2048

2048: A Deep Reinforcement Learning Approach

Machine Learning

Bianca Francesco - 1877011
Magnante Andrea - 1876801

Supervised by Prof. Paola Velardi

Dipartimento di Informatica

M.Sc. in Computer Science

Universitá degli studi di Roma, "La Sapienza"

2019/2020

SAPIENZA
UNIVERSITÀ DI ROMA

WWW.UNIROMA1.IT

First edition, March 11, 2020

Contents

1

Introduction

For the final project of the Machine learning course we decided to focus on a branch of machine learning that is very studied nowadays: *Reinforcement Learning*. In particular, we focused on the use of this category of machine learning algorithms in the field of video games (trained players who can play independently trying to get better and better scores). The report will show the solutions addressed to the problem, the technologies used in the realization of the system and the performance evaluation.

1.1 | Reinforcement Learning

Reinforcement learning (RL) is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. RL is focused on learning a series of actions for optimizing an overall reward - for example, winning at a game of chess. So, in practice:

- The loop:
  - Agent perceives the state of the environment
  - Agent acts
  - Agent receives reward/punishment, the state of the environment changes
- The task:
  - Learn a sequence of action and states (e.g. a policy, mapping from states to actions) to maximize rewards

One of the most important algorithm in Reinforcement Learning is ***Q-Learning***. Q-learning is an off policy algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. Let consider the deterministic case of the algorithm for simplicity:

Algorithm 1: Q-learning algorithm (deterministic)

```
Input: Q matrix with all zeros, initial state ;
1   Set current state = initial state;
2   Select randomly a possible action;
3   Compute next state;
4   From next state, find action that produce maximum Q value (based on Q
    table);
5   Update Q;
6   Set current state = next state;
7   Go to 2 until (current state == goal state)
```

In Q-learning, we build a memory table *Q[s, a]* to store Q-values for all possible combinations of s and *a*. If you are a chess player, it is the cheat sheet for the best move. Technical speaking, we sample an action from the current state. We find out the reward *R* (if any) and the new state s' (the new board position). From the memory table, we determine the next action a' to take which has the maximum $Q(s', a')$. In a video game, we score points (rewards) by shooting down the enemy. In a chess game, the reward is +1 when we win or -1 if we lose. So there is only one reward given and it takes a while to get it. However, if the combinations of states and actions are too large, the memory and the computation requirement for *Q* will be too high. To address that, we switch to a deep network *Q (DQN)* to approximate *Q(s, a)*.

1.1.1 | Deep Q-Learning

In many practical applications, e.g., games, the dimension of a Qtable is very large. For example, consider Atari games: If apply a pixel-level processing of board we can have millions of possible game state. This means millions of rows in our Q-table. The intuition is that, in order to learn Q values, we can use neural networks. We train for some state and action, and we can then use the trained network to compute *Q* for any state and action.

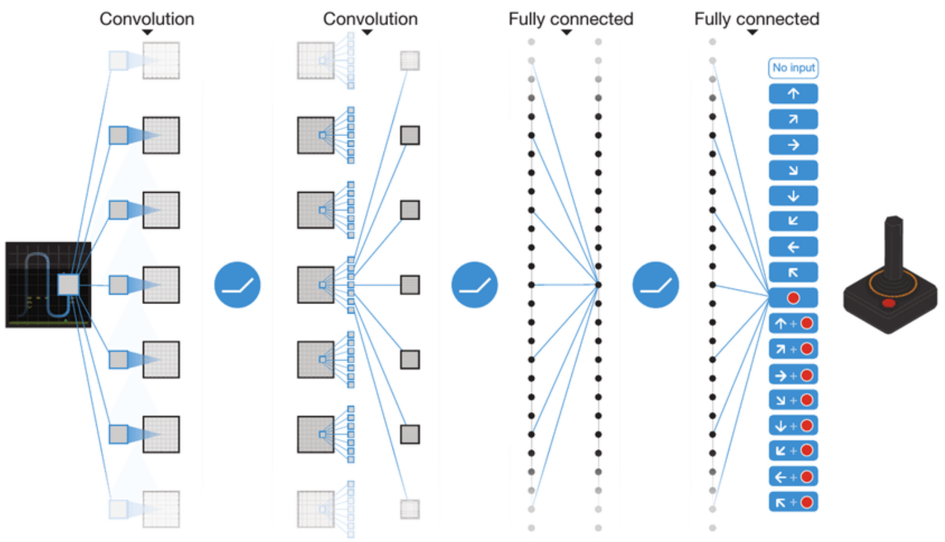


Figure 1.1: Deep-Q-Network architecture.

