Neural Networks and Deep Learning 2021/2022

# Deep Reinforcement Learning

**Capettini, Hilario (2013031)**

September 12, 2022

## Contents

## Abstract

In this work we explored how to use neural networks to solve reinforcement learning problems under the paradigm of Deep Q-Learning. A learning agent was trained to solve the *CartPole-v1* environment using the state representations provided by the environment. Also a learning agent was implemented to solve the *LunarLander-v2* environment directly from the state representation.

**Keywords**
Reinforcement Learning, Q-learning, PyTorch, Neural Networks

## 1 Introduction

Reinforcement learning mainly consist in training an agent that interacts with an environment through different actions along the time $t$. Every action $a_t \in A$ that the agent choose in a specific environment state $s_t$ produce a change in the environments state $s_t \to s_{t+1}$ and as a result the agent receives a reward $r_t$. The agents target is to maximise the long term reward $R_t$ by picking the best actions.

$$R_t = r_t + \gamma t_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^T r_T. \qquad (1)$$

When using the Q-learning method [1], the agent learns to associate the expected cumulative reward to each possible state-action pair $Q^\pi(s_t, a_t) = E(R_t|s_t, a_t, \pi)$ under a exploration policy $\pi$ (A policy is no more than a probability distribution over the set of actions $A$). Following this line, the optimal Q-value function is defined as the maximum return achievable after seeing a sequence of states $s_t$ followed by actions $a_t$:

$$Q^*(s_t, a_t) = \max_\pi E(R_t|s_t, a_t, \pi). \qquad (2)$$

This action-value function obeys the Bellman equation:

$$Q^*(s_t, a_t) = E_s'(r_t + \gamma \max{}_a' Q^*(s', a')|s_t, a_t). \qquad (3)$$

At the end what the agent does is to learn $Q^*$ by approximating it with a Neural Network.

$$Q(s, a, \theta) \approx Q^\pi(s, a), \qquad (4)$$

where $\theta$ are the weights of the network. This is done by minimising the objective function:

$$L(\theta) = E[((r + \gamma \max{}_a' Q^*(s', a', \theta)) - Q(s, a, \theta))^2], \qquad (5)$$

which can be done using the gradient descent algorithms.

### 1.1 Gym Environments

In order to train our agent we need an environment, for this work we use two environments provided by the *OpenAI* Gym [2]; a well known toolkit for developing and comparing reinforcement learning algorithms. The choosen environments for this work were **Cartpole-v1** where the goal is to balance a pole which is attached to a cart by moving the cart to the right or to the left. And **Lunarlander-v2** where the goal is to land a lunar vehicle in a landing pad by firing different engines.
**CartPole:**

## 2 Method

All the implementation of the code for this notebook was done using PyTorch [3].

As already mentioned the actions are sampled from a policy $\pi$, in general there are two well known policies [4]:

- **$\epsilon$-greedy policy**: is a policy where optimal actions are chosen with probability $\epsilon - 1$ while non optimal actions with probability $\epsilon$. The $\epsilon$ value can be modified along time to allow exploration at first and exploitation in a later stage.
- **Softmax policy** the action is chosen according to a softmax distribution of the Q-values, at a specific temperature $T$. This parameter can start with a high value and decrease along time allowing for exploration of states during the first stages.

For the training of the agent we used a **replay memory** which helps the algorithm to train with less correlated data. And the other technique that we used to have a more stable training and to break the correlation between the target function and the Q-network was the implementation of a **Target Network**.

## 2.1 Cartpole

In this environment the goal is to balance a pole which is attached to a cart by moving the cart to the right or to the left (see Figure 1). The state representation is

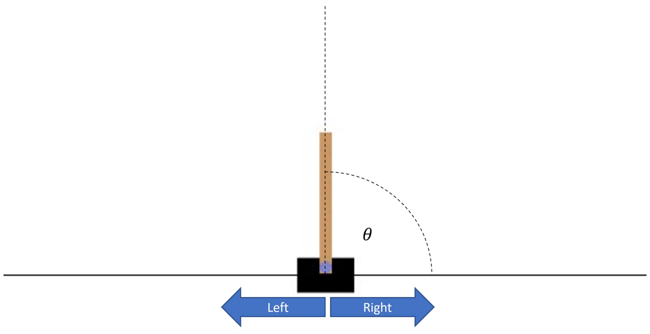

Figure 1: CartPole representation.

