

## Tetris with Reinforcement Learning

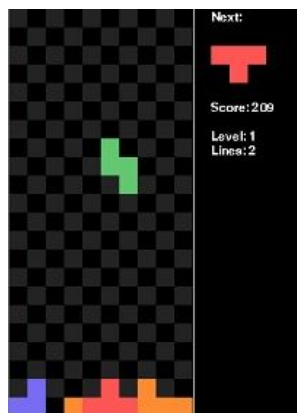
### Introduction

#### Overview

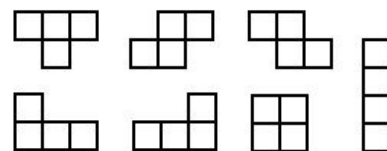
The objective of this project is to train an AI agent to play tetris using reinforcement learning. This technique consists basically on training a software agent that takes actions based on some state of the environment and tries to maximize what it considers a reward.

Tetris is a very popular game in which predefined stones, made of four blocks, fall - once at a time - from the top of a grid board (Image 1). While the stones are falling the player can rotate the stone 90 degrees and move it to the sideways. The player scores when a complete line of blocks is formed on the grid. This line is then cleared and the blocks above this line fall. The game is over if the stack reaches the top of the board. It is a fun game to play and watch, especially if the player is in a high level - which makes the stones fall faster - and it motivated me to try to build an agent that could learn to play a variation of this game.

Many studies on Tetris have already been made to try to analyse its strategies, such as Brzustowski's [1] and Heidi's [2], that showed that it is impossible to play a game of tetris forever, because there is a sequence of pieces that will eventually end the game. Some authors have also explored the artificial intelligence applications on this game, such as Böhm, Kókai and Mandl [3], who successfully used an evolutionary algorithm and Stevens and Pradhan [4], who trained an AI agent to play tetris using deep reinforcement learning.



*Image 1. A version of Tetris.*



*Image 2. The types of stones of the original tetris.*

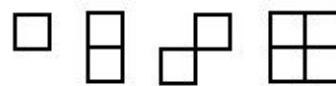
## Problem Statement

The original Tetris exhibit a grid board of 10 columns by 20 rows and seven types of blocks (Image 2). This game presents many combinations of movements, stones and grid board states, making the process of training slow. In order of making the process of training faster, and still graphically visualizable, I have decided to make a lighter variation of this game, with less possible combinations of movements and stones. This change also lessen the number of possible game states.

In this version I will have only six columns (Image 3) and the four types of stones shown below (Image 4).



*Image 3. A new variation of Tetris.*



*Image 4. The four types of stones of this project of Tetris.*

The goal of this project is to make the learner agent play this variation of the game and compare with an agent that chooses its actions randomly.

## Metrics

In Tetris, every time the player clears a line he receives more points in the score and better he seems to be playing. Therefore, an intuitive metric of the performance of an agent playing Tetris is the number of lines it has cleared.

## Analysis

### Algorithms and Techniques

This project uses Q-learning, a model-free reinforcement learning technique, in which the learning agent is not going to learn a specific model of state transitions and rewards. Instead, it tries to discover a policy

function ( $\Pi$ ) which maps the best action (a) to perform in a specific state (s) of the environment by trial and error (I).

$$\begin{array}{l} \Pi : S \rightarrow A \\ s \in S, a \in A \end{array} \quad (I)$$

The learner agent uses a Q-function that provides a positive or negative reward (r), depending on the state and action taken (II). Since we are using a deterministic environment, where the same action in a specific state leads to the same result, the agent is able to learn the action that returns the maximum reward at certain state (the best policy).

$$\begin{array}{l} Q : S \times A \rightarrow \mathbb{R} \\ r \in \mathbb{R} \end{array} \quad (II)$$

The actions the agent chooses depends on the exploration factor, epsilon, that determines the probability of the agent choosing a random action instead of the action indicated by the best policy. This value is important in reinforcement learning, because it needs to be configured in a way which results in a balance between the exploration of unknown responses to an action and using what the agent have already learnt.

## Benchmark

Since this version of Tetris was made specifically for this project, it is difficult to compare its results with another published works. Therefore, I will use this same version of Tetris and compare the number of cleared lines from two different agents: a learning agent and a random action agent.

