Reinforcement Learning

Match the Tiles

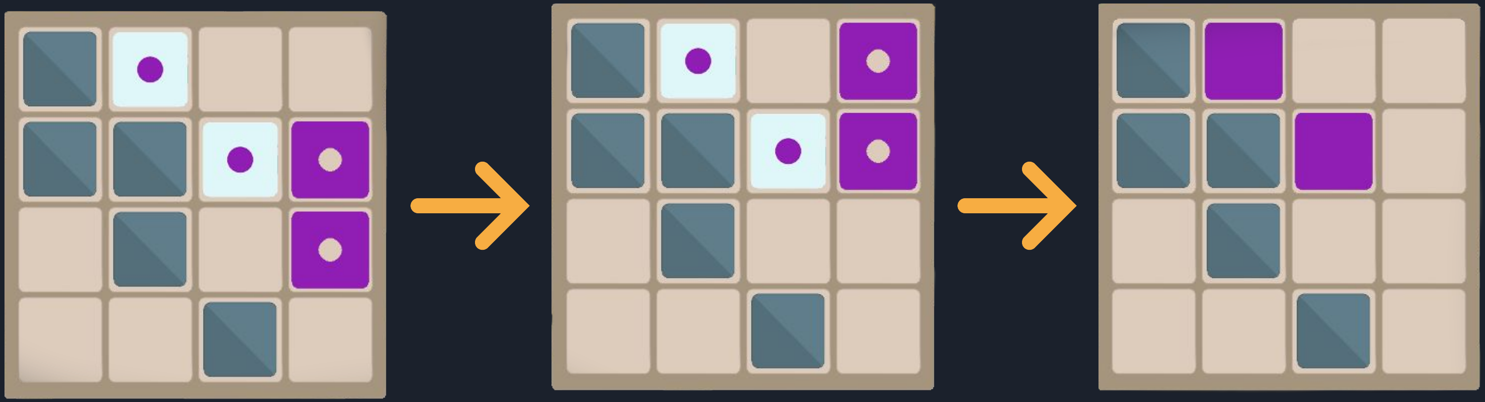
IARTistas
João Diogo Martins Romão, up201806779
João Diogo Vila Franca Gonçalves, up201806162
Rafael Valente Cristino, up201806680

Game Specification

Type of board and pieces | The board is represented by a NxN grid containing obstacles that block the pieces' movement and a target position for each piece (position of the board where each piece must be to win the game). There are K, K<N-1 pieces and each occupies one position (where no obstacle exists) in the grid.

Rules of movement of the pieces | The player moves all pieces simultaneously. They can only move left, right, up or down. Each piece's movement stops if it encounters another piece, an obstacle or the board's border.

Conditions for ending the game | To win the game all pieces need to be in their respectively colored target position.



Tools and Algorithms

To address this reinforcement learning problem we chose the ML-Agents library provided by Unity.

ML-Agents provides the implementation of two deep reinforcement learning algorithms:

- Proximal Policy Optimization (PPO)
- Soft Actor-Critic (SAC)

The algorithms are executed by ML-Agents using our implementation of an *Agent* - a class that uses the following methods:

- ***OnEpisodeBegin*** - is executed at the beginning of each episode (allows us to do any setup that is required).
- ***OnActionReceived*** - is executed after an agent makes a decision. In here we execute the action that was chosen and give the agent its rewards.
- ***CollectObservations*** - is executed when the agent needs to observe the environment. In here we can give it what we think it will need to make its decision.

Developed Work

- Implementation of the game in the Unity environment.
- Adaptation of the game to the ML-Agents library:
- Definition of the agent and its required properties by ML-Agents.
- Definition of the agent’s observations and actions according to ML-Agents specification: the agent will need to observe the game state (the board with the pieces and obstacles) and to choose from among 4 actions (move right, left, up or down).
- Tested some reward functions, such as giving a negative reward on each move (with the objective of obtaining optimal solutions to the levels of the game - in the least amount of steps possible) and others that include giving negative rewards when the agent chooses an invalid move.

First approach and interesting conclusions

As a first approach we started by giving a negative reward of -1 for each move that was made (that effectively moved the pieces) so that the reinforcement learning algorithm would find the shortest path to the solution. By mistake, we were not attributing a negative reward when the agent opted by a move direction that does not change the state of the game (an invalid move).

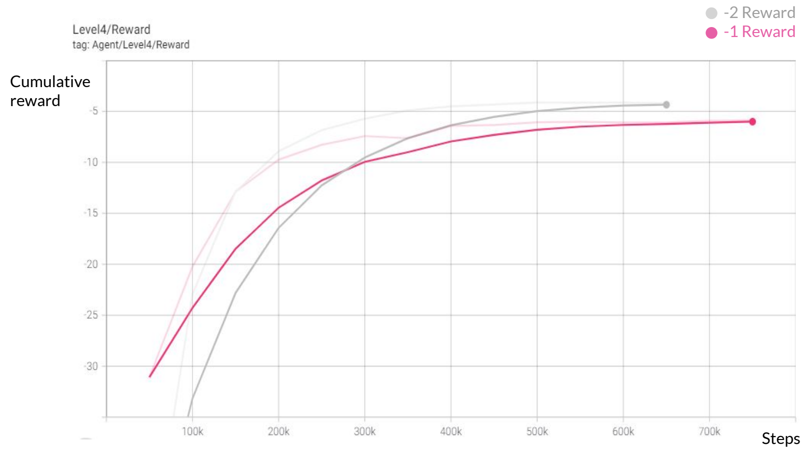
This led to an interesting learning approach by the agent, that started to select these invalid moves and not changing the state of the game (the cumulative reward was still the same and not lower!).

