Reinforcement Learning Lab

Lesson 8: Deep Q-Networks

Luca Marzari and Alberto Castellini

University of Verona
email: luca.marzari@univr.it

Academic Year 2023-24

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
```

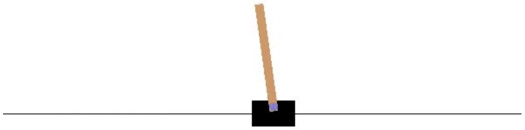
- Activate the *miniconda* environment:

```
conda activate rl-lab
```

- **Install new requirements**:

```
pip install gymnasium
pip install seaborn
pip install pandas
```

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 them.

- **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*, *update_rule*, *use_keras*, ε, *updates*, *episodes_to_train*
Ensure: *neural_network*, *score_queue*

initialize the experience buffer ▷ A fixed size queue
initialize the score queue ▷ An infinite size queue
for *ep* $\leftarrow$ 0 **to** *episodes* **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*, *use_keras*, *buffer*) ▷ Call the training function specifying which framework to use
        update state $s \leftarrow s'$
    until s is terminal
    update *score_queue*

return *score_queue*

DQNupdate()

This function updates the neural network weights according the selected framework and using to the formula:

$$w_{t+1} = w_t - learning\_rate * \nabla[R_{t+1} + \gamma * \max_a(\hat{Q}_\pi(S_{t+1}, a, w_t)) - \hat{Q}_\pi(S_t, A_t, w_t)]$$

Require: *neural_network*, *use_keras*, *experience_buffer*(*MB*), *optimizer*, *batch_size*, γ
Ensure: *neural_network*

Sample mini-batch MB of experiences from the buffer
for $s, a, s', r \in$ MB **do** ▷ (state, action, next_state, reward)
    target ← PREDICT(*neural_network*, *s*)
    if s' is terminal **then**
        target[a] = r
    else
        max-q = max(PREDICT(*neural_network*, *s'*)) ▷ max q-value from s'
        target[a] = r + (γ * max-q)
    mse = MSE(*neural_net*, *state*, *target*)
    gradient = COMPUTE_GRADIENT(*mse*, *neural_network*)
    BACK_PROPAGATE_GRADIENT(*gradients*, *neural_network*)
return *neural_network*

Expected Results

Expected results

The plot on the right is the expected result (it can be slightly different). Notice that we used an ϵ-greedy strategy, starting with $\epsilon = 1.0$ and multiplying it by 0.99 for each episode.

Seeding

Given the high stochastic nature of both the method and the environment, evaluating every algorithm on multiple random seeds (at least 10) is essential before stating the correctness of the proposed solution!

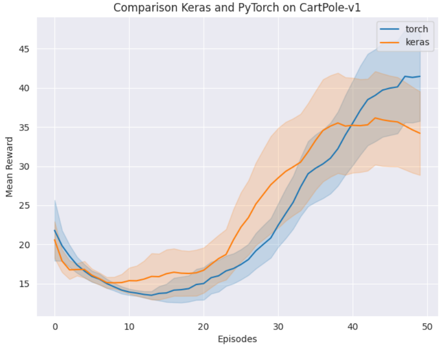


Figure: Mean and Standard Error using 10 random seeds for both Keras and PyTorch DQN implementation. Note that obtaining this result requires time.

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 and PyTorch 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 the data (e.g., memory buffer shuffle).