Previous tests with an agent playing randomly showed it was very difficult to it to reach level 2. Based on that, I've decided that when the learning agent reaches level 5 (clears 24 lines), the trial could be considered a success, since it is a difficult level to reach just playing randomly moves and could clearly show that the agent has learnt something. After the training I will compare the number of successes of the trained agent and the agent that played randomly.

## Methodology

### Data Preprocessing

Since this project doesn't use any data sets, the only preprocessing needed before training was to make some changes in the environment, as described on the item below.

## Environment

This project uses an open version of the tetris [5] implemented using the pygame library. It was then modified to enable the purposes of this project. The changes that were made are basically: 1 - the number of columns of the grid to six; 2 - the types of predefined stones (Image 4); 3 - new stones start to drop in the grid from the leftmost top position instead of the center to make the implementation simpler; 4 - increased speed (fps), to make the training faster; 5 - after changing level the speed does not increase, because it is already faster (item 4); 6 - a display showing the trial and training / testing information was added; 7 - code was added in the application to enable the learner agent to execute actions and gather information about the state of the game.





## Actions

When a stone is falling in the grid board the player can execute a series of moves, such as rotating the stone, moving it to the sideways and down. For this project I will consider a sequence of rotating and moving right (since the stone starts at the left side) as an action. It is represented as a tuple  $(n, m)$  of the number of clockwise rotations  $n$  and the number of moves to the right  $m$ . As an example, the action  $(1, 5)$  means the stone should rotate 90 degrees once and move five cells to the right. After the action is performed the stone is instantly dropped and the player can't execute another action on that stone.

The order of execution of the moves of an action always starts at rotating the stone clockwise  $n$  times, then it is followed by moving the piece to the right  $m$  times and ends with an instant drop.

## Stones

This project presents only four types of stone and to be easier to reference each one of them, they will be called as “.”, “I”, “/” and “O” stones. Each stone can perform a different number of possible actions, according to the number of possible rotations that can change its format in the board and the number of possible moves to the right (or it can stay in the initial column). This results in a different number of possible combinations of moves, and therefore, the number of actions each stone can perform. (Table 1).

Stone	Alias	Rotations	Moves right	Max. Height	Actions (approx.)
	.	0	6	1	6
	I	1	6 (approx.)	2	12
	/	1	5	2	10
	O	0	5	2	5

*Table 1. Number of possible actions of each type of stone.*

## State of the Game

The state of the game are a series of attributes that can be used to help the agent choose an appropriate action for the current state. Besides choosing the action, it is also useful to get its attributes to analyse if the action taken was bad or good, for example, and calculate the reward related to the reinforcement learning.

This project uses as game state the current stone falling and a light snapshot of the board - the skyline. It considers the height of the top block of each column and normalizes it to column to the left. So it is a sequence of 6 numbers, where the leftmost column always receive a value of 0. If the column to the right is one block higher it receives the value of plus one. If it is two blocks lower it receives the value of minus two. Also, to minimize the combinations of possible skylines, it is considered the minimum and maximum value of a column respectively the negative and positive values of the maximum height of the current stone (see Table 1).

In the example of Image 5, the skyline would be (0, 0, -2, -2, 1, 0). The first column is always zero; the second is in the same level, so it stays zero; the third presents a great cliff, and would be -6, but since the maximum height of the I stone (the one falling) is 2, it is -2; the fourth has a cliff of -2; the next column is one block higher, receiving 1; and the last stays in the same level, so it receives a value 0.



*Image 5. Example of skyline in red.*

## Reinforcement Learning

Before the training, a Q-table that stores the results of the Q-function for each state and action, is initialized with zero values. Then after each action the table is updated using the Q-function below (III), where the  $Q(s, a)$  on the right side of the equation represents the previous value on the Q-table and  $r$  the reward for the action taken on that state.

$$Q(s, a) = Q(s, a) + \alpha(r - Q(s, a)) \quad (III)$$

In order to be able to learn new information and also maintain the learning from previous iterations, I have decided to consider the learning rate alpha as 0.5 and since I am not considering the importance of future rewards at each iteration - and to keep this project simpler - the discount factor is not being used.