How does Deep Q-Learning works?
Given a transition **<s, a, r, s′>**, the Q-table update rule in the "classic" algorithm must be replaced with the following:

- 1) Do a feedforward pass on the Deep Neural Network for current state *s* and get predictions for *Q(s,a)*, for any possible action *a*.
- 2) Do another feedforward pass for the next state *s′* and calculate maximum over all network outputs $\boldsymbol{max_{a'}Q(s',a')}$.
- 3) Set *Q-value* "target" (ground truth) for *Q(s,a)* in step 1 to: $\mathbf{[r + \gamma \cdot max_{a'}Q(s',a')]}$ (use the max calculated in previous step)
- 4) For all other actions in step 1, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs. Therefore after step 4 we only have one error on the prediction for *Q(s,a)*. To compute the error on *Q(s,a)*, use the standard loss (error) function:

$$loss = \left(\underbrace{\overset{\text{Reward}}{r} + \overset{\text{Decay Rate}}{\gamma} \max_{a`} \hat{Q}(s,a`)}_{\text{Target}} - \underbrace{Q(s,a)}_{\text{Prediction}} \right)^2$$

- Use gradient descent with back-propagation to update network weights and minimize error.

Issues: approximation of Q-values using nonlinear functions (such as NNs) is not very stable and very slow.
Solution: several "tricks" can be used to speed the convergence: the most important is **experience replay**. For instance, we put the last million transitions (or video frames) into a buffer and sample a mini-batch of samples of size 32 from this buffer to train the deep network. This forms an input dataset which is stable enough for training.

Algorithm 2: Deep Q-learning with experience replay.

```
Initialize replay memory $D$ to capacity N;
Initialize action-value function $Q$ with random weights $\theta$;
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$;
for episode = 1,M do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
    for t=1,T do
        With probability $\epsilon$ select a random action $\alpha_t$;
        otherwise select $\alpha_t = \text{argmax}_a Q(\phi(s_t),\alpha;\theta)$;
        Execute action $\alpha_t$ in emulator and observe reward $r_t$ and image$x_{t+1}$;
        Set $s_{t+1} = s_t,\alpha_t,x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$;
        Store transition $(\phi_t,\alpha_t,r_t,\phi_{t+1})$ in $D$;
        Sample random minibatch of transitions $(\phi_j,\alpha_j,r_j,\phi_{j+1})$ in $D$;
            $Set\ y_j = \begin{cases} r_j, & \text{if episodes terminates at step j + 1} \\ r_j+\gamma max_{a'}\hat{Q}(\phi_{j+1},\alpha',\theta^-), & \text{otherwise} \end{cases}$   (1.1)
        Perform a gradient descent step on $(y_i - Q(\phi_j,\alpha_j,\theta))_2$ with respect to the
        network parameters $\theta$;
        Every C steps reset $\hat{Q} = Q$
    end
end
```

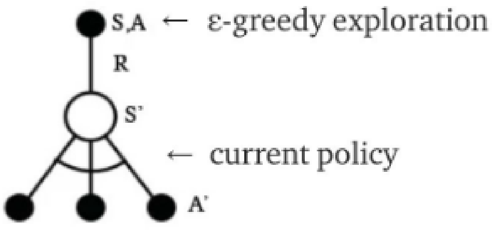


Figure 1.2: DQN uses -greedy to select the first action.

1.2 | Technologies

Figure 1.3: Used technologies

1.2.1 | OpenAI Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball (see 4.2 [2]). There are some environments already defined such as: CartPole, LunarLander, SpaceInvaders etc. A new environment can be defined by extending the main class and overriding some methods (*step(action)*, *render()*, *reset()*).

1.2.2 | Keras

Keras is an open-source neural-network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML.Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. In particular, for our project we made use of an extension called **keras-rl**. Keras-rl implements some state-of-the art deep reinforcement learning algorithms in Python and seamlessly integrates with the deep learning library Keras. Also keras-rl works with OpenAI Gym out of the box. This means that evaluating and playing around with different algorithms is easy. (see 4.2 [3])

1.2.3 | Python modules

Other python modules were used during the project:

- **Numpy:** is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.
- **Matplotlib:** is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across plat-

forms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

- **Pandas:** is a library that provides structures and tools for analyzing data in Python language. The core of the library are DataFrame objects, 2D data structures indexed on both columns and rows
- **Tkinter:** is one of the most used libraries for making python graphic interfaces. It was used in our case to provide a graphical interface to our 2048 environment.

