DEEP REINFORCEMENT LEARNING

Matteo Bortoletto, 1142935

June 26, 2021

1 Introduction

1.1 Reinforcement learning

Reinforcement learning is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize a cumulative reward. It differs from supervised learning in not needing labelled input/output pairs be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).
The environment is typically stated in the form of a Markov decision process, in which we have a set of environment states $S = \{s_1, \dots, s_n\}$, a set of actions that can be performed $A = \{a_1, \dots, a_n\}$, a set of environment observations $O = \{o_1, \dots, o_n\}$, a set of transition rules $T(s_{t+1}|s_t, a_t)$ and a set of rules that determine the immediate reward or punishment associated with a transition $R(r_{t+1}|s_t, a_t)$. At time t, the agent receives a new observation o_t which consists in the current state s_t and a reward value r_t. Then, following a policy $\pi(a|s)$ it chooses an activation a_t from the set of available actions, which causes the transition to the next state s_{t+1}, associated to a reward r_{t+1}. The weighted sum of the accumulated rewards is called the *return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots, \tag{1}$$

where $\gamma \in [0, 1]$ is called *discount rate* and specifies how much we care about future rewards.

1.2 Q-learning

We will use Q-learning, in which the agent learns how to associate a value Q (long-term reward) to each state-action pair:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s, a]. \tag{2}$$

The initial Q-values are improved over time using the Bellman equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)). \tag{3}$$

To better explore the environment, the action is chosen probabilistically. Two examples of policy are:

- ε-*greedy policy*, in which we choose a non-optimal action with probability ε and the optimal one (i.e. the one with highest Q-value) with probability $1 - \varepsilon$. Since exploration is crucial in the initial phase, usually the initial value of ε is high and then it decays;
- *softmax policy*, in which we choose an action according to a softmax distribution (with temperature) of the Q-values.

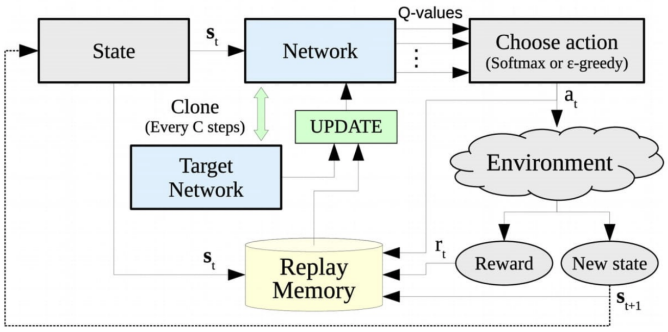


Figure 1: Deep Q-learning system architecture.

1.3 Deep Q-learning

In deep reinforcement learning, rather than maintain a table with all the estimated Q-values, we use a deep neural network to approximate their values. The objective function, which will be minimized through gradient descent algorithms, is

$$L(\boldsymbol{\theta}) = \mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q(s', a', \boldsymbol{\theta}^-) - Q(s, a, \boldsymbol{\theta})\right)^2\right]. \tag{4}$$

It is known that the main problem of using neural networks as Q-values estimators is unstability. To solve this problem we use two workarounds:

- we use two networks: a *prediction network* that estimates $Q(s, a, \boldsymbol{\theta})$ and that is trained at each iteration, and a *target network*, which is used for action selection and is updated every C steps.
- we use *experience replay*, storing in a memory buffer previous learning episodes, i.e. state-action-reward-state tuples (s_t, a_t, r_t, s_{t+1}) and then randomly sample from it during learning.

A schema of the process is shown in Figure 1.

1.4 The Gym environment

In this work we will try to solve two reinforcement learning environments, both developed by OpenAI in the Gym package:

- CartPole-v1, in which a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track, and the goal is to prevent it from falling over.
- LunarLander-v2, in which a lander needs to land at a specific location.

2 CartPole optimization

2.1 Methods

In the CartPole-v1 environment a pole 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. 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.

