.

The primary motivation for this project was to apply deep learning in Tetris games. Tetris is a game that involves a fair degree of strategy. The player is continually given pieces of varying shapes that must be rotated and positioned, and then dropped. Since these pieces keep piling up, the player must try to stack them efficiently and delete as many lines as possible to keep height as low as possible and gain score in this way. People who play usually spend hours on this addicting game. We hope to build a simple virtual agent to learn the mechanism of this process.

2. Related Work

Playing Atari with Deep Reinforcement Learning [1] presented the first deep learning model, which trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. It outperforms all previous approaches on six of the Atari games and surpasses a human expert on three of the Atari games. This work serves as the milestone. Its follow-up, Human-level Control through Deep Reinforcement Learning [2] bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that can excel at challenging tasks.

There is previous work on Tetris agent building. Back in 2005, [3] investigates the possibility of applying reinforcement learning to Tetris and discusses consideration surrounding implementation. In 2007, [4] presented the results of experiments of playing the game of Tetris with reinforcement learning and neural networks. There is also work on improvements on learning Tetris with cross entropy [5] and building a simplified version of networks to play Tetris based on these papers [6].

3. Methods

3.1 Deep Q-learning

Because actions in the Tetris game will not be too many and Q-learning is very fast in evaluation, Deep Q-learning is chosen to be used for training the Tetris agent, and the function (1) needs to be evaluated in the model.

$$Q(s,a) = r(s,a) + \gamma \cdot max\ Q(s',a) \qquad (1)$$

In the function, s is the current state and s' are next states. There is one action according to each state. The state is defined and discussed in section 4.1. The action is in section 4.2. R is the reward function and is discussed in section 4.3 Reward. γ is the discount rate or the learning rate in Deep Q-learning.

3.2 Agent Design
The network of the agent is simple; the state is not complicated, and the whole playfield of the tetris is not used as the input. There are only three layers in the network. The first layer has 64 neurons and the second layer has 16 neurons and only one output. The activation function of all the layers is relu.

The train is offline. The network is trained every time when the game is lost or reaches the setted maximum steps. 100,000 thousand states are stored in the memory as a queue. The old states will be replaced by new ones when the memory is full.

4. Datasets and Features
There is no package or built version Tetris available. Hence, a Python Tetris emulator is designed and coded in the project. This emulator is designed to extract states and perform actions, and render the game into images. Some existing Tetris game emulators have the next piece's information. In this emulator, the next piece is not told to the agent. The rendered game is shown as fig1.

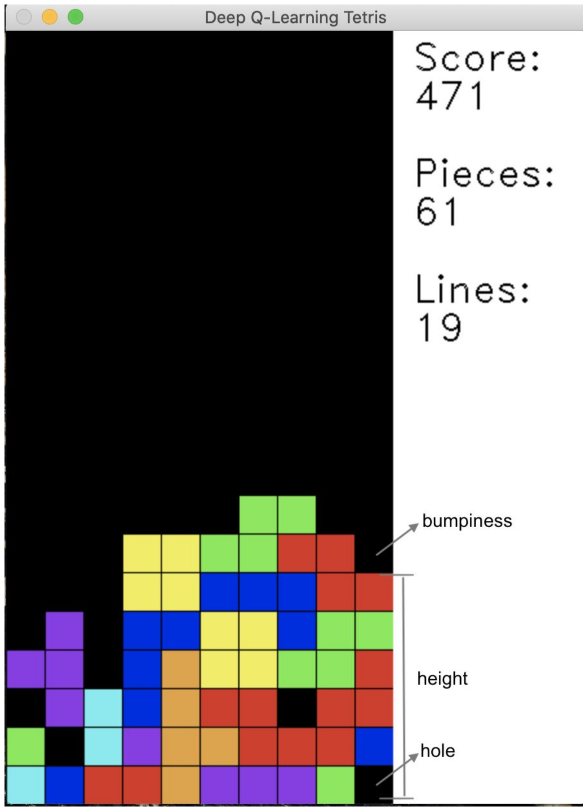

fig1. Tetris Game Image

4.1 States
The state has four properties in our model. They are cleared lines, holes, height and bumpiness.

4.1.1 Cleared lines
Cleared lines are the number of lines being cleared in this state. For example, in the fig1, after the purple piece dropped, the second last line will be cleared.