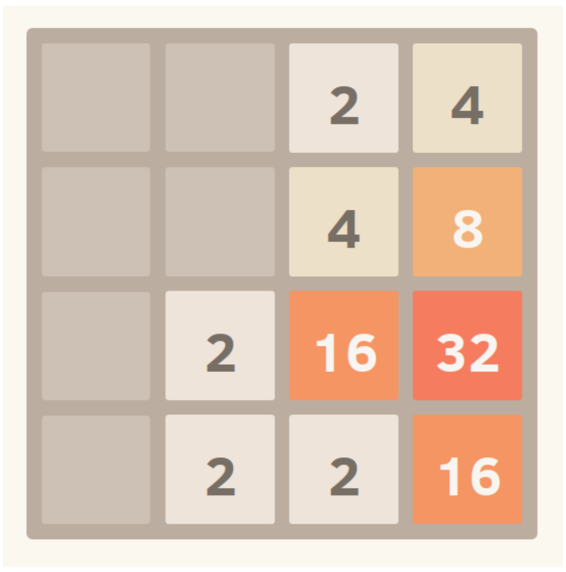
Development of the project

The game we have chosen is 2048, a very famous app that exploded a few years ago. This type of game is very difficult to make a machine understand because there are a number of heuristics that the human player can understand. In fact, getting the biggest reward in every single move is not always the best thing. We approached the problem in different ways until we reached an optimal solution using reinforcement learning. Although it is difficult for a machine to maximize the scores on these types of games, we managed to reach the victory score of 2048 in some cases in the last training phase of the network (over 2 days of training).

2.1 | 2048: How to play?

2048 is a single-player sliding block puzzle game designed by Italian web developer Gabriele Cirulli. The game's objective is to slide numbered tiles on a grid to combine them to create a tile with the number 2048. However, one can continue to play the game after reaching the goal, creating tiles with larger numbers. 2048 is played on a gray 44 grid, with numbered tiles that slide smoothly when a player moves them using the four arrow keys. Every turn, a new tile will randomly appear in an empty spot on the board with a value of either 2 or 4. Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. The resulting tile cannot merge with another tile again in the same move. The user's score starts at zero, and is increased whenever two tiles combine, by the value of the new tile. The game is won when a tile with a value of 2048 appears on the board, hence the name of the game. After reaching the 2048 tile, players can continue to play (beyond the 2048 tile) to reach higher scores. When the player has no

legal moves (there are no empty spaces and no adjacent tiles with the same value), the game ends.
Let's try to play at the original game → ***https://play2048.co/***



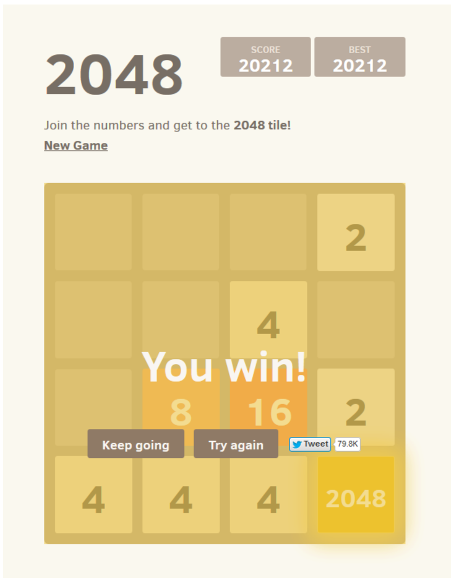


Figure 2.1: A game of 2048 in progress A completed game of 2048.

2.2 | Gym-2048 environment definition

To define the logic of the game and make it compatible, we used the openAI gym framework (see [2] 4.2). Gym doesn't provide a default environment for 2048 so we had to define one from scratch by implementing the super class and overriding some methods. After having initialized some environment fields (like the score, the matrix that represent the game, the action space, the seed etc), the main method of gym interface was implemented:

```
# Implementation of gym interface:
    def step(self, action):
        """Perform one step of the game. This involves moving and
           adding a new tile."""

        ...

         return observation, reward, done, info
```

The agent (in the case of the DQN) or the Neural Network (in the supervised cases) makes a predict on an action and tries to call this method which will make the move in the positive case or return an IllegalMove exception in the negative case. Then inside the method there are a series of function that deal with the logic of the game (increase the score, merge the cell and so on). The method returns a series of data useful to the agent (or NN) to continue his training:

- **observation:** current state of the 2048 matrix.
- **reward:** reward obtained in the step. (es If I combine 2 and 2 in one step I get reward 4).
- **done:** boolean value indicating whether the game has ended or not.
- **info:** dictionary with some informations (like max_tile).

