Ivancich Stefano 1227846

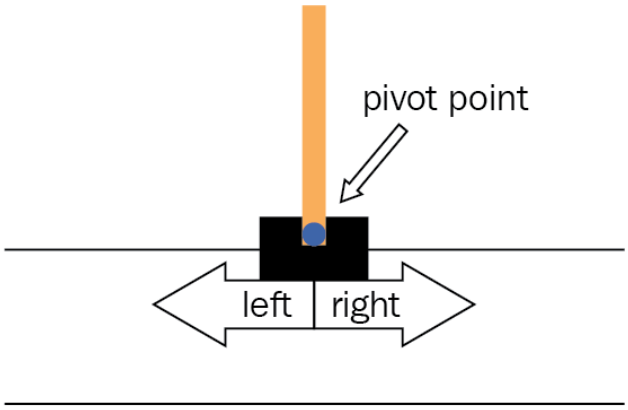
Neural Network and deep learning course 2020/21
Homework 3

---

1. Introduction

In this homework we have to implement and test neural network models for solving reinforcement learning problems.
The basic tasks require to implement some extensions to the code that we have seen in the Lab, that is trying to improve the learning convergence (learn to obtain maximal score with fewer epochs).
The second task require to train and test the learning agent using the screen pixels as state.
The final tasks consist of train and test the learning agent on a different environment, we choose MountainCar-v0.

The ***CartPole-v1*** **environment** consist of a pole that is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart's velocity. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.



State space vector:

- 0: Cart Position ∈ $[-2.4, 2.4]$
- 1: Cart Velocity ∈ $[-Inf, Inf]$
- 2: Pole Angle ∈ $[-41.8°,\ 41.8°]$
- 3: Pole Velocity At Tip ∈ $[-Inf, Inf]$

Actions space vector:

- 0: Push cart to the left
- 1: Push cart to the right

Note: The amount the velocity is reduced or increased is not fixed as it depends on the angle the pole is pointing. This is because the center of gravity of the pole increases the amount of energy needed to move the cart underneath it.

Reward: Reward is 1 for every step taken, including the termination step. The threshold is 475.
BUT since the cart start going outside the screen, we modified the reward by applying a linear penalty when the cart is far from the center.

```
pos_weight = 1
reward = reward - pos_weight * np.abs(state[0])
```

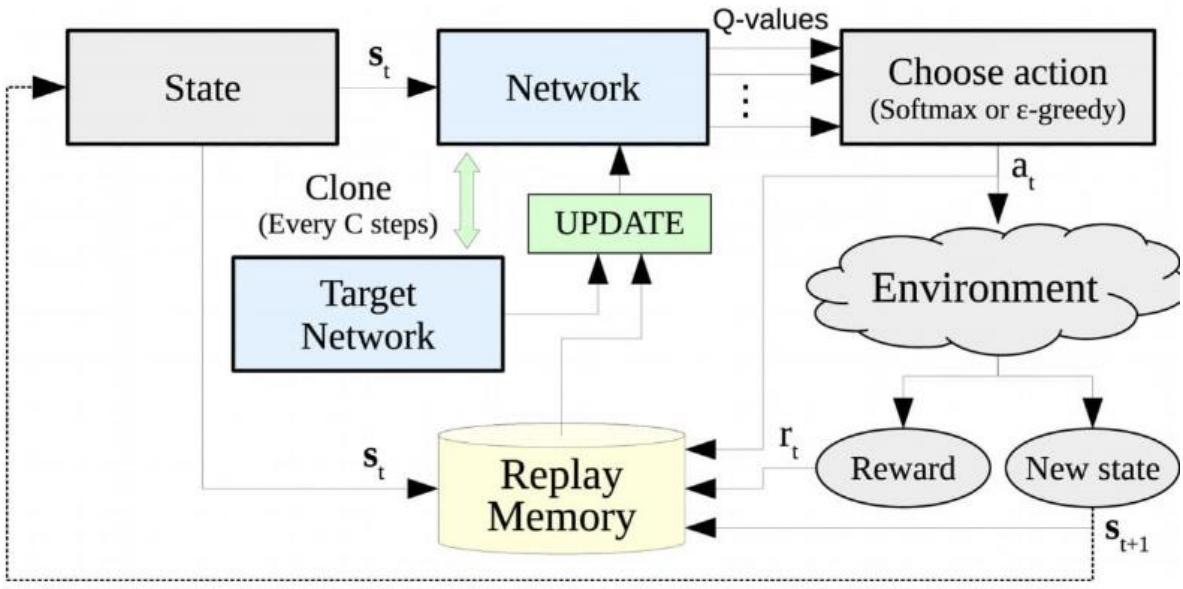
Starting State: All observations are assigned a uniform random value between ±0.05.