In reinforcement learning there needs to be a balance between the exploration and exploitation. Thinking about that I decided to test two different functions that decrease the exploration factor epsilon as the number of trials increase: a linear decreasing function (IV) and a cosine function (V). Both of them use the number of the current trial (n) and the number of total trials (t). This value represents the probability of the learning agent choosing a random action instead of the action that returns the maximum reward on that state and it is given by the best policy (VI).

$$\epsilon = 1 - \frac{n}{t} \quad (IV)$$

$$\epsilon = \cos\left(\frac{\pi}{2} \times \frac{n}{t}\right) \quad (V)$$

$$n, t \in \mathbb{N}, n \leq t$$

$$\Pi^*(s) = \arg \max_a Q(s, a) \quad (VI)$$

## Rewards

Since this project uses reinforcement learning the combination of metrics I have tried are: negative reward if the action increases the number of holes; negative reward if it increases the stack height; negative reward if it results in a game over; positive reward if it clears a line; and positive reward if it lessen the number of empty cells.

To clarify, the number of holes represents the number of empty cells with a block over it (Image 6). The stack height is the size of the highest column of the stack (Image 7). Empty cells are cells with no blocks and under the stack height level (Image 8).

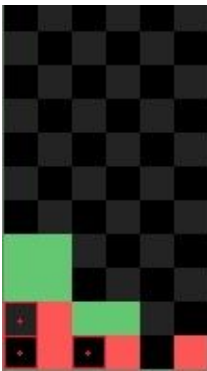


Image 6. Example of a board with 3 holes.

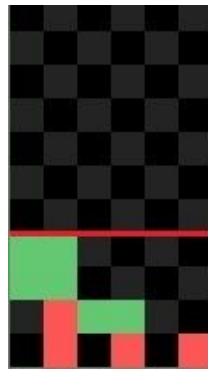


Image 7. Example of the stack height.



Image 8. Example of a board with 14 empty cells.

## Number of Trials

To be able to learn, the agent should go through a fair amount of the combination of game states (stone and skyline) and actions. Combining the information of number of possible actions of each stone and skyline combinations, it results in a total of approximately 85.000 possible states and actions the agent has to learn. Considering the mean of 50 stones played in a single game trial with randomly chosen actions, it would take at least 1.700 trials ( $85.000 / 50$ ) to go through each state and action. Since it has a lot of repetitions, I decided to test with 2.500 and 5.000 trials of training.

Stone	Skyline range	Combinations	Approx. of skyline combinations and actions
.	( 0, [-1, 1], [-1, 1], [-1, 1], [-1, 1], [-1, 1] )	$3^5 = 243$	$6 * 243 = 1458$
I	( 0, [-2, 2], [-2, 2], [-2, 2], [-2, 2], [-2, 2] )	$5^5 = 3125$	$12 * 3125 = 37500$
/	( 0, [-2, 2], [-2, 2], [-2, 2], [-2, 2], [-2, 2] )	$5^5 = 3125$	$10 * 3125 = 31250$
O	( 0, [-2, 2], [-2, 2], [-2, 2], [-2, 2], [-2, 2] )	$5^5 = 3125$	$5 * 3125 = 15625$
		9618	85833

*Table 2. Combinations of game state and actions.*

## Tests

After each different training I would test with 100 trials of the agent playing randomly and 100 trials of the agent playing actions based on the Q-table that it had learnt on that training.

## Implementation

This project is composed of three python files: `learner_run.py`, `tetris.py` and `learner.py` and they relate with each other as shown in a very simplified way in Image 9.

### Learner\_run.py

`Learner_run.py` is the main file for running the training and tests trials. At its beginning it has a section to configure whether it should train or just test, the number of training and test trials and the number of lines it should use as threshold to consider a trial a success.

At the *run* function it processes the training trials (if configured to do so), the test trials with an agent playing random actions and the test trials with the learning agent. When running these trials, this function also records some information like the number of the trial, the number of holes it ended with and the number of lines it cleared for further analysis. After running a different training and tests I would

manually copy these log files to different folders (e.g.: lin\_2500 folder contains logs of the training and tests trials of a 2500 trials training with a linear exploration function).

## Tetris.py

Tetris.py contains the implementation of the Tetris game with some modifications on the environment (as described at the “Methodology - Environment” section). It was a simple implementation of Tetris [5] build using the pygame library and the main change to enable the learner agent to get the current game state, rotate and move stones and learn, is between lines 319 and 333.

