Practice 4
Deep Reinforcement Learning
CartPole with DQN altgorithm

Learning Objectives

The objective of this Practice is to built a DQN algorithm for Cartpole. In this practice you must use a neural network to build a Q-Learning table that will control the pole using the DQN algorithm.

The DQN algorithm was introduced in a famous paper where David Silver and Mihn showed how to use this approach to solve Atari Games. The reference of this document is [Mni+13].

1 Files for this Practice

```
030_DQN_CARTPOLE_Keras_(empty).ipynb
```

2 CartPole environment Background

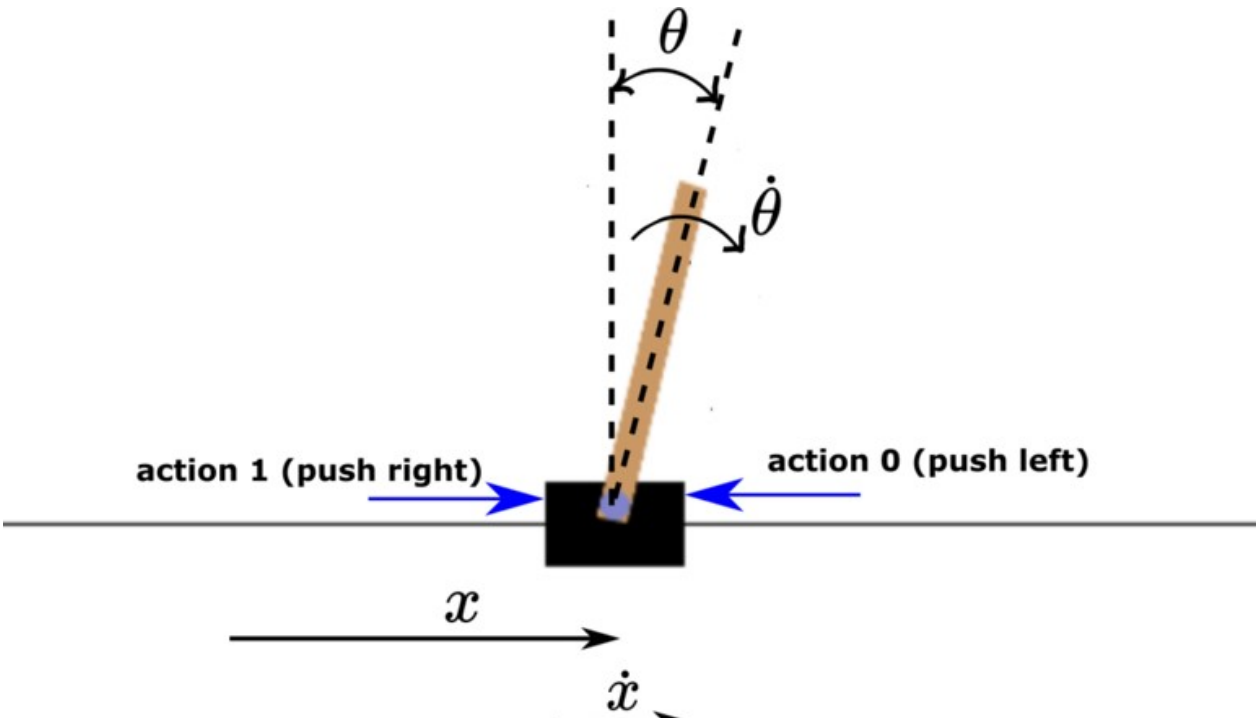

Figure 1: Cartpole environemnt and movement dynamics

The CartPole environment in Gymnasium is a classic control problem where the objective is to balance a pole on a cart. The cart can move left or right along a frictionless track, and the pole is attached to the cart by an unactuated joint. The system receives a reward for every time step the pole remains upright, with the goal being to maximize this reward by preventing the pole from falling over. The environment is characterized by continuous state space representing the cart's position and velocity, as well as the pole's angle and angular velocity, while actions are discrete, consisting of applying a force either left or right on the cart. The challenge lies in developing a control strategy that effectively maintains the pole's balance despite the inherently unstable nature of the system.

