Deep Reinforcement Learning for the mobile game Triple Town

Andrea Scotti, Sherly Sherly

December 2019

1 Problem description

Triple Town is a turn-based, single-player game in which the player must build a new settlement. The game takes place on a 66 grid of fields on which some tiles are randomly placed.

Players are given random tiles, most often grass tiles, that they must place on the grid. When three or more identical tiles adjoin, they merge into one more advanced tile at the position of the last tile placed: three grass tiles become a bush, three bushes a tree, three trees a hut, three huts a house, and so forth. Merging four or more tiles earns coins or a tile which is held in the inventory located in the sidebar. There are three special tiles: bears, crystals, and imperial bots. Bears move to a neighboring square each turn, blocking building sites until they are trapped. Ninjas act the same, except they can move to any empty square on the board. When they are trapped or an imperial bot is used on them, they turn into a gravestone. Three gravestones make a church, three churches a cathedral, and so forth. Crystals can be used as a wild card to make any match. Imperial bots remove individual tiles, or, if used on bears, turn them into gravestones. The player may keep one tile in reserve in the storehouse, usually located in the top left corner of the screen, and use it when needed.

The objective of the game is to upgrade one's settlement's tiles to as high a rank as possible, earning an accordingly high score. The game ends if all fields of the grid are filled.

The goal is to implement a reinforcement learning agent able to learn to play the game cleverly.

2 Tools

The tools we are going to use are TensorFlow 2.0, Jupyter Notebook, PyCharm and Hopswork.

3 Data

There are no existing data or any offline dataset for this project. The bot will learn the best strategy by interacting with a simulated environment. The data will be generated by the environment and takes the form of a sequence of states and rewards that the algorithm collects every time it takes an action on the environment. The environment has been simplified with respect to the original version of the game. Actually each tile is a number between 1 and 8 and the rules have been simplified as it follows:

- an action can be placing a tile in blank cell or moving a tile from the upper left corner to any blank position.
- when placing a tile changes the grid, such that there are three or more neighbor cells with the same number inside, they will become blank cell except for the last selected one whose number inside will increase by one.