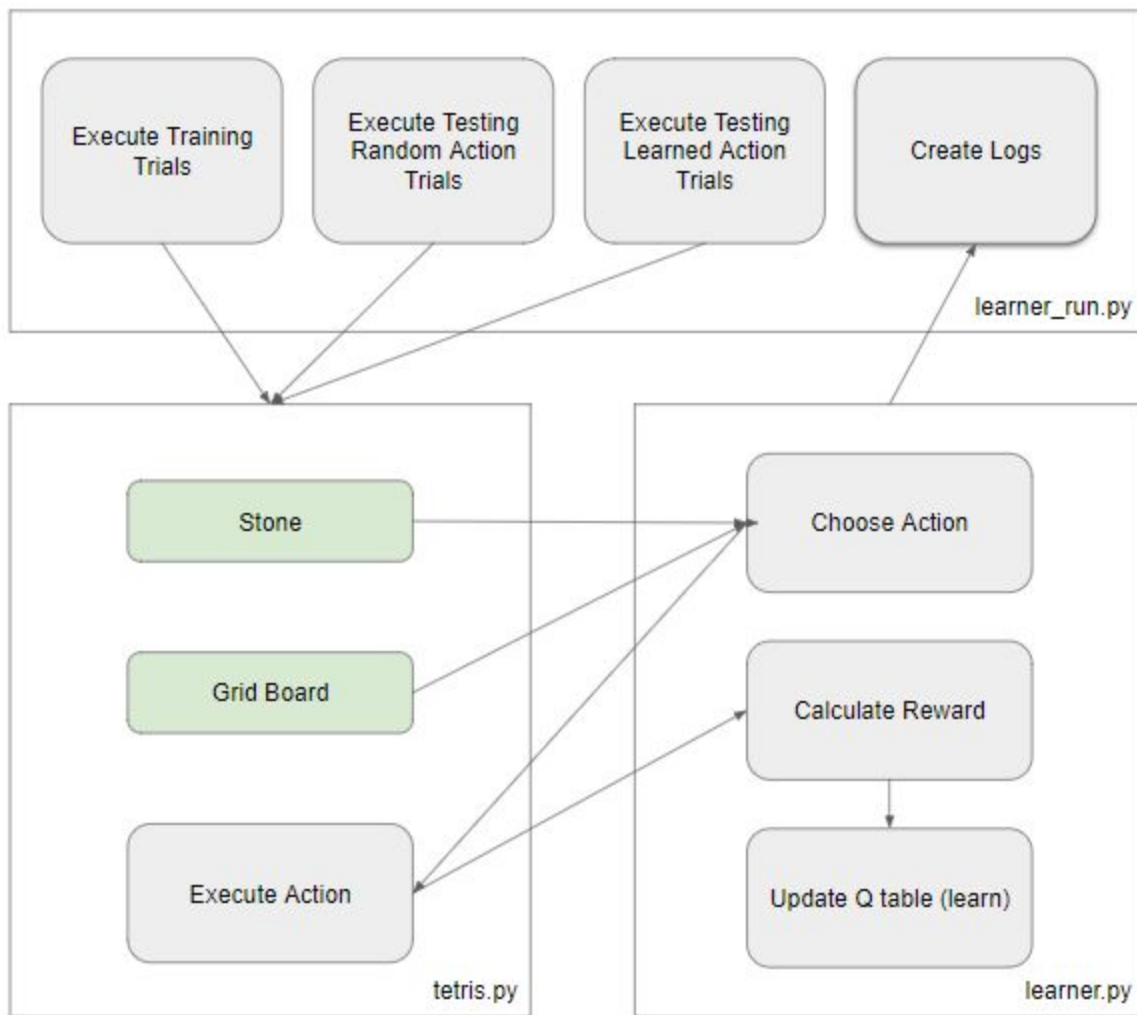
## Learner.py

Learner.py contains the implementation of the learner agent class and also some other classes it uses as support, such as GameState, Stone, Stones and Action.

Basically, at the beginning of each training trial the agent updates its exploration factor and then at each new stone it gets the current state of the game (based on the stone and gridboard / skyline), creates a Q-table (if it doesn't already exists) of that state and possible actions, chooses an action, perform the action, get the state of the game after the action to calculate the reward and then learn from it.

For the test trials, at each new stone, the agent gets the current state of the game, chooses an action (random or from the previous learnt Q-table) and perform the action.





