Neural Network and Deep Learning - A.A. 2021/22

Assignment 3

Deep Reinforcement Learning

Fontana Francesco 2026924

July 9, 2022

Abstract

In Reinforcement Learning (RL), an agent learns which actions to take within an environment to maximize a given reward. A common RL algorithm is Q-learning, based on finding a mapping between pairs of states and actions and their expected cumulative rewards. In this homework, we investigate the ability of this model to learn by maximizing its cumulative reward in two different Gym environments: the CartPole and the LunarLander

1 Introduction

In Reinforcement Learning the aim is to make an agent able to choose among some possible actions in order to maximize its cumulative reward. The main ingredients for this task are: an *agent* $\mathcal{A}$ (acting within an environment $\mathcal{E}$), a set of states $S = \{S_1, ..., s_n\}$, and a set $A = \{a_1, ..., a_n\}$ of actions per state.
At each timestep t, $\mathcal{A}$ is in a state s_t, performs an action a_t, receives a reward r_t and moves to a new state s_{t+1}.
The goal of $\mathcal{A}$ is to choose the correct action in order to maximize the cumulative reward, given by:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{+\infty} \gamma^k r_{t+k} \tag{1}$$

where $\gamma \in [0, 1]$ is the discount factor and can be tuned in order to give more or less importance to future rewards. In particular a low γ means that the agent is interested in short-term gains, while a $\gamma \approx 1$ means that long-term rewards are important as well.

1.1 Q-Learnig

In Q-learning, the agent learns the *value* of each state-action pair $Q_\pi(s, a)$, i.e. the expected cumulative return resulting from choosing a_t at state s_t, and then sampling all future moves a' in each future state s' from a fixed policy $\pi(s')$:

$$Q_\pi(s, a) = E[G_t | s_t = s, a_t = a, \pi] = E_{s'}\big[r + E_{a' \sim \pi(s')}[Q_\pi(s', a')] | s, a, \pi\big] \tag{2}$$

where the expected value is over the distribution of *next states* s' reached by choosing a in s.

The cumulative reward is maximized by the Bellman equation:

$$Q^*(s, a) = E_{s'}\big[r + \gamma \max_{a'} Q^*(s', a') \,|\, s, a\big]. \tag{3}$$

and this can then be used to learn Q^* with a neural network.

We're going to apply Deep Q-learning methods to two different environments developed by OpenAI in the Gym package:

1. **CartPole**: the goal is to vertically balance a pole attached to a moving cart. This is done in sec. 2 by using as state the phase-space coordinates of the cart and the pole.

2. **LunarLander**: the Atari game from 1969 where the goal is to control the horizontal motion of a spacecraft and to slow down its descent in order to have a safe landing into the moon (sec.3).

2 CartPole

It consists in a pole P linked by a joint to a cart C where the latter is allowed to move horizontally in a frictionless track in a $2D$ space. The aim is to let the pole remains upright as long as possible in a balanced condition (see fig.1). We're allowed to move the cart left or right by applying a discrete force at every frame of the simulation. If the pole overtakes 15 degrees from the vertical or if the cart moves more than 2.4 units from the center. Given that we can consider the state as a 4-tuple $(x_C, \dot{x}_C, \theta_P, \dot{\theta}_P)$ of phase-space coordinates.

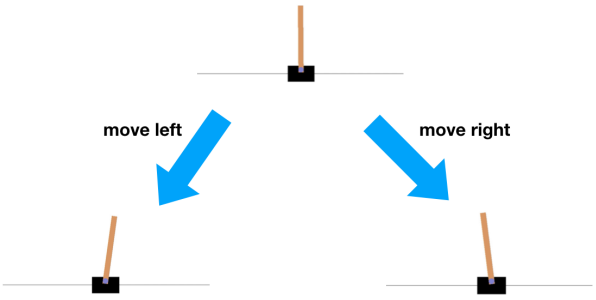

Figure 1: CartPole enviroment and the two possible actions.

The activation function and the optimizer considered are respectively the hyperbolic tangent and Adam optimizer.

