Reinforcement Learning Lab

Lesson 8: Deep Q-Networks

Davide Corsi and Alberto Castellini

University of Verona
email: davide.corsi@univr.it

Academic Year 2022-23

Environment Setup

The first step for the setup of the laboratory environment is to update the repository and load the miniconda environment.

- Update the repository of the lab:

```
cd RL-Lab
git stash
git pull
git stash pop
```

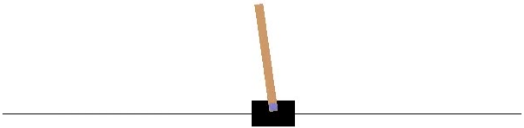
- Activate the *miniconda* environment:

```
conda activate rl-lab
```

- Install *Gymnasium* environments:

```
pip install gymnasium
```

Environment: CartPole



- A pole is attached by an unactuated joint to a cart, which moves along a frictionless track. 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 **state** of the environment is represented as a tuple of 4 values: *Cart Position*, *Cart Velocity*, *Pole Angle*, and *Pole Angular Velocity*.
- The **actions** allowed in the environment are 2: *action 0 (push cart to left)* and *action 1 (push cart to right)*.

Today Assignment

In today's lesson, we will implement the Deep Q-Network (DQN) algorithm to solve the CartPole problem. In particular, the file to complete is:

```
RL-Lab/lessons/lesson_8_code.py
```

Inside the file, two python functions are partially implemented. The objective of this lesson is to complete it.

- **def training_loop()**
- **def DQNUpdate()**

Expected results can be found in:

```
RL-Lab/results/lesson_8_results.txt
```

Suggestions and Code Snippets

1. In today's lesson, the code is already partially implemented. Some functions related to Numpy and Matplotlib should not be modified. All the *entry points* for your code are marked with the keyword TODO.
2. Some methods of Gymnasium (the *new* Gym version) can be slightly different with respect to DangerousGridworld. Following are some snippets for the most important functions:

```
# Generation and Reset of the environment
env = gymnasium.make( "CartPole-v1" )
state = env.reset()[0]
# To generate a random action
action = env.action_space.sample()
# The updated 'step' function
next_state , reward , terminated , truncated , info = env.step(action)
done = terminated or truncated
```

training_loop()

Require: *environment*, *neural_network*, *trials*, *expl_param*, *score_queue*
Ensure: *neural_network*, *score_queue*

initialize the experience buffer ▷ A fixed size queue
initialize the score queue ▷ An infinite size queue
for $i \leftarrow 0$ **to** *epochs* **do**
    initialize s observe current state
    repeat
        Select and execute action a ▷ ϵ-greedy approach
        Observe new state s' and receive immediate reward r
        Add (s, a, s', r) to experience buffer
        DQNUPDATE(*neural_network*, *buffer*) ▷ Call the training function
        update state s $\leftarrow$ s'
    until *s* is terminal
    update *score_queue*

return *score_queue*

DQNUpdate()

This function updates the neural network weights according to the formula:
$w_{t+1} = w_t - learning\_rate * gradient(R_{t+1} + gamma * max_a(\hat{Q}_{pi}(S_{t+1}, a, w_t)) - \hat{Q}_{pi}(S_t, A_t, w_t))$

Require: $neural\_network, experience\_buffer(MB), gamma$
Ensure: $neural\_network$

Sample mini-batch MB of experiences from buffer
for $s, a, s', r \in$ MB **do** ▷ (state, action, next_state, reward)
    target ← $\textsc{predict}(neural\_network, s)$
    if s' is terminal **then**
        target[a] = r
    else
        max-q = max($\textsc{predict}(neural\_network, s')$) ▷ max q-value from s'
        target[a] = r + (max-q * gamma)
    mse = $\mathrm{MSE}(neural\_net, state, target)$
    gradient = $\textsc{compute\_gradient}(mse, neural\_network)$
    $\mathrm{BACK\_PROPAGATE\_GRADIENT}(gradients, neural\_network)$
return $neural\_network$

Expected results

The plot on the right is the expected result. Notice that it has been obtained with an ϵ-greedy strategy, starting with $\epsilon = 1$ and multiplying it by 0.999 each epoch.

Seeding

Given the (particularly) high stochasticity of the method and the environment, for this lesson, we fixed a random seed equal to 15.

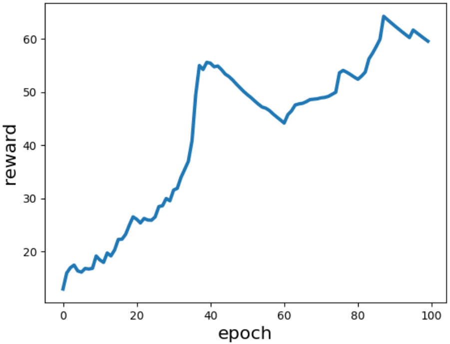


Figure: Note that obtaining this result requires time. You can stop the training after fewer iterations if you observe a growth in the reward.

Final Remarks

Pedagocic Implementation

Today's lesson presents a simplified version of the code. In the next lessons, we will see a more efficient implementation that exploits Numpy, matrix multiplications, and advanced TensorFlow tools.

Number of Updates

By default, the suggested implementation performs the DQN update only once for episodes. A more efficient implementation exploits more iterations. However, this would require some tricks to avoid overfitting on the data (e.g., memory buffer shuffle).