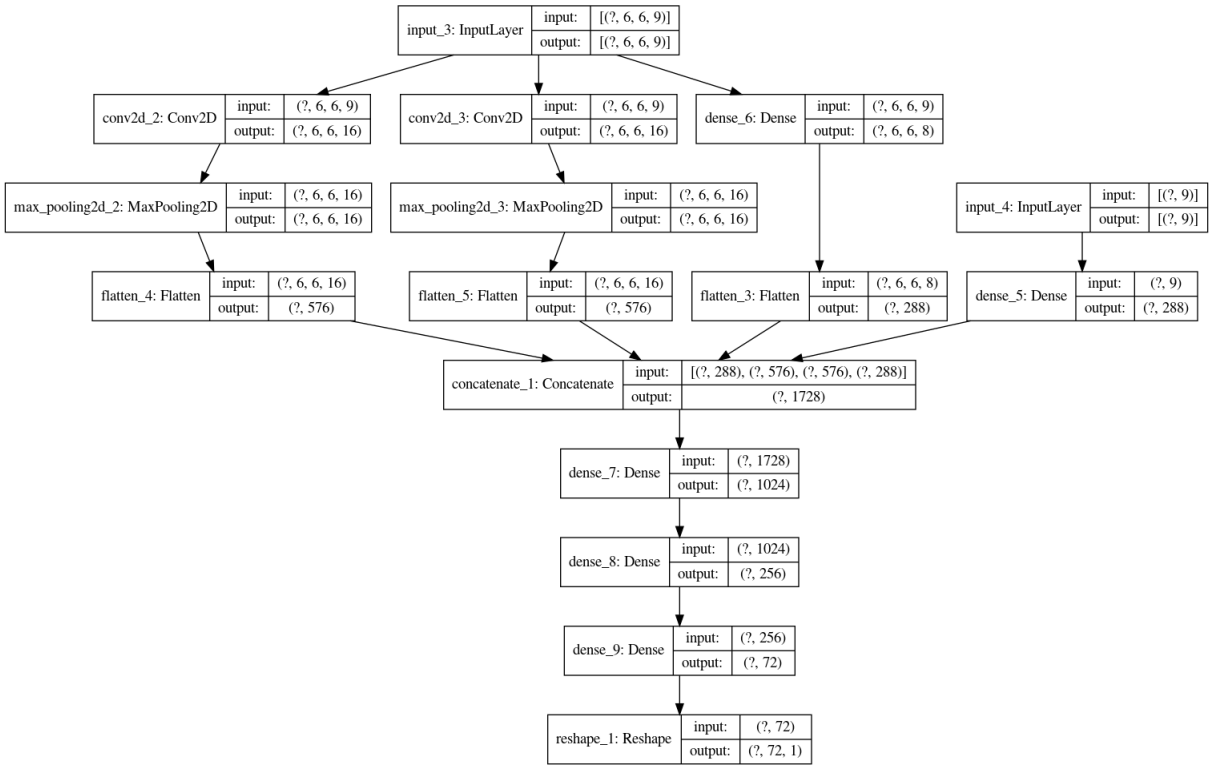
4 Methodology and algorithm

Several implementations have been utilised for this project as listed below:

- Double Deep Q-Learning: the deep reinforcement algorithm.
- Random Policy: At each round the algorithm places a tile in a random cell. This is a baseline algorithm.

- Greedy Policy: Improved version of the random policy algorithm where if there is a chance to merge three tiles, it will opt for that action. This is another baseline algorithm.

The implementation follows the algorithm presented in the paper Double Deep Q Network with the following customization:



- **One hot encoding**: each cell of the grid is an one hot vector where the value one appears only in the position corresponding to the number of the tile in the cell.
- **Modified eps-greedy policy**: instead of following a random action, follow with 0.5 probability a random action and with 0.5 probability a greedy action.
- **Weighted loss**: since the number of actions is very high but for each sample we want to train the network for only one action and keep the Q-values of all the others as they were before, we give a weight of 0.5 to the loss related to the selected action and 0.5 to all the others (in the default version the loss of each action is uniformly weighted by 1/NUM_ACTIONS).
- **Reward between 0 and 1** (not in the final version).

- **Action sampling**: during training, instead of following the action with the best Q-value, choose the action to follow by sampling it according to the softmax probability in order to learn also from potential optimal actions (not in the final version).
- **Learn from complete episode**: after each episode train the network with all the episode (not in the final version).

The main effort of the project was to find the best set of hyperparameters to have a stable and meaningful learning.

5 Code Structure

The code structure consists in a folder TripleTown, which contains all the implementations to run the simulation for the random and greedy agent, and a Jupyter Notebook where the implementation of the DDQN agent is customized to run in Hopswork.

5.1 Classes:

- **Environment**: it represents the environment of the game.
- **Env0**: actual implementation of the environment.
- **Learner**: it represents the agent.
- **GreedyLearner**: implementation of the agent that follows a greedy strategy.
- **RandomLearner**: implementation of the agent that follows a random strategy.
- **DQLearner**: implementation of the agent that follows the DDQN policy.
- **DQLNetwork**: implementation of the DDQN.
- **ReplayBuffer**.

6 Results

We run the simulations over 200 game episodes and averaged the rewards:

- random policy: 222
- greedy policy: 800
- DDQN policy: 320 (after 110h of training, growing linearly by a factor of 30 per day in the last 48h of training)

To generalize well, it requires to be trained on more episodes.

7 Further improvement

In order to speed up the convergence of training and increase its stability, a prioritized buffer is required. Instead of random sampling from the buffer, a good choice is to sample the experiences from which the algorithm can learn more, the ones with higher loss. We didn't proceed with this implementation choice because computing the loss for all the buffer can be very expensive in terms of execution time.

8 Problems

We haven't been able to find a way to save the neural networks weights in Hopswork. This didn't allow us to run the simulations with the trained network after the end of the experiment. To overcome this problem, we print the simulation results every ten game episodes by using the current state of the network.