A Reinforcment Learning approach to solve Tetris
Project report

INF581 - Advanced Topics in AI

Abstract. *Here we report on our use of reinforcment learning to train an AI to play Tetris. We began by using a neural network to estimate a Q function that describes the best action to take for each state of the game. Unlike many other approaches of this particular problem, we did not input the raw pixels from the game display, but the information 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. The Deep-Q-Learning approach failed to converge because of a bad reward policy, so we tried a different approach : we decided to fine tune the reward policy itself with Reinforcment learning and then exhaustively try all possible actions to maximize reward. This allowed us to prove Tetris can be played without any supervised learning.*

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. The idea of the game is simple : you begin with a 10×20 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. 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 sevent 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 Reinforcment Learning

We choose Tetris for our Reinforcment learning project since it is complex enough for machine learning techniques to be required to find a really performing solution, 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, Reinforcment Learning appears as a promising way to approximate an ideal solution.

More specifically, we first tried to use a Deep-Q-

learning approach to drive our Markov Decision Process. The idea is to approximate the Q function[1] via a neural network that gets updated as the agent learns through trial and error. However the results were not good so we switched to training the reward policy itself instead. We will explain more in details later.

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 Reinforcment 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 Reinforcment Learning can work for Tetris, provided an inital hardcoded 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[2] 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. Indeed, because the grid is of size 10×20 the state set is of length 2^{200}

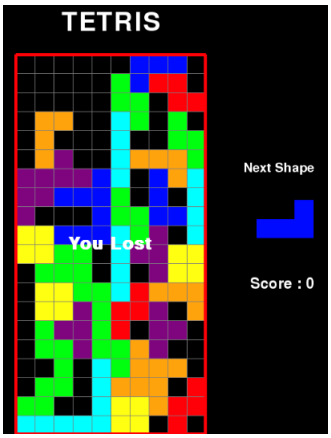

Figure 2: The full environment as originally designed

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 6×9 grid
- One 2×2 square tetromino
- One length 2 bar tetromino with two possible orientations
- One 2×2 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.

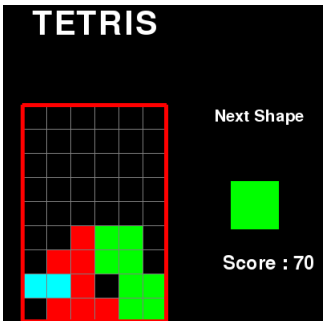

Figure 3: Small environment with the 3 shapes : the green square, the red L-shape and the blue line.

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.

[1] In AI, the Q function is a function that evaluates the quality of an action taken in a given state.

[2] 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 an agent other than a human player.

3 Interacting with the agent

3.1 First try with Deep Q Learning

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.

The neural network is used to approximate the stationnary 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, π 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 γ 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 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 uptated every 200 iterations[3] to prevent instability while training and prevent the small disturbances to impact the convergence of the process.

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 convolutionnal neural network and the raw pixels as input, but rather use the RAM of the programm. 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 ond only if the corresponding column)
- The next 9 digits are the ordinate of the falling piece (1 if ond 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 environnemnt 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 convolutionnal 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 Reinforcment 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.

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.

[3] 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.

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.

This approach did not converge at all after long training time, even for the reduced environment. We suspect that the problem came both from the network architecture and the reward policy. From this observation we decided to adopt a new approach that was a lot easier but also a lot more efficient.

3.2 Training the reward policy itself

Formalizing states and actions First of all we expanded the number of possible actions to all possible lateral translations and all possible rotations : when the shape appears on the board, we allow the agent to apply any combination of moves at once, but we only allow one of those and then we bring down the tetromino until it lands on the locked pieces. The idea behind this is to avoid loosing time as the tetromino goes down 20 rows and we calculate the best move at each step even though everything can be processed from the beginning. This takes a little more time for the agent to choose the action but it saves a lot of time when playing whole games because pieces fall down instantly after the move is choosen, and it also allows the agent to anticipate moves (while it would have required 2 or 3 steps to get to the same state with the previous set of actions). This also makes it closer to how humans process each game state.

Once we modified the actions set, we also adopted a new way to conveive the state. We figured that the maximum height of each column was not enough given the failures of the first approach. This made us consider a more general states representation : from a given grid we calculate four insightful data.

- The sum of heights of each column
- The sum of differences in height between adjacent columns
- Maximum height of the grid
- Number of holes in the locked positions

These four parameters seemed to be enough to caracterize how good a given state is. The big dilemna being to find the trade-offs. For instance minimizing the number of holes or the maximum height of the grid. This is where the Reinforcment learning comes in.