Another important method that we have reimplemented is the *render()* that allows you to view the game during training or testing. In addition to the text display, we had fun creating a graphical interface with *Tkinter*. The other methods all define the logic of 2048.

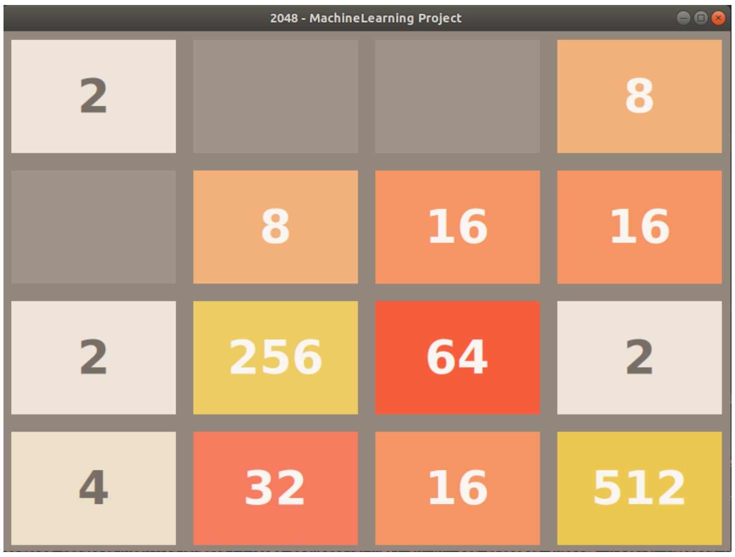


Figure 2.2: Screenshot of the interface.

2.3 | Supervised Neural Network

First approach we try to use is Neural Network. We collect data using the two approaches described below, Naive and Human, saving all game's states, matrices and related next action. Then using saved states, we train a model that learns from them. It takes an input whit dimension 16, as the matrix ones, and for all the input matrices it returns an action of the four (Up, Down, Right, Left).

```
X = np.array([i[0] for i in training_data]).reshape(-1, 16)
y = np.array([i[1] for i in training_data])

model = build_model()
model = Sequential()
model.add(Dense(16, input_dim=16, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(4, activation='linear'))
model.compile(loss='mse', optimizer="sgd")

model.fit(X, y, epochs=500)
```

Finally, we play the game and for all new state we use model's predict to find the best action for the matrix of that state.

```
action = np.argmax(trained_model.predict(previous_state.reshape
    (-1, 16))[0])
```

2.3.1 | Naive approach (Bad)

The Naive approach consist in collecting data playing the 2048 game several times using random actions and saving only data that pass a desired threshold. Then, we test the model and we saved new games returned from the test such that we can improve thanks to experience replay.

```
action = env.np_random.choice(range(4), 1).item()
next_state, reward, done, info = env.step(action)
```

Evaluation. Unfortunately, as described by the plots, reached scores aren't so high. The problem is that NN works on saved games enrolled on random choice of the action for all the matrices. So it's very unlikely that we can reach good tiles/scores collecting data and therefore the model too. Also the experience replay done by the first model it's not good to train the second model, and so on.

