

# A Reinforcement Learning approach to solve Tetris

## Project report

INF581 - Advanced Topics in AI

---

**Abstract.** Here we report on our use of deep reinforcement learning to train an AI to play Tetris. We use a neural network to estimate a  $Q$  function that describes the best action to take at each game state. Unlike many other approaches of this particular problem, we did not input the raw pixels from the game display, but the information present in the RAM of the computer running the game. That means we used the variables of the environment to interact with our agent, such as the position and orientation of the locked shapes and of the falling piece. This approach failed to converge for the original game so we narrowed it down to a reduced environment with fewer and easier shapes and a smaller initial grid. With greater training time, we expect to demonstrate the ability of a neural network to learn to play the original Tetris without any supervised training.

---

## 1 Introduction

### 1.1 A bit about Tetris

Conceived in 1984 by Russian engineer Alekseï Pajitnov, Tetris is a widely known game that has earned its place as one of the biggest classics ever along with Pong, Space invaders and Pac-Man. The idea of the game is simple : you begin with a  $10 \times 20$  empty grid and pieces of particular shapes, called "tetrominos", fall from the roof at a precise tempo. The goal is to manipulate those tetrominos by moving each one sideways and/or rotating by quarter-turns as they go down so that they form complete lines without gaps. When such a line is completed, it disappears and the blocks above fall down to fill the empty space as the score is incremented. If you complete many lines at once, the score increases as a quadratic function of the number of lines cleared. As you clear lines, the shapes begin to fall faster and faster. The game is lost when once piece reaches the top line.



Figure 1: Original tetris game. We can see all seven tetrominos on the right side, the score as well as the next shape that will fall as soon as the currently falling piece reaches the ground.

### 1.2 Why Reinforcement Learning

We choose Tetris for our Reinforcement learning project since it is complex enough for machine learning techniques to be required to find a really performing solution<sup>1</sup>, but the conception of the environment remains relatively easy.

Mathematicians have studied Tetris for quite some time now and it has been shown that there are sequences that alternate between S and Z tetrominos and assure that the game will end in a finite time [4], which proves that the game must almost certainly end. Tetris has also been proven to be NP-complete [5] which makes it computationally impossible to linearly search the entire policy shape and decide the ideal action to be taken. Since there is no memory in this game, ie. the environment satisfies the Markov property, Reinforcement Learning appears as a promising way to approximate an ideal solution.

---

<sup>1</sup>Unlike Pong or Space invaders that could be efficiently solved with deterministic methods.

More specifically, we choose to use a Deep-Q-learning approach to drive our Markov Decision Process. The idea is to approximate the Q function<sup>2</sup> via a neural network that gets updated as the agent learns through trial and error.

### 1.3 State of the art for AI and Tetris

There are two different ways to implement an AI in order to learn Tetris : it can either only have access to the grid configuration and the falling tetromino - then called one-piece implementation, or it can also have access (like a human would) to the next piece - then called a two-pieces implementation. As listed in Colin Fahey’s article Tetris [6], the best real time one-piece algorithm in the world averages around 650000 cleared rows. It is implemented through a static policy using the understanding of the game of Pierre Dellacherie, its programmer. The best two-pieces algorithm, which is also a static policy averages around 7 million cleared lines per game.

As far as Reinforcement learning goes, there is not yet a published promising and efficient solution to Tetris. There have been a few experiments on smaller environments (limited shapes and smaller grid) [7], different reward policy and different ways to store the information about the current state [8], but none of these has been successfully brought to success yet. However, it has been proven in [2] that Reinforcement Learning can work for Tetris, provided an initial hard-coded decision policy based on a heuristic model. Our goal is to provide a fully dynamic decision policy.

## 2 Conceiving the environment

To begin with, we have had to decide how to code our environment. We used python package PyGame to developp our graphical interface<sup>3</sup> and managed to reproduce the original environment.

Here we implemented the full set of tetrominos that are randomly choosen. However, we figured that it would make more sence to begin with a smaller environment since this methods failed to converge with the full environment in all the papers we had read.

<sup>2</sup>In AI, the Q function is a function that evaluates the quality of an action taken in a given state.