4-dimensional consisting of:

- Cart x-position
- Cart x-velocity
- Pole angle
- Pole angular velocity.

The agent´s reward is +1 point for each time iteration the pole still balances. An episode ends as soon as the pole falls beyond a certain angle or the cart goes too far from centre of the screen.

For the policy and target networks we use the simple Linear Fully Connected Network presented in the laboratory:

- **Input Layer:** (input_size = 4, output_size=128)
- **Hidden Layer:** (input_size = 128, output_size=128)
- **Output Layer:** (input_size = 128, output_size=2)

The chosen activation function for this network was the hyperbolic tangent from PyTorch `nn.Tanh()`.

In particular we used this environment to investigate the performance of two policies, the impact of the exploration profile on the learning curve and also the tuning of the hyperparameters.

For each policy here investigated, $\epsilon$-*greedy* and *softmax*, we propose three different exploration profiles. For the first policy in Figure 2 can be observed the $\epsilon$ value as a function of the iteration. For the second policy Figure 3 represent the temperature that is used by the *softmax policy* at each iteration to pick an action.

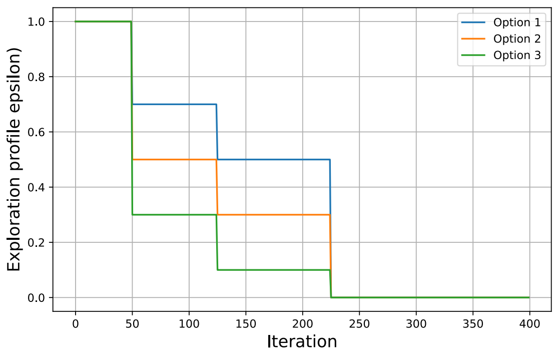

Figure 2: ****

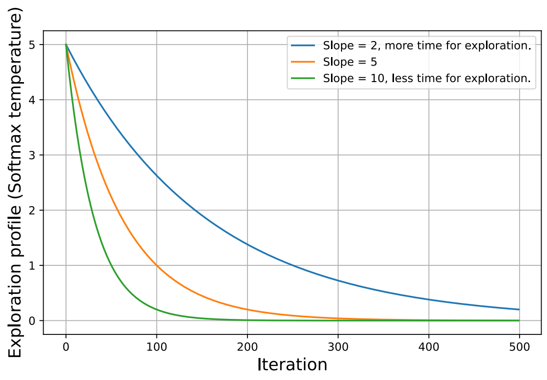

Figure 3: ****

Lastly various models were tested using different hyperparameters (see table 1), for this purpose we used Optuna.

## 2.2 Lunar Lander

For the Lunar Lander environment the state representation is 8-dimensional consisting of:

- Lander x-position

| Hyperparameter | Interval |
|---|---|
| Optimizer | [SGD,ADAM] |
| $lr$ | $[1e-3, 1e-1]$ |
| Gamma | $[0.9, 0.99]$ |
| Initial temperature | $[1, 10]$ |

Table 1: Search space for Optuna

- Lander y-position
- Lander x-velocity
- Lander y-velocity
- Pole angle
- Pole angular velocity.
- Left leg is grounded
- Right leg is grounded

The agent has four possible responses **do nothing**, **fire left**orientation engine, **fire main** engine, **fire right** orientation engine. The reward for moving from the top of the screen to the landing pad and coming to rest is about [100140] points. If the lander moves away from the landing pad, it loses reward. If the lander crashes, it receives an additional $-100$ points. If it comes to rest, it receives an additional $+100$ points. Each leg with ground contact is $+10$ points. Firing the main engine is $-0.3$ points each frame. Firing the side engine is $-0.03$ points each frame. Solved is 200 points.

For the policy and target networks we use a simple Linear Fully Connected Network:

- **Input Layer:** (input_size= 8, output_size= 100)
- **First Hidden Layer:** (input_size= 100, output_size= 200)
- **Second Hidden Layer:** (input_size= 200, output_size= 100)
- **Output Layer:** (input_size= 100, output_size= 4)

The chosen activation function for this network was the hyperbolic tangent from PyTorch `nn.Tanh()`.