4.1.2 Holes
Holes are referred to the squares that are empty and have pieces above it. Every state will return the number of holes in that state. You can find what a hole looks like in fig1 as noted. There are two holes in that state.

4.1.3 Height

Height refers to the height of a column on the playfield. For example, in fig1, the most left column's height is 2 now, and after the purple piece drops, it will be still 2 as the second line is cleared. The second most left column's height is 3 after the piece drops. Every state returns the sum of the heights of the whole playfield.

4.1.4 Bumpiness

Bumpiness refers to the height difference between neighboring columns. For example, in fig1, the most left column and second most column has a bumpiness of 1 and after the piece drops, the bumpiness will still be one.

4.2 Actions

As mentioned earlier, the next piece is not provided to the agent. The action is defined as two parts, the piece and the angle to rotate. In the python code, it is coded as a tuple (piece, angle).

4.3 Reward

The reward is defined in function 1 and is listed below when the game is not over and -1 when the game is over. The γ is 1 in our model.

$$
\begin{aligned}
Q(s,a) &= r(s,a) + \gamma \cdot max\ Q(s',a) && \text{Not Done} \\
Q(s,a) &= -1 && \text{Game Over}
\end{aligned}
$$

5. Results

The result is very satisfying. After training the agent 700 times, the agent can play the game very well. The agent is saved every 50 training. The results are showing on fig2.

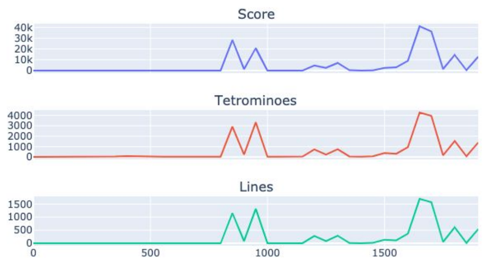

fig2. Agent Results

The results are straight-forward. After a lot of random work, the agent begins to learn to clear the lines and the agent drops when it over-emphasizes on clearing several lines at the same time, because it will have a higher reward. The agent of every 50 training is uploaded to the Github. It can be run with run.py and with some observation, the hole, bumpiness and heights are all influential to agents' decisions. We guess after the playfield has too many holes, the agents will try to eliminate lines to reduce the holes on the playfield.

6. Conclusion and Discussion

From the result, the whole playfield as an image input will not be necessary for a Tetris game, unlike a lot of other games. Tetris is not a very complicated game in this way.

The four states are all significant to our model. The cleared lines will influence the rewards directly and the other three states will influence whether the agent should accumulate more lines to clear one time or clear lines once there is opportunity.

However, training a decent agent with different rewards and penalties to play Tetris costs a lot of time. We assume that the loss after some well trained agent in fig2 is caused by the overemphasis on the number of cleared lines. With limited time, however, we have not confirmed this yet.

For future work, more kinds of rewards, learning rates and penalties can be applied. For instance, penalties on heights can be applied to let the agent focus on reducing the heights. However, it would be hard to find out the influence of those penalties. Some more designs will be needed for the decision making process of the agents.

About networks, though taking the whole playfield as input may not be necessary, it helps find out the patterns of the agent and understand the decisions of the agent. If the whole playfield is taken into consideration like a matrix, then employing CNN models will be a good direction when playing Tetris.

Reference

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602, 2013.*

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature, 518(7540): 529–533, 2015.*

[3] D. Carr. Applying reinforcement learning to Tetris. *Technical report, Rhodes University, 15 pages.*

[4] N. Lundgaard, B. Mckee. Reinforcement Learning and Networks for Tetris. 2007.

[5] C. Thiery, B. Scherrer. Improvements on Learning Tetris with Cross Entropy. *International Computer Games Association Journal, ICGA, 2009, 32.*

[6] M. Stevens, S. Pradhan. Playing Tetris with Deep Reinforcement Learning.