The control objective is to keep the pole in the vertical position by applying horizontal actions (forces) to the cart. The **action space** consists of two actions

- Push the cart left – denoted by 0
- Push the cart right – denoted by 1

The **observation space** for the states is:

1. Cart position, denoted by x in Fig. 1. The minimal and maximal values are -4.8 and 4.8, respectively.
2. Cart velocity, denoted by $\dot{x}$ in Fig. 1 The minimal and maximal values are $-\infty$ and ∞, respectively.
3. Pole angle of rotation (measured in radians), denoted by θ. The minimal and maximal values are -0.418 radians (-24 degrees) and 0.418 (24 degrees).
4. Pole angular velocity, denoted by $\dot{\theta}$. The minimal and maximal values are ∞ and ∞, respectively.

It is also important to emphasize that initial states (initial observations) are completely random and the values of states are chosen uniformly from the interval (-0.05,0.05).

3 Introduction to Deep Q-Networks (DQN)

Deep Q-Networks (DQN) combine Q-learning with deep neural networks to handle high-dimensional state spaces. The key ideas behind DQN are:

1. Use a neural network as a function approximator for the Q-value function.
2. Implement experience replay to break correlations in the observation sequence and smooth over changes in the data distribution.
3. Use a separate target network to reduce correlations with the target.

DQN - Deep Q Network (Mnih, et al. 2015)

Initialize replay memory R with capacity N
Initialize Q-Network with random weights
Initialize target network Q_target with weights θ target $= \theta$
Set learning_rate α, γ, ϵ
For each episode:
  Initialize s
  While s is not terminal:
    Sample action A with ϵ-greedy policy
    Take action A observe R and next state S'
    Store transition (S, A, R, S', done) in replay memory D
    If replay memory D has sufficient samples:
      Sample a minibatch of transitions from D
      Compute target:
      If $done_j$:
        $y_j = r_j$
      else:
        $y_j = r_j + y * max(Q : target(s_j^{'}, a^{'}, \theta_{target}))$
      Perform Gradient descent step on loss:
        Loss $= y_j - Q(s_j, a_j, \theta))^2$
    Every C steps, update target network
      $\theta_{target} - \theta$
    Update state $s = s^{'}$

There are two novelties in this algorithm respect the previous ones

- The replay buffer
- Optimizing the loss function

3.1 The replay buffer

To perform an experience replay, the algorithm stores all of the agents experiences $s_t.a_t, r_t, s_{t+1}$ at each time step in a data set. Normally in a q-learner, we would run the update rule on them. But, with experience replay we just store them.

Later during the training process these replays will be drawn uniformly from the memory queue and be ran through the update rule. There are 2 ways to handle this and I have coded both in the past. The first is to run them on every loop and the other is to run them after X amount of runs. In this code below, I run them each time.

The replay buffer that addresses stability and efficiency challenges in training DQN.

It has several objectives,

- **Breaking correlation between experiences**. consecutive states in the environment are often highly correlated. Training directly on this data can cause the neural network to overfit to specific sequences of events and destabilize learning. The replay buffer stores past experiences (state, action, reward, next state) and samples them randomly during training. This randomization helps break the correlation between the consecutive states.
- **Improve data efficiency:** Each interaction with the environment is stored in the buffer and reused multiple times for training. This increases the efficiency of learning since valuable experiences are not discarded after a single use.
- **Off-Policy learning** DQNs are off-policy algorithms, meaning the behavior policy used to collect data can differ from the target policy being optimized. The replay buffer allows the agent to learn from past experiences generated by older policies increasing the available reach of the off-policy search.

3.2 Optimizing the loss function

As in training neural networks we will use an optimization approach to obtain the optimal Q function. Firstly we define a loss function that we will optimize using gradient descent or similar algorithms.

The Loss function calculates the error between the actual Q-value estimate and the target Q-value, derived from the Bellman Equation (see equation 1)

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a; \theta)\right)^2\right] \quad (1)$$

- $L(\theta)$ is the loss function parameterized by θ.
- $\mathbb{E}$ denotes the expectation.
- $(s, a, r, s') \sim U(D)$ indicates that the tuple (s, a, r, s') is sampled from a replay buffer D uniformly.
- r is the reward received after taking action a in state s.
- γ is the discount factor.
- $\max_{a'} Q(s', a'; \theta')$ is the maximum estimated future reward for the next state s' and all possible actions a', using the target network parameterized by θ'.
- $Q(s, a; \theta)$ is the estimated reward for taking action a in state s, using the current network parameterized by θ.