In particular we used this environment to investigate the effects of tweaking the reward function to obtain a faster convergence of the learning curve. Our proposal is to penalise states with large angles in order to avoid spins, to penalise the distance $x$ to the centre in order to avoid going away from the pad and to penalise the $y$ distance in order to make it land faster.

## 3 Results

### 3.1 Cartpole

In figures 4,5,6 can be observed the effects of the different exploration profiles in the learning curves. In general we see that a smaller slope parameter produce a more stable learning curve however it obtain worst results in the long term. This is something expected given that a smaller slope parameter implies less exploration (taking less risk) and therefore the agent can not dicover more adequate strategies.

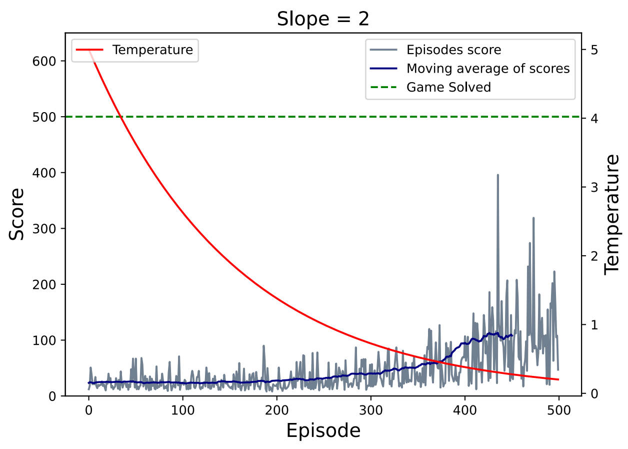

Figure 4

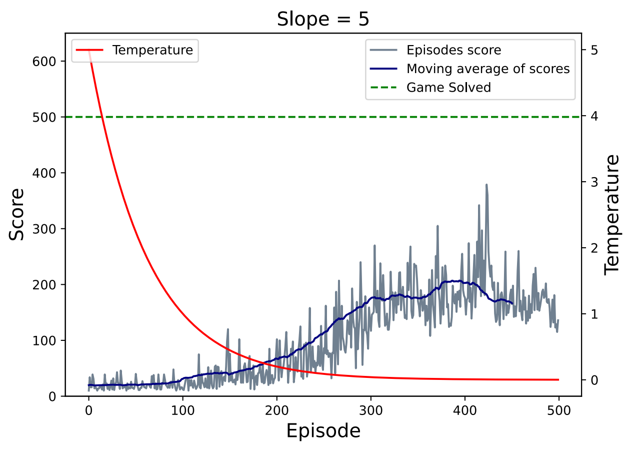

Figure 5

In figure 11 that when using the $\epsilon$-greedy policy the results are similar independently on the $\epsilon$ profile. We were expecting to obtain similar results as with the soft-max policy, however here the results are poorer.

Finally after going through 20 trials the best hyperparameters according to Optuna were:

- Optimiser SGD
- Learning rate = 0.0215
- Gamma = 0.98.

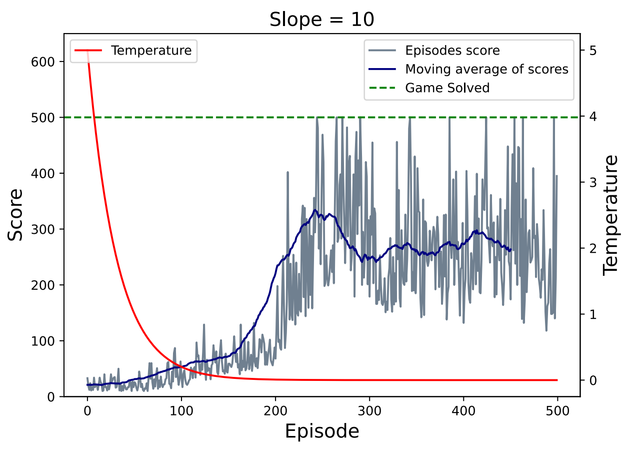

Figure 6

- Initial Temperature = 5.