As we anticipated in Section 1.3, we will use a deep neural network to approximate the Q-values. To improve stability, we will implement experience replay and a target network. The target network has the same architecture of the policy network:

- a first linear layer with 4 input neurons (corresponding to the observables: cart position, cart velocity, pole angle, pole angular velocity) and 128 output neurons;
- a second linear layer with 128 inputs and outputs;
- a third linear layer with 128 inputs and 2 outputs (corresponding to the possible actions: push left or push right);

The activation function is the hyperbolic tangent. As optimizer we choose the stochastic gradient descent with no momentum, since it is known that in reinforcement learning it is not always good to keep the previous direction of the gradient. In fact, since training can be unstable and requires some trials and errors, it is better to keep things simple.
To speed up convergence we adopt the following strategies:

- we use an exponentially decreasing exploration profile with a softmax policy. The exploration profile computes the softmax temperature τ starting from an initial value τ_0 and then decays exponentially in way that keeps the same shape for different values of the number of episodes N. The temperature τ_i at episode i is given by:

$$\tau_i = \tau_0 \exp\left(-\frac{i\beta \log(\tau_0)}{N}\right) \tag{5}$$

  where β is a hyper-parameter that can be optimized (the default value is $\beta = 6$).
- we try to optimize the discount rate γ and the optimizer learning rate η;
- we add a penalty proportional to the absolute value of the distance from the center, in order to avoid the cart to exit from the screen:

$$reward = reward - |cartPosition|. \tag{6}$$

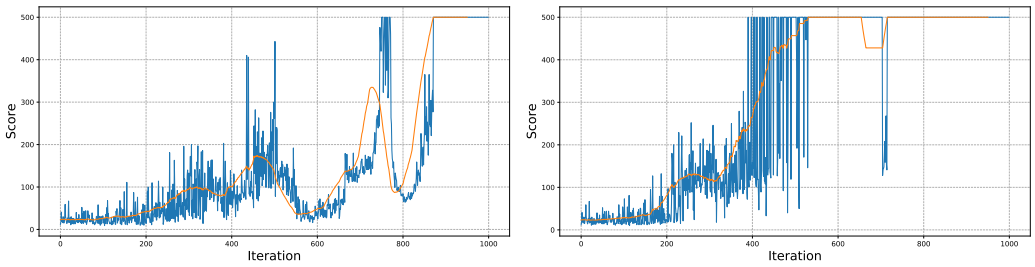
2.2 Results

After a (very little, due to limited computational power) random search, the best hyper-parameters turned out to be $\gamma = 0.99$ and $\eta = 0.05$. The plot of the scores obtained with these hyper-parameters is shown in Figure 2b. We see that the maximum score (500) is reached around the 390-th episode. With the configuration used during the lab, the same score is obtained around the 800-th iteration, as we can see in Figure 2a. So, just by tuning this two hyper-parameters we obtained a huge improvement.
If we change the learning curve, for example making it steeper, the maximum score is reached in a smoothest way, as shown in Figure 3.

3 CartPole with screen pixels

3.1 Methods

In this section we want to learn to control the CartPole environment using directly the screen pixels, rather than the compact state representation (cart position, cart velocity, pole angle, pole angular velocity). Since the image is quite large, we crop it and we rescale it, obtaining a image of size 80×100. Moreover, since the color does not matter, we consider only one channel. The result is shown in Figure 4.
Since we are dealing with images, we use a convolutional network. The convolutional section is made up of:



(a) Lab hyper-parameters: $\gamma = 0.97$, $\eta = 0.01$.

(b) Best hyper-parameters: $\gamma = 0.99$, $\eta = 0.05$.

Figure 2: Train scores with moving average (window size = 50) for CartPole with (a) default (used in the lab) and (b) optimized hyper-parameters. The temperature parameter is $\beta = 6$. As we can see, the maximum score (500) is reached much faster if we tune the long term reward parameter and the optimizer learning rate.

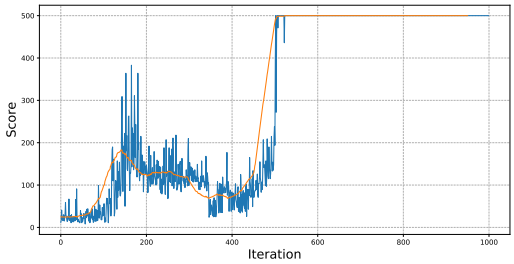


Figure 3: Train scores with moving average (window size = 50) for CartPole with optimized hyper-parameters and exploration profile parameter $\beta = 15$.

- a first convolutional layer with 64 filters of size 4×4, stride 2, padding 0;
- a second convolutional layer with 64 filters of size 4×4, stride 2, padding 0;
- a third convolutional layer with 32 filters of size 3×3, stride 1, padding 0.

The linear layer is composed of:

- a flatten layer;
- a first linear layer with 128 neurons;
- a second linear layer with 128 neurons;
- an output layer with 2 neurons.