*Image 9. Simple representation of elements and responsibility of each file of the program.*

## Results

The data used in this section are from the logs generated from `learner_run.py` and can be found in the folders inside “report” directory. The charts and data from this section, along with the code that generated them can be found at the jupyter notebook file `Tetris_with_Reinforcement_Learning_Results` (ipynb or html).

## Training

Analysing the combinations of different number of training trials (2500 and 5000) and learning exploration functions (linear decreasing and cosine) we can observe - as expected - that the more trials the agent had gone through, the more lines it is able to clear. This number of lines cleared increases rapidly

especially at the last 20% training trials, when the agent makes less random actions. Also, the stack it builds gets more “dense”, with less empty holes in it along the time.

While increasing the number of trials from 2500 to 5000 using cosine exploring function didn’t show much difference, using the linear function exhibited a great improvement on the number of lines the agent was able to clear - from over 40 to approximately 120.

The 5000-trial using cosine function also seemed to reach a baseline of the number of lines it cleared near to 60.

The function of number of holes in all chart seem similar, going from near 40 to approximately 12.

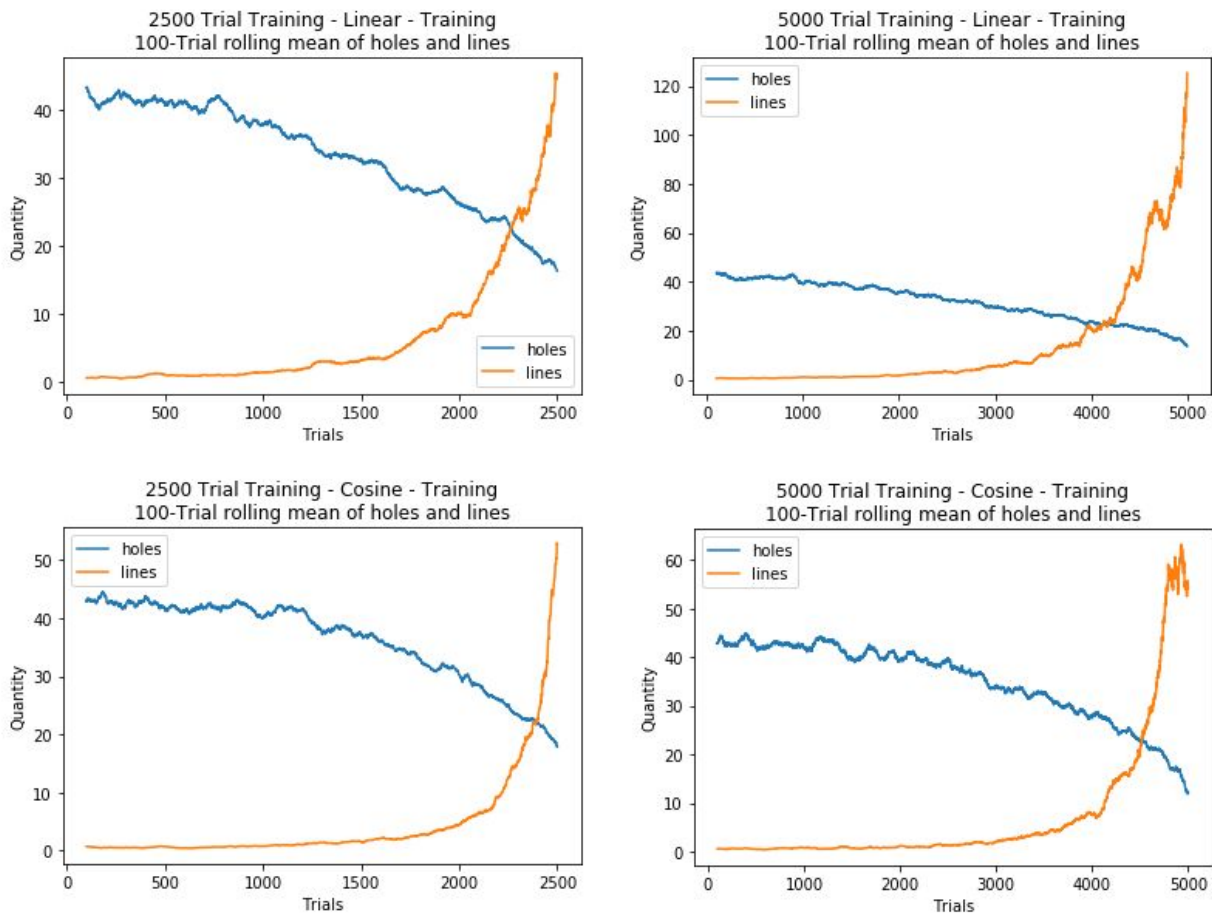


Image 10. Charts of training trials with different configurations.