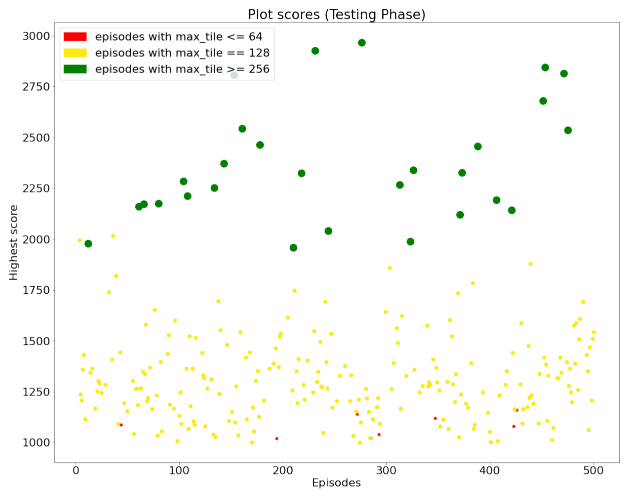


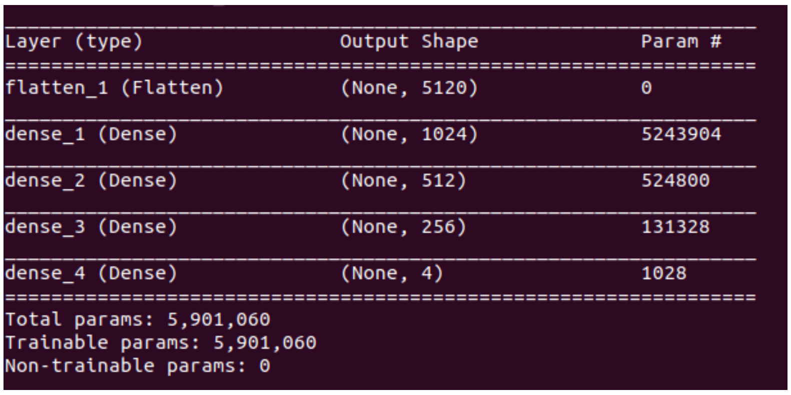
Figure 2.3: in this plot the results of the testing carried out with naive supervised NN.

2.3.2 | Human approach (Trivial)

The goal of the Human approach is to train the model using games played by human. We use the env with our graphic interface to play 2048 and save our states.
Evaluation. In this case, problems is that we have done a poor number of games (50) and even if the results for single game are good, data are not enough to train a model that returns good scores. Contrariwise, whit more data, it depends on how good is the player. In conclusion, we are not satisfied whit this trivial approach because it depends substantially on data acquired and the model has only the task to predict the best action. What we want is making possible, for the learning model, to predict the game from scratch.

2.4 | Deep Reinforcement Learning

The best solution we have found to deal with this type of problem is the use of reinforcement learning algorithms. In particular, we used Deep Q-learning with experience replay. First, an environment in *gym* has been defined. The second step was to define an agent capable of interacting with the environment (the logic of 2048 in our case). As we saw in the previous chapter, Deep Q Learning relies on a neural network to learn the Q-values (instead of the classic Q-table used by the Q-learning algorithm).



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 5120)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              5243904
_________________________________________________________________
dense_2 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_3 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_4 (Dense)              (None, 4)                 1028
=================================================================
Total params: 5,901,060
Trainable params: 5,901,060
Non-trainable params: 0
```

Figure 2.4: Definition of our Neural Network.

There will be an output neuron for each possible action (4) that will predict the Q value of that action for each step. By taking the argmax of the outputs, we can choose the action with the highest Q value, but we don't have to do that ourselves as Keras-RL will do it for us.
Once the neural network has been defined, there are a series of keras-rl hyperparameters on which to perform hypertuning: Keras-RL provides us with a class called rl.memory.SequentialMemory that provides a fast and efficient data structure that we can store the agent's experiences in:

```
memory = SequentialMemory(limit=6000, window_length=1)
```

We need to specify a maximum size for this memory object, which is a hyperparameter. As new experiences are added to this memory and it becomes full, old experiences are forgotten. The policy must be defined:

```
TRAIN_POLICY = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps',
    value_max=0.05, value_min=0.05, value_test=0.01, nb_steps=100000)
TEST_POLICY = EpsGreedyQPolicy(eps=.01)
```

Keras-RL provides a greedy Q Policy called rl.policy.EpsGreedyQPolicy that we can use to balance exploration and exploitation. With a model, memory, and policy defined, we're now ready to create a deep Q network Agent and send that agent those objects. Keras-RL provides an agent class called rl.agents.dqn.DQNAgent that we can use for this, as shown in the following code:

```
dqn = DQNAgent(model=model, nb_actions=4, test_policy=TEST_POLICY,
    policy=TRAIN_POLICY, memory=memory, processor=processor,
                nb_steps_warmup=5000, gamma=.99, target_model_update
                    =1000, train_interval=4, delta_clip=1.)
```