Trial and error To decide the policy we want to use, we instantiate a list of weights $[w^{(1)}, w^{(2)}, w^{(3)}, w^{(4)}]$ that caracterizes the agent's policy. Each of them represents the percentage of importance we give to the four given parameters we spoke about earlier. We instantiate a random repartition of weights and then at each new piece we are doing an exhaustive search of all the possible actions according to the following timeline :

1. We are in state s_t and generate a new tetromino.
2. We try every possible action a_t on a virtual grid and for each move we compute the four parameters $[s_t^{(i)}]$ = [sum_height, sum_diff, max_height, holes] and we associate a score to each action with the weighted sum

$$S(s_t, a_t) = -\left[\sum_{i=1}^{4} w_t^{(i)} s_t^{(i)}\right]$$

3. We choose the move that gives the highest score and bring down the piece to the bottom of the grid after taking the corresponding action on the tetromino. We clear the completed lines if necessary.
4. We update the weights from step t to $t+1$

$$w_{t+1}^{(i)} = w_t^{(i)} + \alpha w_t^{(i)}(r_t - s_t^{(i)} + \gamma s_{t+1}^{(i)})$$

   Where α is the learning rate that we chose equal to 0.01, γ is the discounted expected reward that we set equal to 0.95 and r_t is the reward that chose equal to number of cleared lines squared diminished by the increase in total height[4].
5. We move on to next piece.

This approach allowed to obtain an efficient scoring system to evaluate how good a state is and hence decide the best action to take. Just like in the Deep Q Learning approach we used an -greedy policy to make sure to explore enough poissibilities.

4 Results and improvements

The DQN approach gave terrible results that do not deserve to be presented. However we obtained good results with the second approach. The training was fairly quick and the policy converged within a few hundred games (less than an hour of training) with both the reduced and full environment.

To try out the efficiency of the method, we began with the reduced environment.

[4]Obviously we began by setting the reward equal to the number of cleared lines squared added to the increase in general score of the grid, but as we tried many possibilities, only taking the sum of height into account gave way better results.

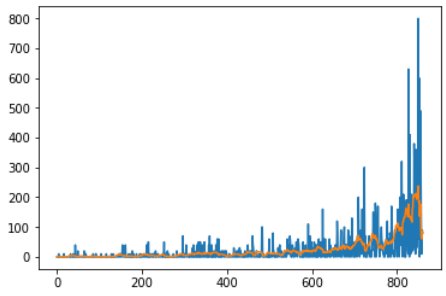

Figure 4: Score evolution with the reduced environment with averaged curve to get a better idea of the general trend.

It worked really well so we decided to apply it to the full environment and really see how it reacted to varying hyperparameters.

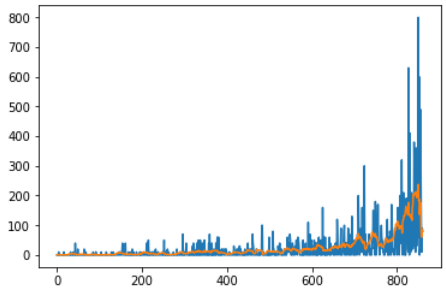

Figure 5: Score evolution with the full environment.

First we fathomed that the variance of the scores is huge, which makes it difficult to evaluate the quality of the agent. However, the averaged score is explicit and shows that the learning is efficient. Here are a few remarks we drew from experimenting with the agent.

- Keeping a high ϵ parameter is important because the weights vector can easily converge towards a non optimal solution and be stuck around it.
- Depending on γ, the discount parameter that caracterizes the importance of the future rewards, the policy will converge to a different result. For instance, a high gamma factor (above .9) will lead to a tolerance to holes and a high limitation of total height that allows the agent to clear multiple lines at once, while a low gamma factor will put more emphasis on filling lines one by one without ever getting to high on the grid.
- The game often puts an emphasis on limiting the sum of heights of each columns, but it fails to take into account when the pieces reach high rows. For instance, it will continue to keep an empty column even if it gets really high, which is often what causes it to lose because no straight line tetromino appears. A real player would be okay to sacrifice the highest lines and clear them one by one instead of trying to clear multiple lines at once with such a high risk of not getting the needed tetromino.

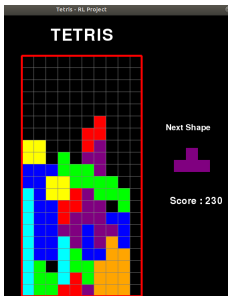

Figure 6: Here we can see that the game willingly lets an empty column to allow a bar to fill multiple lines at once.

Obviously many improvements could still be performed. First we know that this is not the most impressive use of RL since we gave the agent the "idea" that the four parameters we calculate are the interesting ones and it did not figure it out itself. We hope to have time to modify our DQN approach to incorporate the optimal policy computed with our current agent and then obtain a fully non-supervised agent that performs very well.

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., Hoogeboom, H. J., Kosters, 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.*