<sup>3</sup>We partially followed "freeCodeCamp" tutorial on how to use PyGame for the developpment of our environment but is was only to get started, and our implementation widely differs because theirs is not meant to interact with a agent other than a human player.

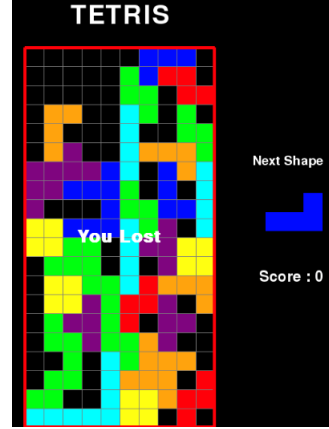


Figure 2: The full environment as originally designed

Indeed, because the grid is of size  $10 \times 20$  the state set is of length  $2^{200}$  which is pretty much impossible to compute, even after finding ways to reduce its size. We decided to work on a smaller environment of the following features :

- A  $6 \times 9$  grid
- One  $2 \times 2$  square tetromino
- One length 2 bar tetromino with two possible orientations
- One  $2 \times 2$  L-shaped tetronio with four possible orientations

The tetrominos we choose are obviously a lot more likely to form complete lines by luck than the original ones, and we can hope it can clear enough lines with random actions for it to learn what action is good.

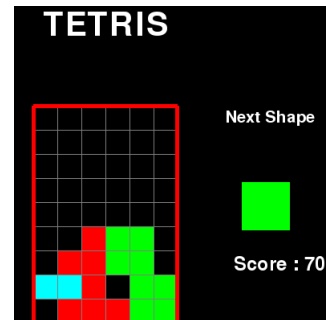


Figure 3: Small environment with the 3 shapes : the green square, the red L-shape and the blue line.

Within the environment, we made sure to use a few global variables such as the list of locked positions in the grid, the number of cleared lines and a the current falling tetromino, so that the agent could easily interact with the environment and train its neural network efficiently.

### 3 Interacting with the agent

To devise a learning agent that can progress through trial and error, we had to take a few decisions before beginning. First we had to choose a few hyperparameter to decide how our model of Q learning would take into account the future rewards, then we add to decide what neural network architecture we wanted and finally we had to adjust the reward policy.

#### 3.1 About Deep Q Learning

The neural network is used to approximate the stationary Q function

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[ \sum_{i=0}^{\infty} \gamma^i r_{t+i} | s_t = s, a_t = a, \pi \right]$$

Where  $s_t$  is the state at time  $t$ ,  $a_t$  is the action taken at time  $t$ ,  $\pi$  is the policy function indicating what actions to take given the states,  $r_t$  is the reward obtained by taking action  $a_t$  at the state  $s_t$  and  $\gamma$  is the discount factor for the future gains. The Q function is thus the highest expected sum of discounted future rewards achievable by following a fixed policy. From this notation follow the Bellman equation that is satisfied by  $Q^*$  :

$$Q^*(s, a) = \mathbb{E} \left[ r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

Where  $s'$  is the resulting state from taking action  $a$  on state  $s$ , and  $a'$  is the action taken given  $s'$ .

This equation allows us to regularly update our current Q function and expect it to eventually converge towards  $Q^*$  :

$$Q(s, a) = r + \max_{a'} Q'(s', a')$$

Where  $Q'$  is a the latest version of the neural network. This version is updated every 200 iterations<sup>4</sup> to prevent instability while training and prevent the small disturbances to impact the convergence of the process.

#### 3.2 The neural network

**Formalizing states and actions** First of all, in order to minimize the size of the input data and get significantly lower training times, we decided not to use a convolutional neural network and the raw pixels as input, but rather use the RAM of the program.

<sup>4</sup>This number is a hyperparameter that need fine tuning but it is a widely used size of minibatch to train our neural network and hope that the stochastic gradient descent converges.

Since we designed the environment ourselves, we were able to access all the required information and decided to use a 28-uplet as input.