```
dqn.compile(Adam(lr=.00025), metrics=['mse'])
```

- *nb_steps_warmup*: Determines how long we wait before we start doing experience replay, which if you recall, is when we actually start training the network. This lets us build up enough experience to build a proper minibatch.
- *target_model_update*: The Q function is recursive and when the agent updates it's network for Q(s,a) that update also impacts the prediction it will make for Q(s', a). This can make for a very unstable network. The way most deep Q network implementations address this limitation is by using a target network, which is a copy of the deep Q network that isn't trained, but rather replaced with a fresh copy every so often. The target_model_update parameter controls how often this happens.

Keras-RL provides several Keras-like callbacks that allow for convenient model checkpointing and logging. We defined our callback class so as to save both in the train and in the test, the csv files used later for the evaluation (next chapter).

3

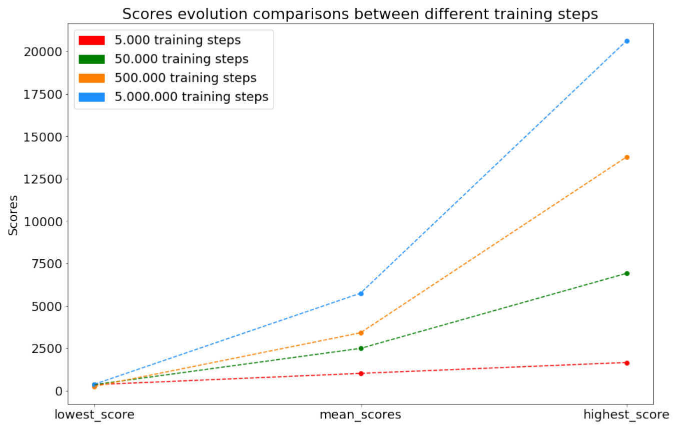
Evaluation

3.1 | Evaluation Deep Reinforcement Learning

The following pandas dataframes show the best results achieved in training (the episodes with the max tile reached). The max tile reached in each episode corresponds with the highest scores obtained in the various training phases performed. In particular the:

```
dqn.fit(env, callbacks=_callbacks, nb_steps=NB_STEPS_TRAINING,
    visualize=False, verbose=0)
```

method of the *DQNAgent* keras-rl object takes an nb_steps parameter. In each game (episode) a series of steps (moves that are made) are performed (look at the *episode_steps* column in the dataframe). When the sum of *episode_steps* == *NB_STEPS_TRAINING* the fit method ends. We trained the model with various values of *NB_STEPS_TRAINING* to see progressively improvements.



3.1.1 | Train & test 5-thousands

For our first train, we set NB_STEPS_TRAINING = 5000 and results are shown below with two plots. We reach good results but tiles are not high enough to win the game (2048). Trying higher value for NB_STEPS_TRAINING, we find that, step by step, time to evaluate increased, but fortunately scores too.

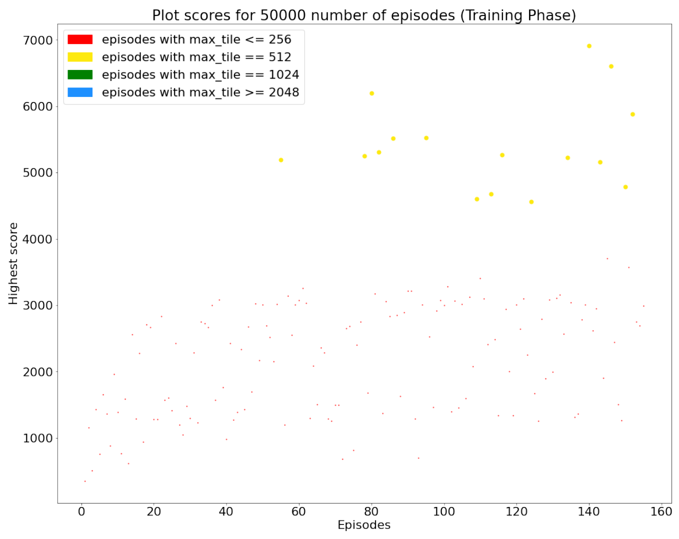


Figure 3.1: in this plot the results of the training carried out with nb_steps = 5.000 are reported.

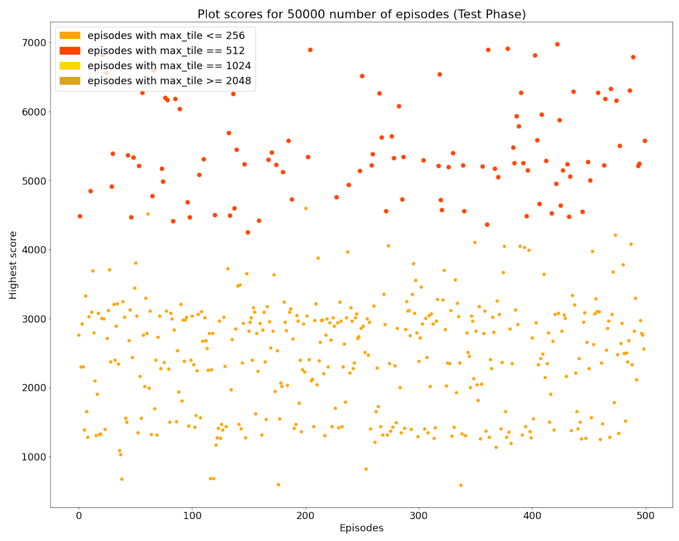


Figure 3.2: This plot shows the results of the testing carried out with the model created with 5.000 training steps. (see 3.1). The pc played 500 games on this model.

3.1.2 | Train & test 5-milions

Training requires high computational resources (just think that 5.000.000 of training_steps take 2 days). Despite this, with 5 million we see very significant results as evidenced by the dataframe and the plots below (2048 is reached several times).

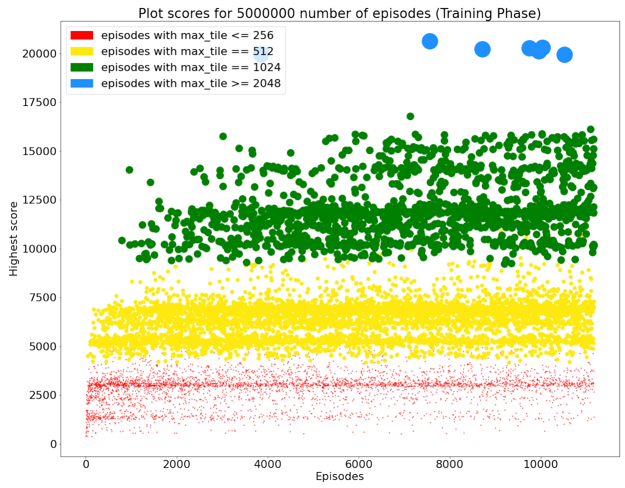


Figure 3.3: in this plot the results of the training carried out with nb_steps = 5.000.000 are reported.

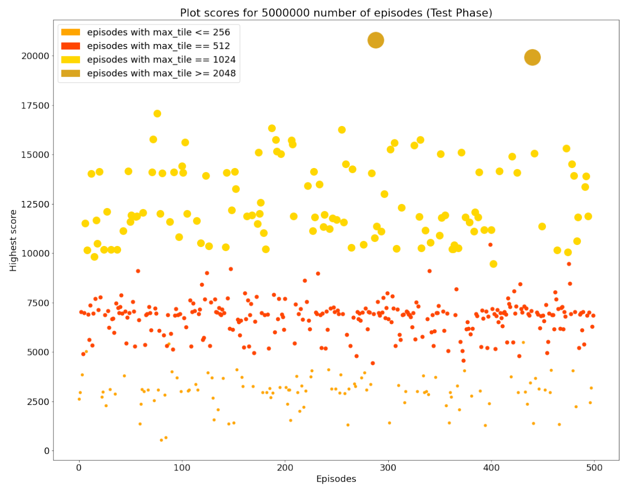
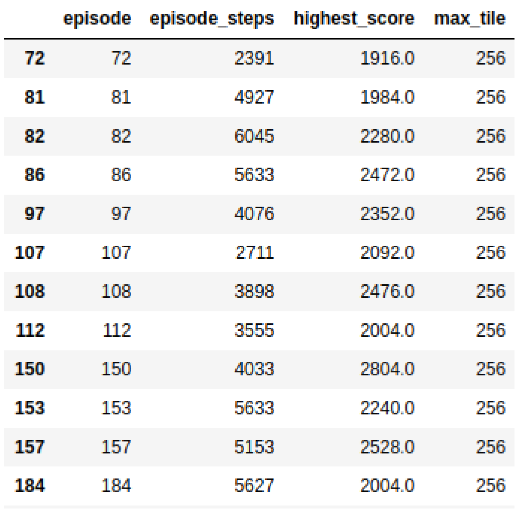


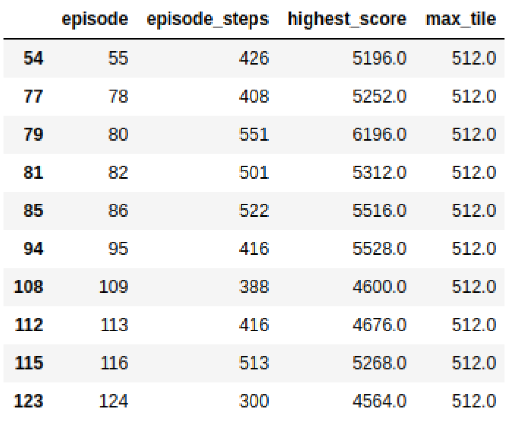
Figure 3.4: This plot shows the results of the testing carried out with the model created with 5 million training steps. (see 3.3). The pc played 500 games on this model.

3.1.3 | Train & test final recap

Complete notebook of the evaluation at this link → `https://github.com/francescobianca/2048-DeepQLearning-Approach/blob/master/evaluation.pdf`

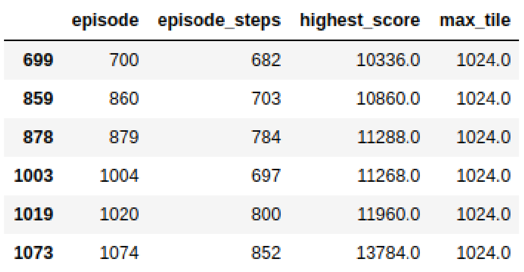


	episode	episode_steps	highest_score	max_tile
72	72	2391	1916.0	256
81	81	4927	1984.0	256
82	82	6045	2280.0	256
86	86	5633	2472.0	256
97	97	4076	2352.0	256
107	107	2711	2092.0	256
108	108	3898	2476.0	256