The most performing model training can be seen in figure 7 where it can be seen that the game is solved after 700 episodes. Apart from that we see that the learning curve is constantly increasing which indicates a good learning process where the amount of exploration and exploitation is the right one.

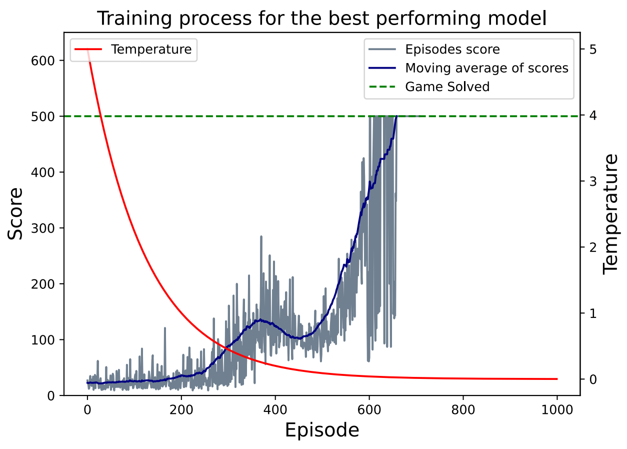

Figure 7: Best performing model.

## 3.2 Lunar Lander

In figure 9 it can be observed the model results using the original reward function while in figure 10 we can see the results of using the tweaked reward function. We can observe that the scores suffer sudden changes from one episode to the other. In figure ?? we can observe the approaching to a successful landing using the tweaked reward function. What I have observed by watching the simulations is that several times the agent moved quite slowly through the environment achieving long survival times while loosing a lot of points. The tweaked reward function was an attempt to produce faster landings, and we see that the survival times are shorter in effect, however it also produces crashes more often. Obtaining good results for this environment was challenging, due to the computational demand I did not manage to perform much experiments to tune the hyperparameters on the tweaked reward function weights.

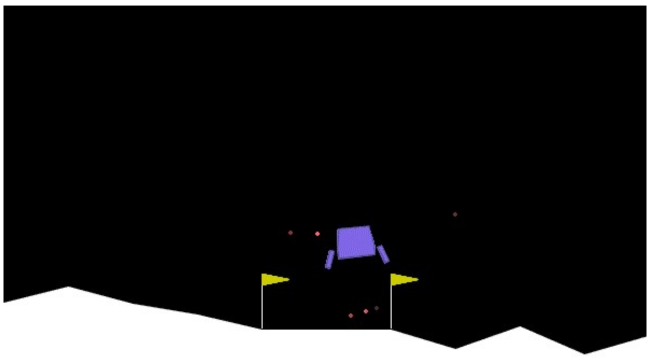

Figure 8: A few seconds before a successful landing occurred.

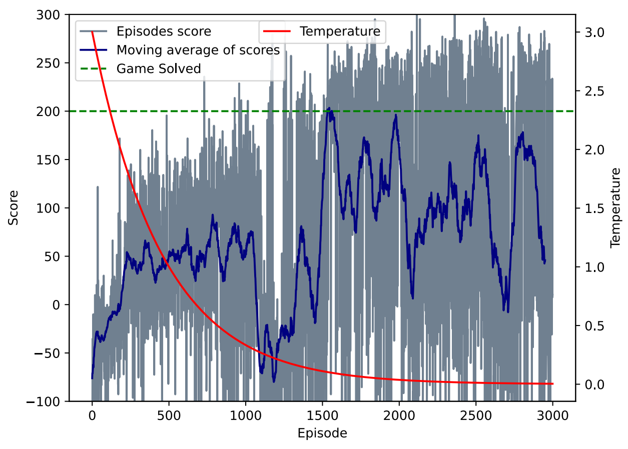

Figure 9: Agent trained using the default reward function.

## 4 Conclusion

To conclude, in this work we managed to implement a deep reinforcement learning technique called Q-learning to solve two different environments. In the process we managed to show the impact of the temperature profile while using the softmax policy allowing the agent to explore more in the initial times and to exploit it in later episodes. We also compared the performance of the $\epsilon$-greedy policy with the soft-max policy and showed the importance of the hyperparameters tuning for the models.