## Tests

From each configuration of training it was also possible to count the total number of the combinations of state (stone and skyline) and actions. Using the total possible combinations given in Table 2, it is possible to calculate the percentage of covered data from each configuration (Table 3).

Comparing the configurations, the more trials in training, the more combinations the agent had covered. The cosine function type also had more combinations than the linear one. The unseen data is between 13 and 23%.

Configuration	Combinations of stones * skylines * actions covered	Combinations covered (%)
Linear, 2500 trials	66786	77.80
Linear, 5000 trials	73570	85.71
Cosine, 2500 trials	68114	79.35
Cosine, 5000 trials	74477	86.76

*Table 3. Total combinations covered by each configuration.*

Like observed in training, the 5000-trial linear function learning agent was the one able to clear more lines. In a 100-trial test it was the one with the greatest percentage of success (88%).

One unexpected observation was that the cosine exploration function agent did poorly with more training trials considering the mean of number of lines and success rate. It can be explained because, even if it has the highest value of combinations covered, it still has 13% of unseen data, that added to the randomness of the sequence of the stones of the game can result in bad test trials. Even with this result, this configuration is still significantly better than the random agent.

The random agents were very similar, with a mean of holes near 42 and 0 success. Therefore all configurations of the learning agents did significantly better than the benchmark - random agent.

Exploration Function	N° of Training Trials	Random Agent Test			Learning Agent Test		
		Holes (mean)	Lines (mean)	Success (%)	Holes (mean)	Lines (mean)	Success (%)
Linear	2500	43.86	0.61	0	12.59	58.03	77
Linear	5000	41.93	0.56	0	12.05	138.98	88
Cosine	2500	43.61	0.68	0	14.39	68.63	78
Cosine	5000	43.73	0.5	0	9.24	51.55	66

*Table 4. Results of tests from different training configurations.*

## Conclusion

### Summary

The best configuration for training tested in this project is the 5000-trial with a linear decreasing exploration function, since it did more lines and 88% of the test trials reached level 5 at the game. The

second best was the 2500-trial cosine exploration function, presenting 78% of success. It was not as good as the best configuration, but since it takes less trials to learn (consequently less time) it also did a very good job, and better than I had expected.

## Visualization

A small video of the agent trained with 5000 trials and linear exploration function was included in the “report” folder (pygame\_window\_playing\_tetris.mp4) to show some seconds of it playing tetris.

## Improvement

One point to consider as an improvement for this project is to use convolutional neural networks to read the grid board, instead of using the skyline. This way the agent can learn patterns of the board that repeats in different locations (columns and rows).

## Reflection

One interesting aspect of using reinforcement learning is to apply the metrics in which the agent is going to be rewarded. It is intriguing to transform something that a human player knows “by instinct” in metrics that are represented by numbers.

The difficult part of the project was to keep the trials visualizable, in a way I could follow the training trials whenever I wanted and still keep it in a speed that the whole training process wouldn’t take more than a few hours.

Overall, it was a fun project to implement and very satisfying to see the agent playing the game and actually clearing some lines. Even better, clearing more than 700 lines in some of the trials.

## References

- [1] JOHN BRZUSTOWSKI, Can You Win at Tetris?, The University of British Columbia, March 1992.
- [2] HEIDI BURGIEL, How to Lose at Tetris, The Mathematical Gazette Vol. 81 No. 491, July 1997.
- [3] NIKO BÖHM, GABRIELLA KÓKAI, STEFAN MANDL, An Evolutionary Approach to Tetris, The Sixth Metaheuristics International Conference, August 2005.
- [4] MATT STEVENS, SABEEK PRADHAN, Playing Tetris with Deep Reinforcement Learning.
- [5] Original implementation of Tetris used in this project: <https://gist.github.com/silvasur/565419>