112	112	3555	2004.0	256
150	150	4033	2804.0	256
153	153	5633	2240.0	256
157	157	5153	2528.0	256
184	184	5627	2004.0	256

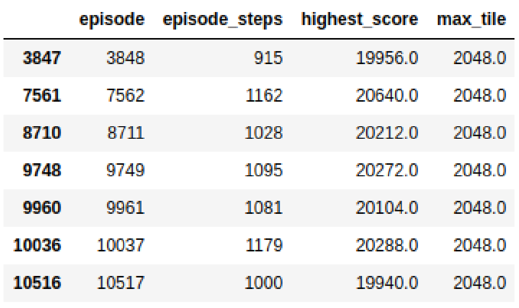


	episode	episode_steps	highest_score	max_tile
54	55	426	5196.0	512.0
77	78	408	5252.0	512.0
79	80	551	6196.0	512.0
81	82	501	5312.0	512.0
85	86	522	5516.0	512.0
94	95	416	5528.0	512.0
108	109	388	4600.0	512.0
112	113	416	4676.0	512.0
115	116	513	5268.0	512.0
123	124	300	4564.0	512.0

Figure 3.5: DataFrames with training results for nb_steps=5.000 and nb_steps=50.000



	episode	episode_steps	highest_score	max_tile
699	700	682	10336.0	1024.0
859	860	703	10860.0	1024.0
878	879	784	11288.0	1024.0
1003	1004	697	11268.0	1024.0
1019	1020	800	11960.0	1024.0
1073	1074	852	13784.0	1024.0



	episode	episode_steps	highest_score	max_tile
3847	3848	915	19956.0	2048.0
7561	7562	1162	20640.0	2048.0
8710	8711	1028	20212.0	2048.0
9748	9749	1095	20272.0	2048.0
9960	9961	1081	20104.0	2048.0
10036	10037	1179	20288.0	2048.0
10516	10517	1000	19940.0	2048.0

Figure 3.6: DataFrames with training results for nb_steps=500.000 and nb_steps=5.000.000

4

Conclusion

4.1 | Conclusion

Deep Reinforcement Learning turned out to be the best methods to understand a game, find the best way to improve results using replays and then, from zero, reach game's goal: in this case 2048! The game's logic force to find, and use, the action that allows to get the best score starting from the current state. The goal, achieved, was to create an automatic player that managed to perform better or equal to different mid-level human players. Obtained results, in particular for the last trained model, are very satisfying.
Future goals: model training requires high computational resources as the number of training steps increases. We saw that the improvement was significant from 5000 to 5.000.000 training steps, therefore it would be interesting to try also with larger numbers to try to reach 2048 steadily (computational resources permitting). Furthermore, it would be interesting not only to focus on reward but also to train "to think" like a human.

4.2 | Reference

[1] https://en.wikipedia.org/wiki/2048_(video_game)
[2] https://gym.openai.com/docs/
[3] https://keras-rl.readthedocs.io/en/latest/
[4] https://github.com/francescobianca/2048-DeepQLearning-Approach
[5] https://youtu.be/swc_otVJJrA