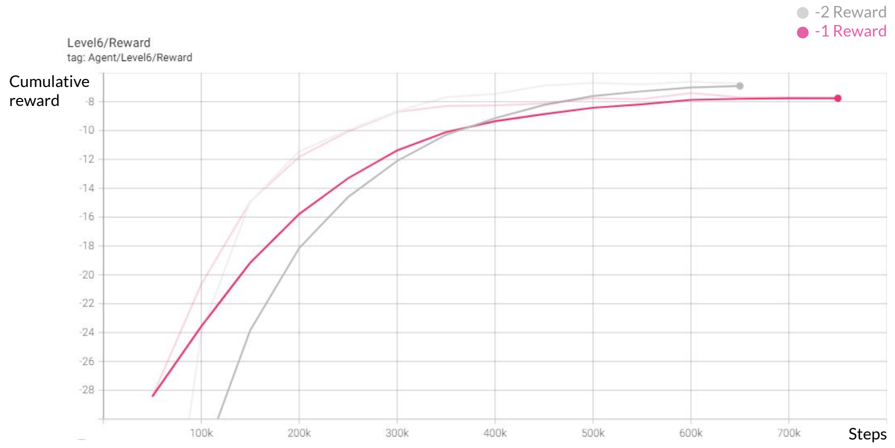
If the agent only chose invalid moves throughout an episode, that episode would have a cumulative reward of 0 which is the minimum possible reward, so the agent would always opt for this strategy.

Improving on the first approach

To improve on the first approach, we decided to also give a negative reward of -1 when an invalid move was chosen. That allowed the agent to actually learn how to reach the goal while executing valid moves.

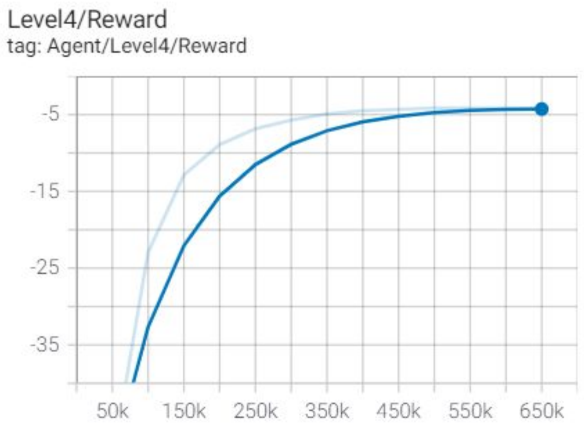
Predicting that if we increased this value we would obtain better results we changed it to -2, and the experimental results show that most of the times using this value made the agent learn better (the agent converged to the optimal solution faster).

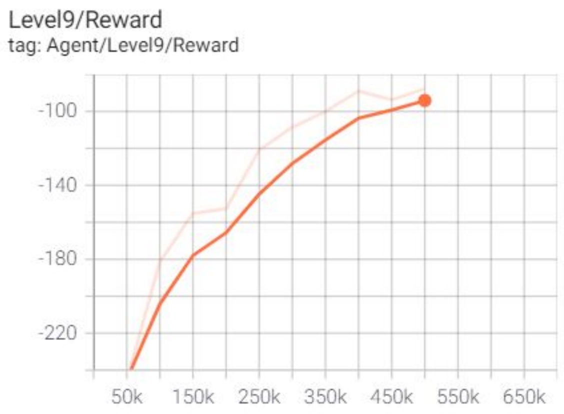




Proximal Policy Optimization (PPO)

Being PPO the ML-Agents default algorithm, it was the most used. The algorithm managed to train models capable of solving the proposed puzzles, however it faced some difficulties training levels with higher difficulty (with a bigger minimum number of plays needed to solve the puzzle) and bigger dimension.