2.1 Methods

The network architecture we used, consists of an input layer of 4 neurons, two hidden layer with 128 neurons each followed by an $Tanh$ activation function and an output layer of 2 neurons. Finally, for the network we use the Adam optimizer. Moreover, in order to avoid instability, we'll use two copies of the network: the first (online network) is updated every step, whereas the second (target networ) is updated only after a certain number of steps and it's used for action selection. Thus we want to minimize:

$$\mathcal{L}\left[{}^{online}_{net}(s_t, a_t); r_t + \gamma \max_{a'} {}^{target}_{net}(s_{t+1}, a')\right] \tag{4}$$

where $\mathcal{L}$ is the *loss* function, chosen to be the `SmoothL1Loss` from PyTorch, which behaves like a L1 loss for errors over 1, and a quadratic (L2) loss for smaller errors, while being differentiable everywhere.
Some more considerations are done to facilitate learning:

- **Replay memory**. In order to have an higher generalization, samples are first collected in a buffer *(replay memory)*, from which random batches are extracted to train the network. In this way, samples shown one after the other may belong to very far timesteps (or entirely different simulations), and are thus less correlated, increasing the generalization of the model.

- **Exploration profile**. Since training samples are representative of the whole state/action space, the agent is propelled to initially explore the environment, by taking random actions. Thus we considered two exploration policies, respectively:

  - **Epsilon-greedy** policy, in which the agent picks a random non-optimal action with probability ϵ or the optimal one with probability $1 - \epsilon$;
  - **Softmax**, in which the agent selects the action from a softmax distribution (tuned with a temperature T).
    These variables T and ϵ are initially set to high values, and decrease exponentially during training, thus allowing the agent some time to explore the environment before settling in a learned optimal behavior.

- **Early stopping**. If the score reaches the maximum value (500) for 10 consecutive epochs,training is stopped.

- **Gradient clipping**. Gradients during training are clipped to a maximum norm of 2, to improve the stability.

- **Adaptive learning rate**. The learning rate is manually reduced when the score rises, improving convergence.

2.2 Results

For our task, the agent follows a softmax policy:

$$T(n) = T_0 \exp\Big(- n\frac{\beta \log T_0}{N}\Big)$$

where $n = \{1, ..., N\}$ (N is the number of iterations), initial temperature $T_0 = 5$ and $\beta = 6$. We perform an initial training with the following hyperparameters:

$\gamma = 0.97$; Learning rate $= 0.01$; Batch size $= 128$; optimizer: SGD; loss: `SmoothL1Loss`

without any raward update and in fig. 2 there is a plot showing the rewards per episode.

Knowing that small tweaks to hyperparameters highly affect the speed and quality of convergence in this kind of RL models, a small scale search has been manually done, in order to speedup the convergence.

At the end, best results was achieved giving a bit more weight to future rewards (increasing γ) and using both a different optimizer with a different initial learning rate. In fact, the final network was trained with Adam optimizer with an adaptive learning rate starting from $\lambda_0 = 0.001$ and decresing to a $\lambda_0/2$, $\lambda_0/5$, and $\lambda_0/20$ when the score reached respectively 100, 200, and 400.
Moreover a small reward of $k|x_C|$ is added to penalize the agent from moving the cart too far from the screen's center, where k is a sort of positional penalization weight (used $k = 2$). Other hyperparameters are:

$\gamma = 0.98$; Batchsize= 256; optimizer: Adam, loss: `SmoothL1Loss`

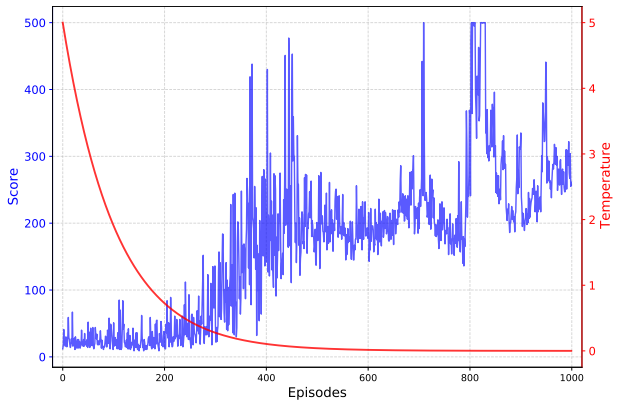

Figure 2: Score results for the initial training.