- The first 6 digits are the height of the six columns
- The next 6 digits are the absciss of the falling piece (1 if on only if the corresponding column)
- The next 9 digits are the ordinate of the falling piece (1 if on only if the corresponding line)
- The next 7 digits are to describe the shape and its orientation

Using only the highest piece in each column is not bothering for our purpose since what happens beneath does not change much the action to take in order to clear the highest line. Some papers mention taking into account the number of holes in order to add a penalty when too many holes occur but we figured it was too complicated for our small environment and would only slow down the training. Each falling piece has one coordinate and one shape attribute ranging from 1 to 7. The agent has no knowledge of the next piece (one-piece approach).

The set of possible actions is only key up (rotate piece), down (move down piece), right and left (move the piece sideways).

**Network architecture** Since we do not have convolutional layers, we decided to have two hidden fully connected layers with 20 and 15 nodes leading to the 4 nodes output. We used non-linear ReLU activation function except for the output since it is a simple enough architecture that has proven to be efficient for many Reinforcement Learning agents learning to play easy games. Unfortunately, we do not yet have the time nor the expertise to try more advanced architectures, even though it could probably give better results.

We used a *stochastic gradient descent* optimizer training the network on minibatches of size 200 (that hyperparameter obviously calls for some serious tuning but we will require more training time to get there), at the same time at which we refresh the current Q estimator.

Besides, we use an epsilon-greedy policy to make sure we explore the possible actions enough not to converge to fast on a bad policy. We go from 0.9 to 0.15 through 500000 minibatches.

### 3.3 The reward policy

Each time the environment asks the user for an action, we intercept this function and call our own function that gets the desired input for the neural network from the environment, finds out what action to take, sends it to the environment, calculates the rewards and potentially updates the  $Q$  policy and the neural network if we reached the minibatch threshold.

We decided to use a basic reward policy : each time lines are cleared, we reward the number of cleared lines squared times 20, and at each action taken, we punish by a small amount the agent for each height difference that exceeds 2 between adjacent columns. This aims to incentivize it to try and stack the shapes in the most compact way it can and bring falling tetrominos to the lowest point on the grid where they fit.

## 4 Results and improvements

Until now we have had a lot of issues tuning the network and have not yet managed to converge towards an efficient policy.

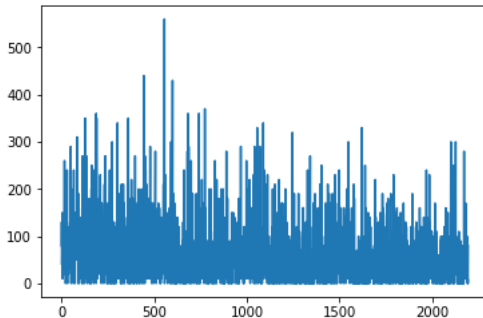


Figure 4: After 50 hours of learning, the network did not efficiently learn what policy to use and still averaged around the same score as the random agent.

We believe the issue comes from the reward policy and the network architecture. We will put a lot of thoughts and training time this week to investigate other solutions and hope to improve our agent's behaviour.

Obviously we also wish to be able to transfer the learning to the full environment.

## References

- [1] Carr, D. (2005). Applying reinforcement learning to Tetris. *Department of Computer Science Rhodes University*.
- [2] Stevens, M., Pradhan, S. Playing Tetris with Deep Reinforcement Learning.
- [3] Carr, D. (2005). Adapting reinforcement learning to Tetris. BSc (Honours) Thesis, *Rhodes University*.
- [4] Flom, L., Robinson, C. (2005). Using a genetic algorithm to weight an evaluation function for Tetris.
- [5] Breukelaar, R., Demaine, E. D., Hohenberger, S., Hoogetboom, H. J., Kusters, W. A., Liben-Nowell, D. (2004). Tetris is hard, even to approximate. *International Journal of Computational Geometry Applications*, 41-68.
- [6] Fahey, C. P. (2003). Tetris. *Colin Fahey*.
- [7] Melax, S. Reinforcement learning tetris example. 1998. URL <http://www.melax.com/tetris>.
- [8] Driessens, K. (2004). On afterstates and learning tetris. In *5th*.