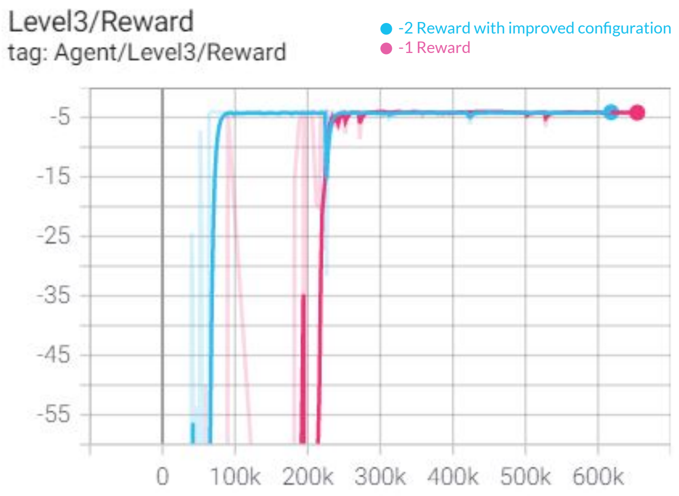


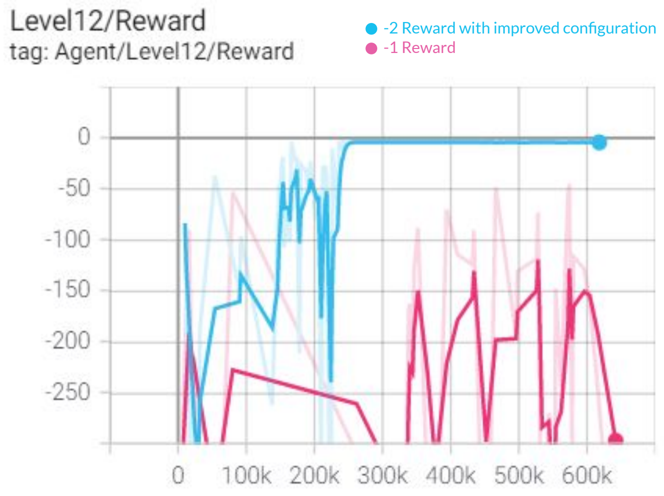


Soft Actor-Critic (SAC)

We trained twice with the SAC algorithm. On the first one we gave a negative reward of -1 when he tries to make an invalid move. On the second one we changed the ml-agents configuration file and increased this negative reward to -2.

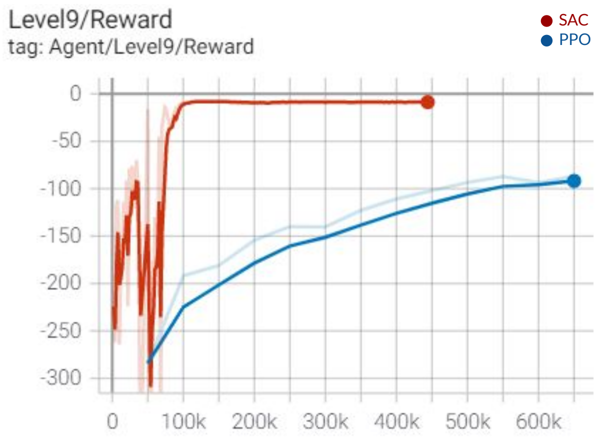
The changes we made to the configuration were: the batch_size from 128 to 256, the buffer_size from 100000 to 200000, the buffer_init_steps from 0 to 1000 and the hidden_units fom 128 to 512.

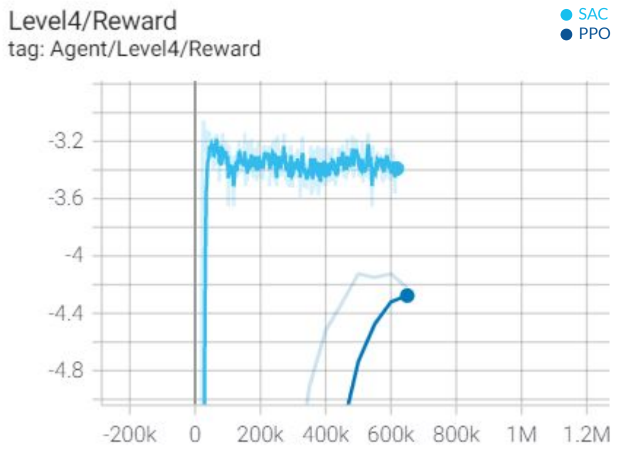




Proximal Policy Optimization (PPO) VS Soft Actor-Critic (SAC)

Comparison of both of the algorithms training with the same reward function (-1 for each move and -2 for each invalid move).





Conclusion

- This project was a great first contact with Machine Learning, more specifically Reinforcement Learning.
- We were able to use different Reinforcement Learning algorithms (PPO and SAC) and test different results.
- We also tested different rewarding functions and were also able to verify the consequences of the tested attributes directly on the behaviour of the agents.

References and Technologies

References

Game | https://play.google.com/store/apps/details?id=net.bohush.match.tiles.color.puzzle

Sliding tiles AI algorithms | https://visualstudiomagazine.com/articles/2015/10/30/sliding-tiles-c-sharp-ai.aspx

Composition of Basic Heuristics for the Game 2048 | https://theresamigler.files.wordpress.com/2020/03/2048.pdf

Deep Copy of an Object with C# | https://stackoverflow.com/questions/129389/how-do-you-do-a-deep-copy-of-an-object-in-net

How to use Machine Learning AI in Unity! (ML-Agents) | https://www.youtube.com/watch?v=zPFU30tbyKs

Technologies

Unity (version 2019.4.21f1) for the graphical interface.

ML-Agents (version 1.0.7 on Unity, version 0.20.0 on pip package manager) to train the models using reinforcement learning.

C# for the game logic and search algorithms.