Furthermore, we also increase the number of step in which we update the target network (see above sec.2.1) from initial 10 to 20 The final learning curve is shown in fig.3 and we can see how the agent completely solves the environment after just $\sim$ 150 episodes, reaching the maximum score. Also, a brief video rendering (available at this link shows an example of how this model is able to solve the environment/game.

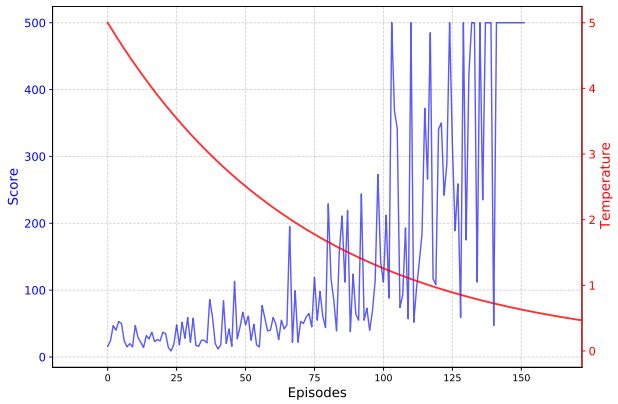

Figure 3: Score results with the best hyperparameters choice.

3 LunarLander

In this part, we play with the Atari game "LunarLander-v2" in its discrete version. The game, as mentioned above, consists in assure a secure and safe landing of a spacecraft into the moon'surface, through the action of its thruster. Unlike the previous task, this time the spacecraft could move horizontally but also vertically in the down direction or do nothing (see fig.4). From the online documentation we can find the following:

Landing pad is always at coordinates (0,0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

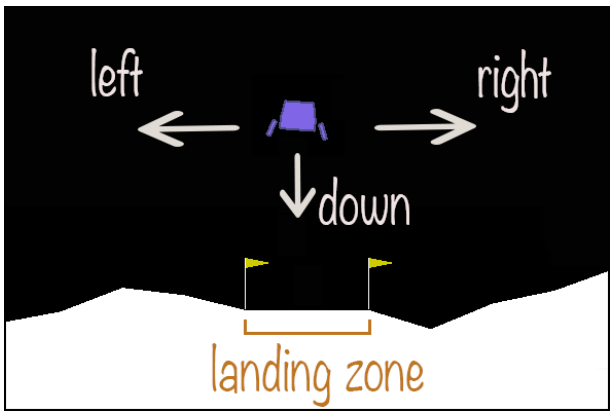

Figure 4: Example of LunarLander game with illustrated the possible actions

4 Methods

The network implemented in this task is similar to the one in the section above with the following differences:

- It has 1 more layer (in total 3 hidden layer) composed by 256 neurons;
- the activation function for each layer is the $ReLU$ function

giving the following network' structure:

Input (8n) $\longrightarrow$ F.C. layer (128n) $\longrightarrow$ F.C. layer(256n) $\longrightarrow$ F.C. layer (128n) $\longrightarrow$ Output (4n)

The input layer is composed of 8 neurons due to the fact that it must correspond to the number of state variables associated with the state space, that are:

$$states \longrightarrow \begin{cases} x \text{ coordinate of the lander} \\ y \text{ coordinate of the lander} \\ v_x \text{ horizontal velocity} \\ v_y \text{ vertical velocity} \\ \theta \text{ orientation in space} \\ v_\theta \text{ the angular velocity} \\ \text{Left leg touching the ground} \\ \text{Right leg touching the ground} \end{cases}$$

The x coordinate of the lander is 0 when the lander is on the line connecting the center of the landing pad to the top of the screen. Therefore, it is positive on the right side of the screen and negative on the left. The y coordinate is positive at the level above the landing pad and is negative at the level below. The output of the network instead have the same dimension of the *actions space* that in this case consists of four discrete actions:

1: do nothing **2**: fire left orientation engine
3: fire right orientation engine **4**: fire main orientation engine

Note that the left and right movements introduces a torque on the lander, which causes it to rotate, and makes stabilizing difficult.