**Code**
All the code used for this notebook can be found at https://github.com/hcapettini2/NNDL.

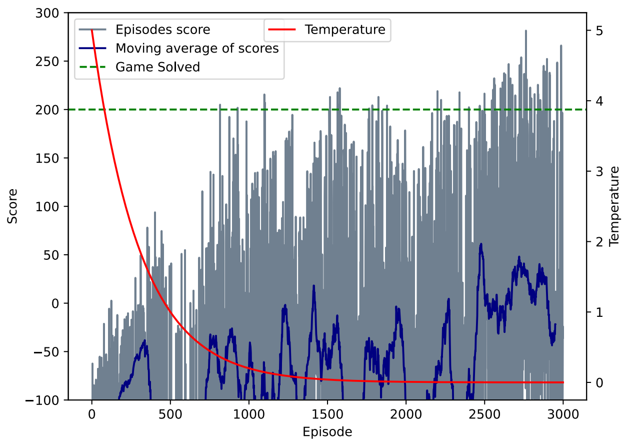

Figure 10: Agent trained using the tweaked reward function.

## References

[1] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3):279–292, 1992.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. arXiv preprint arXiv:1606.01540, 2016.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[4] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction. MIT press, 2018.

## A Appendix

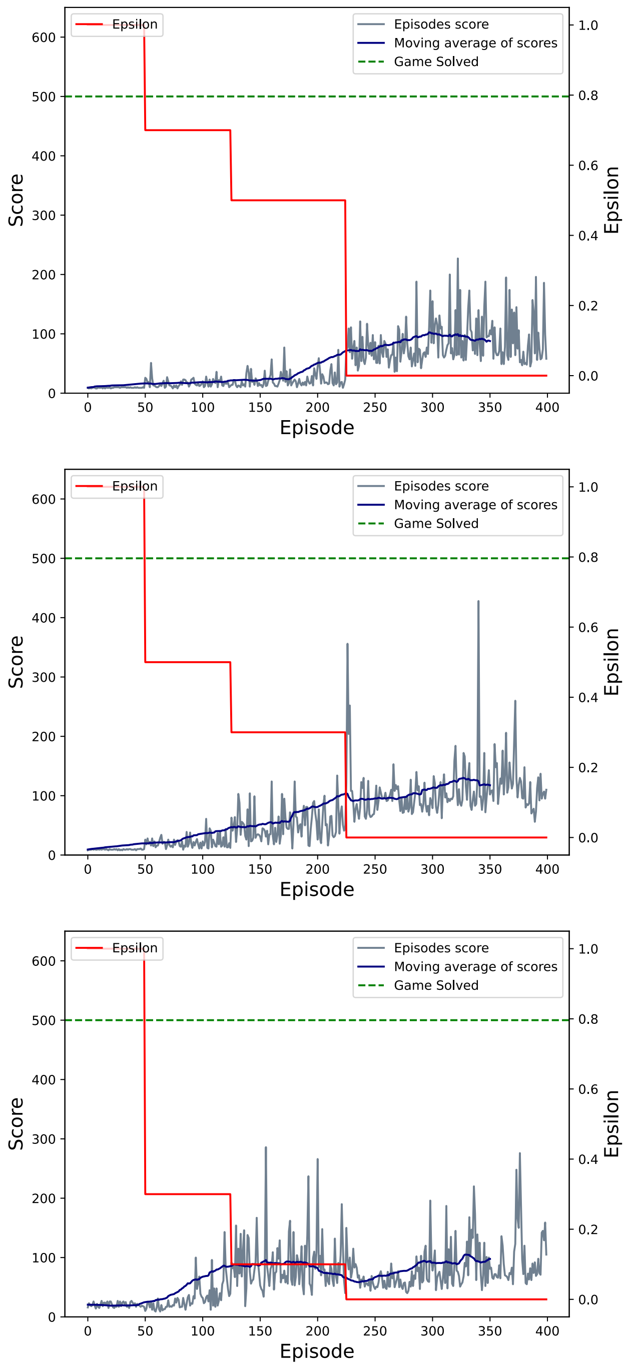

Figure 11