3.2 Results

Figure 5 shows the plot of the scores. As we can see, the agent is not able to solve the environment: the score wildly oscillates since the beginning of learning, and the trend is pretty flat. We have also tried different crops of the image and rescalings, but the result does not change. This is probably due to the fact that using static images we loose information about the dynamics, which is crucial.

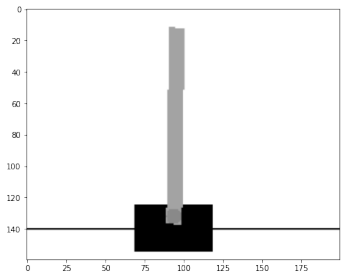


Figure 4: CartPole screen image resized and in greyscale.

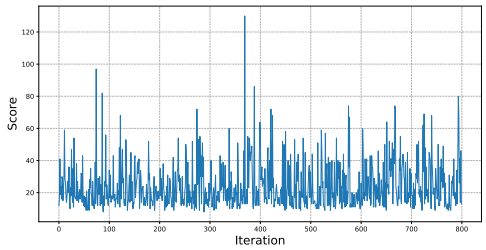


Figure 5: Train score for CartPole with pixels. The hyper-parameters used are $\gamma = 0.99$, $\eta = 0.05$ and $\beta = 6$.

4 LunarLander

4.1 Methods

In this section we will train a deep reinforcement agent on a different Gym environment. In particular, we will consider the LunarLander-v2 discrete environment. We have a landing pad that is always at coordinates $(0,0)$ and the lander has to land (possibly) in it. Reward for moving from the top of the screen to landing pad and zero speed is about 100-140 points. If the lander moves away from landing pad it loses reward back. The episode finishes if the lander crashes or comes to rest, receiving additional -100 or $+100$ points. Each leg ground contact is $+10$ and firing the main engine is -0.3 points each frame (fuel is infinite). The score at which the environment is considered to be solved is 200 points.
Four discrete actions are available: do nothing, fire left orientation engine, fire main engine and fire right orientation engine. The observables are the following: the x coordinate of the lander, the y coordinate of the lander, the horizonal velocity, the vertical velocity, the orientation in space, the angular velocity, left leg touching the ground (boolean), right leg touching the ground (boolean). All the coordinate values are given relative to the landing pad. 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.
For the deep Q-networks we use the following architecture:

- a first linear layer with 8 input neurons (corresponding to the observables) and 128 output neurons;
- a second linear layer with 128 inputs and 128 outputs;
- a third linear layer with 128 inputs and 64 outputs;
- a final linear layer with 64 inputs and 4 outputs (corresponding to the possible actions);

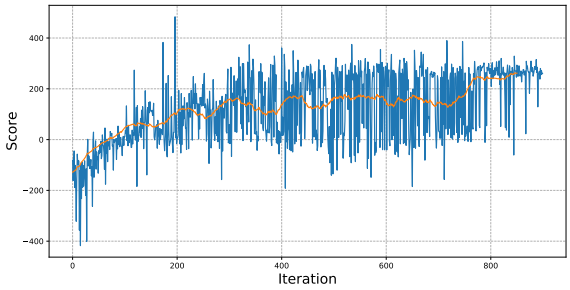


Figure 6: Train score for LunarLander, with moving average (window size = 50).

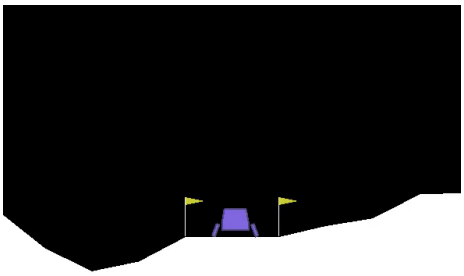


Figure 7: Final position of the lander for an event with score 281.

To optimize the learning procedure we add a bonus to the standard reward, which is obtained when the vertical velocity is negative:

$$reward = reward - verticalVelocity = reward + |verticalVelocity|. \tag{7}$$

In this way we "encourage" the lander to land instead of continuing to stay in the air.

4.2 Results

The plot of the training scores is shown in Figure 6. As we can see, the "solved" score (200) is reached quite fast, even if the reward rule we implement is very simple. The video of the best performance, with score 281, can be found together with this report and the code. The final frame is shown in Figure 7.
However, since the environment is much more complex compared to CartPole-v1, it would be interesting to explore much complex rewards. For example, we could add some bonuses if the lander maintains a good position with respect to the landing pad or if it decreases the speed to land smoothly.