As in the previous parts we make use of the same method to facilitate the learning such as replay memory, *softmax* exploration profile, early stopping (i.e. when the average of the last 30 episodes reach a value > 200), gradient clipping and adaptive learning rate.

4.1 Results

Due to the computationally expensive task for such models only a manual search of hyperparameters has been done, and as cited above, the environment is considered solved when we achieve a score > 200. Searching the best combination of hyperparameters, we used both a fixed and adaptive learning rate .In particular the latter has been changes in 4 steps when the mean value of the last 30 episodes reaches a prefixed threshold (tab.1):

Learning rate	Threshold
$\lambda_0 = 0.1$	-
$\lambda_0/10$	≥ -20
$\lambda_0/50$	$\geq +70$
$\lambda_0/100$	$\geq +120$
$\lambda_0/150$	$\geq +170$

Table 1: Adaptive learning rate implemented

Also we applied a reward penalty if the block have an inclination respect to the horizontal line ($\theta \neq 0$) and also when the lander is far away from the origin (i.e. $(x, y) \neq (0, 0)$) (a search of their combination has been done).
In this task we obtained that the best score (fig.5 *Right*) has been achieved with a fixed learning rate $\lambda = 0.001$, $\gamma = 0.99$, batchsize= 256, Adam optimizer, *SmoothL1Loss* and with a simple reward penalty due to lander's orientation. It solves the environment in just ~ 450 episodes.
Nevertheless in fig.5(*Left*) is plotted another learning curve (900 episodes) of a model that not completely solves the game despite of much more episodes. Just for completeness, in the latter the adaptive learnig rate technique and a penalty reward related to the position has been implemented; the other hyperparameters are

$\gamma = 0.98$; Batchsize= 128; optimizer: Adam, loss: `SmoothL1Loss`

This was done with a fixed learning rate and without *reward updating*; the corresponding the learning curve is reported in fig.5(left).

An example of the solved environment with the best parameters is available at this link.

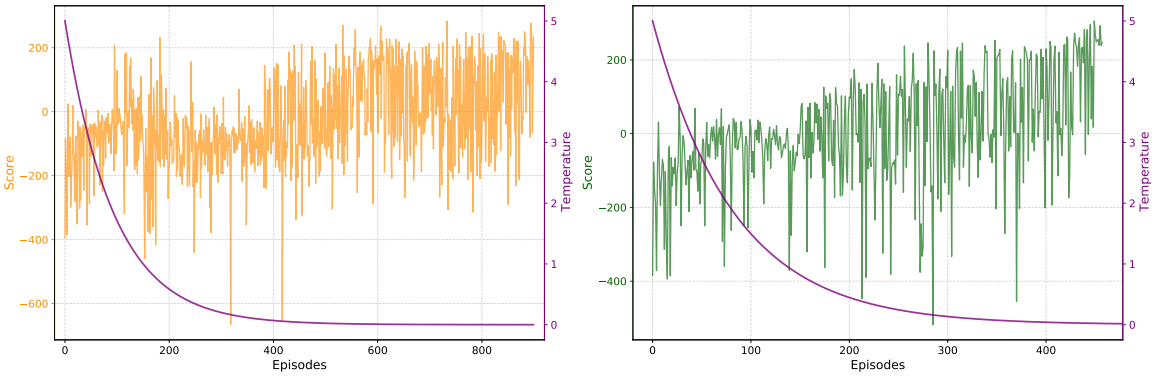

Figure 5: **Left:** Learning curve associated the non optimal hyperparameters choice. **Right:** Softmax policy used overimposed with learning curve of the training of the final network. Environment solved in just 458 episodes