Episode Termination:

- Pole Angle is more than $\pm 12°$
- Cart Position is more than ± 2.4 (center of the cart reaches the edge of the display)
- Episode length is greater than 500

2. Speeding up learning Convergence

As we saw in class, since using the bellman equation is too expensive, the architecture we use to make the agent learn is composed of a policy network that takes a state as input and provides the Q-value for each of the possible actions.
Then we choose the action according to a Softmax distribution (with temperature) of the Q-values. If the temperature is 0, we just select the best action using the eps-greedy policy with epsilon = 0.
A replay memory (a queue) will store the state, action, next state and the reward of the current episode.
Update the target network every `target_net_update_steps` episodes by training on the examples that are in the reply memory and then coping the weights of the policy network to the target network.

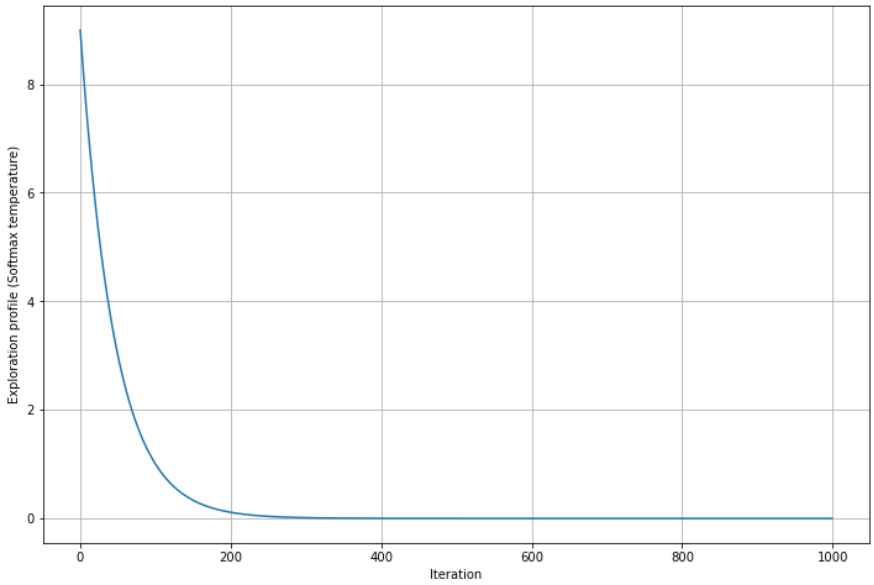


The **network** used is the following:

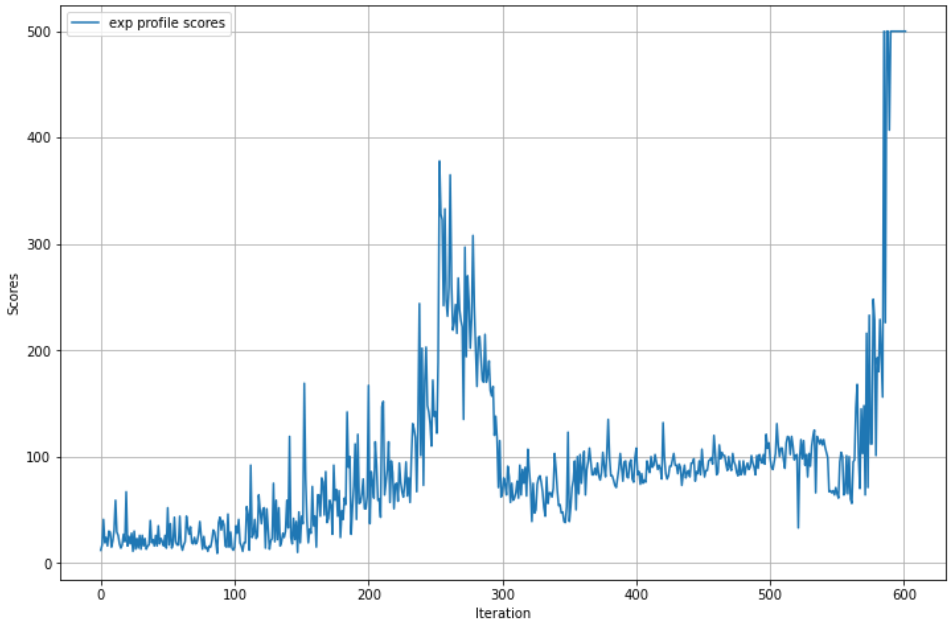
- **Input**: 4 (state space dimension: Cart position, Cart velocity, Pole angle, Pole velocity)
- **FC1 number of neurons:** 128
- **FC2 number of neurons:** 128
- **Output layer:** 2 (action space dimension: push left or push right)
- **Activations:** tanh

The **best hyperparameters** turned out to be:

- **gamma** = 0.98, parameter for the long-term reward
- **replay_memory_capacity** = 10000
- **lr** = 1e-2, Optimizer learning rate
- **target_net_update_steps** = 10 # Number of episodes to wait before updating the target network
- **batch_size** = 128, Number of samples to take from the replay memory for each update
- **bad_state_penalty** = 0 , Penalty to the reward when we are in a bad state (in this case when the pole falls down)
- **min_samples_for_training** = 1000, Minimum samples in the replay memory to enable the training
- **Exploration Profile**:
  - **initial_value** = 9
  - **exp_decay** = np.exp(-np.log(initial_value) / num_iterations * **10**)

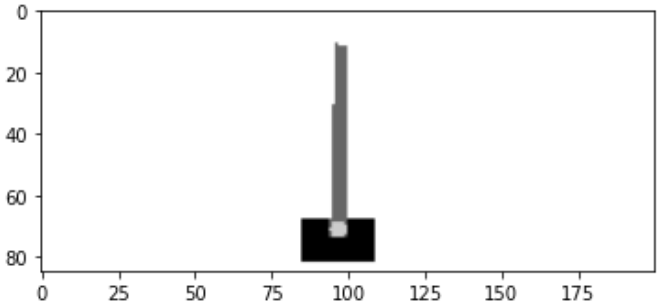


This configuration can reach the perfect score (500) around the **590th** episode, instead of 800 of the original seen in class.



3. Control CartPole using the screen pixels

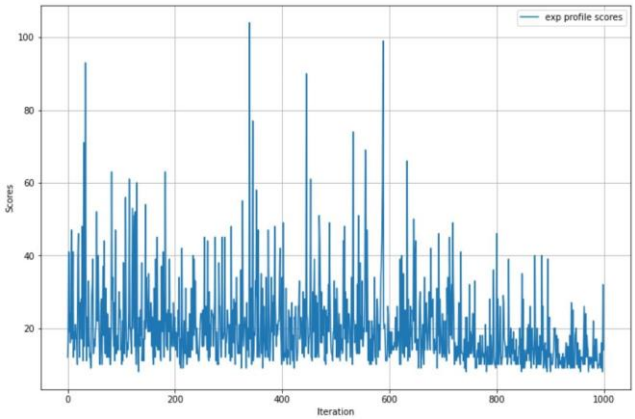
To get the state in the previous approach we used: `next_state, reward, done, info = env.step(action)` but now we will ignore the `next_state` variable and use as state the 800x600x3 tensor returned by `env.render(mode='rgb_array')`
Since this image is quite big, we just took the third RGB channel, because the colors do not matter, then we cut it because there are parts of the screen that does not matter like all pixels above or below the cart are always white, and finally we make it more compact by rescaling it 2 times, obtaining a final image size of `1x85x200`.



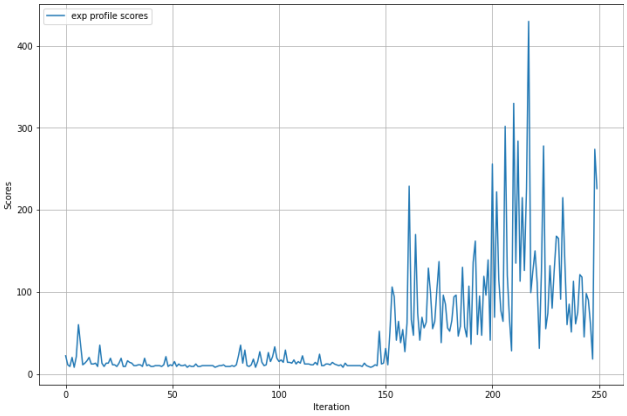
We tried 2 types of networks: a fully Connected (the same used in the previous point) and a Convolutional network.
The **convolutional network** architecture is composed as following:

- **Input dimension:** 1x85x200
- **Conv1**: 64 filters 4x4, stride 2, padding 0, ReLu
- **Conv2**: 64 filters 4x4, stride 2, padding 0, ReLu
- **Conv3**: 32 filters 3x3, stride 1, padding 0, ReLu
- **Flatten**: 25024
- **FC1 number of neurons:** 128, tanh
- **FC2 number of neurons:** 128, tanh
- **Output layer:** 2 (action space dimension: push left or push right)

Unfortunately, the learning speed for this task is very slow also with a small network, so we couldn't explore too much the hyperparameters and configurations.
For the **Fully connected** network the average **test score** is around **20**.
With the **Convolutional Network** we obtained slightly better results, but we are still not able to win the game, the average **test score** is around **102**.
The following are the training scores:



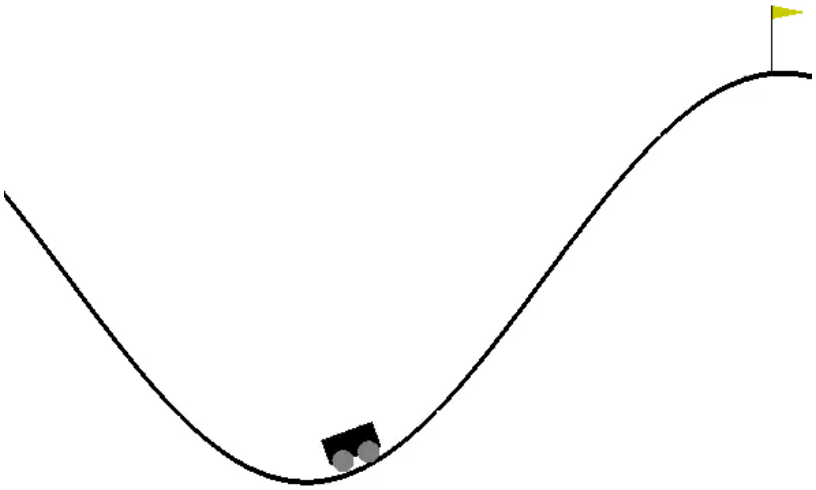
Fully connected



Convolutional

4. MountainCar-v0 Gym Environment

The ***MountainCar-v0*** **environment** consist of a car running on a hill, that can only choose to push left, push right or do nothing. The objective is to get the car to the top of the hill (top = 0.5 position).



State space vector:

- 0: position ∈ $[-1.2, 0.6]$
- 1: velocity ∈ $[-0.07, 0.07]$

Action space vector:

- 0: push left
- 1: no push
- 2: push right

Reward: -1 for each time step, until the goal position of 0.5 is reached. There is no penalty for climbing the left hill, which upon reached acts as a wall.
BUT we decided to create our own reward function based on the position and direction (velocity) of the car, that is the following:

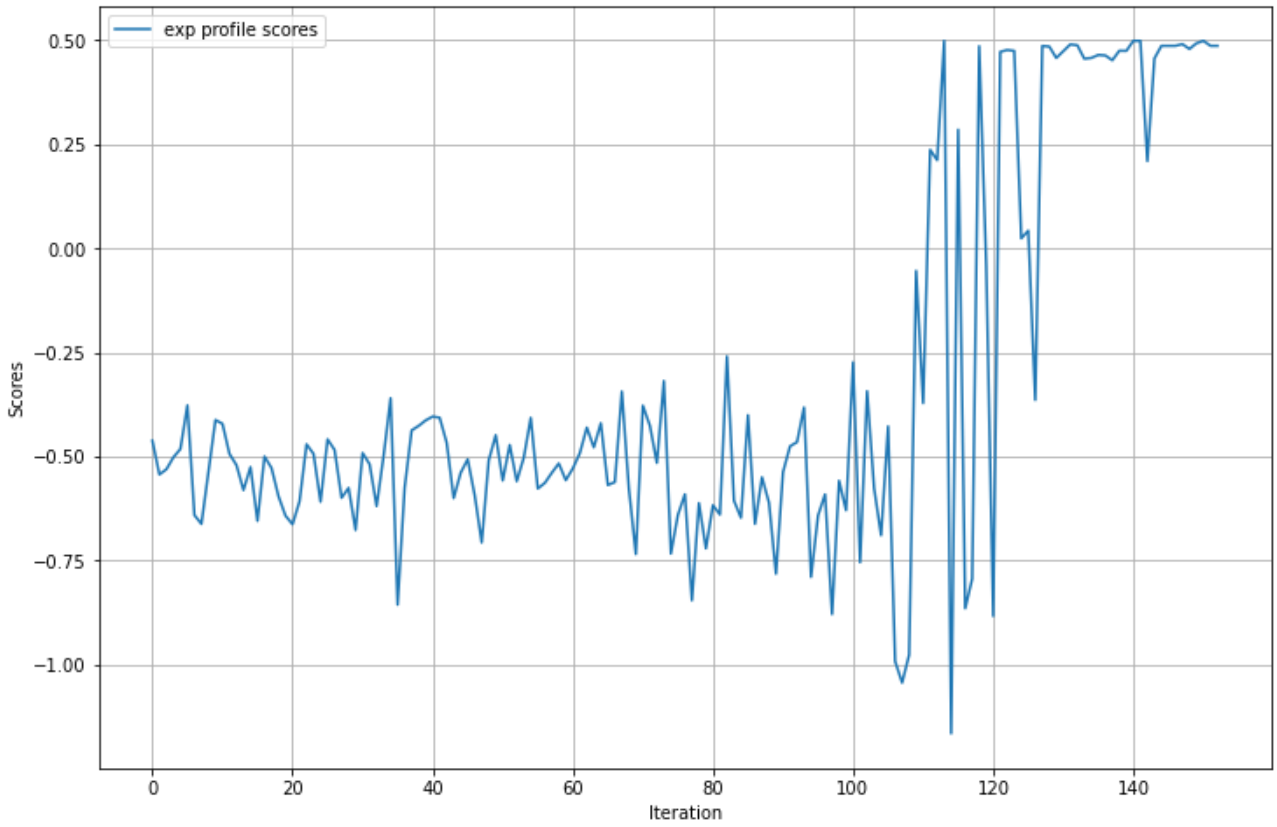
```
pos_weight = 6
if (action ==0 and state[1]<0) or (action==2 and state[1]>0):
      reward= reward + pos_weight * np.abs(state[0]+0.5)
else: reward=reward-2
```

Basically, if the Car is pushing left and the velocity is negative (direction to the left) or the car is pushing right and the velocity is positive (direction to the right) we give a positive reward. Because we want that the car stays away as much possible from the center (we don't want that the car remains stationary), otherwise we give a negative reward. (The specific values of the parameters were found after some fine tuning).

Starting State: Random position from -0.6 to -0.4 with no velocity.
Episode Termination: when you reach 0.5 position, or if 200 iterations are reached.

The network and training procedure are exactly the same used in the *CartPole-v1* environment except for the reward function, and the score that here we define as the final position of the car.

We used the same hyperparameters of the first point of the homework for the training, and the agent can reach perfect score after about **140** episodes.


exp profile scores
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Scores
0
20
40
60
80
100
120
140
Iteration