4 Practice: Implementing DQN for CartPole

In this practice, you will implement a DQN agent to solve the CartPole environment from OpenAI Gym. AS we already discussed in previous assignments there are two major approaches to Deep Learning, TORCH and KERAS. In our exercise we will use KERAS, but you can find resources like [Mor20] based in pytorch.

I provide you with 3 programs each one with different levels of complexity and efficiency. Try them one by one. It is important to use vectorization to speed up the program advance. To understand what we mean for vectorization compre the model 'PLAIN' with the model 'VECTORIZED'.

The third program is much more complex and it has all different kinds of tricks. You can apply them when your basic solution is working.

4.1 Setup

First, install the required libraries, you must install torch as you will have to use a Neural Network.:

Make sure you have installed KERAS and gymnasium correctly in your environment

4.2 The different parts of the program

Implement an agent trained with DQN with the following components:

1. Q-Network: A neural network that takes the state as input and outputs Q-values for each action.
2. Replay Buffer: A memory to store and sample experiences.
3. Epsilon-greedy policy: For exploration-exploitation balance.

Use the following flow and try to get each part from the 3 programs provided

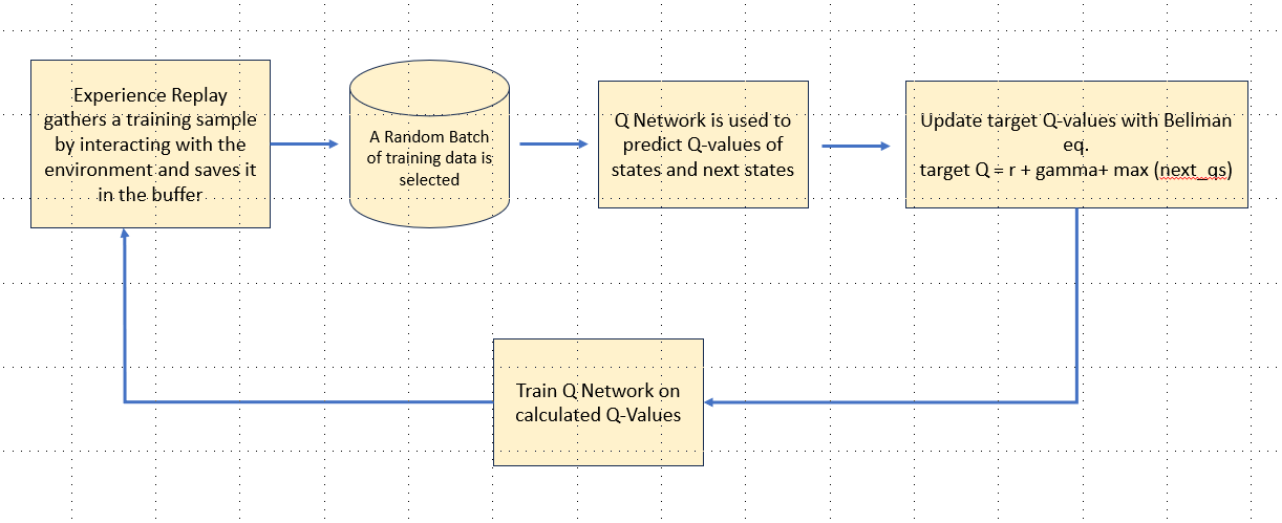

Figure 2: DQN Flow for a program

You should note that in this example you don't need discretization (as you used in the previous Q-Learning approach with CARTPOLE), as the neural network can generate real numbers, not only natural numbers.

4.3 Analyzing the performance

To analyze the performance of DQN methods we would like to understand how the training evolves over time and when the threshold of the agent (the point where we consider the agent has completely learned

the environment). You should print a figure similar to 3 in your example. To make this figure possible, append each reward to a list of cumulative rewards and rolling rewards (depending on a window - 20 in the example) after each inner loop finishes. The solved threshold is at the 195-200 reward line.

In this way we can compare different executions. to see how long it takes the agent to train. Time your executions as well to see how long it takes. Use the time package and see how long it takes for your progrma to complete.

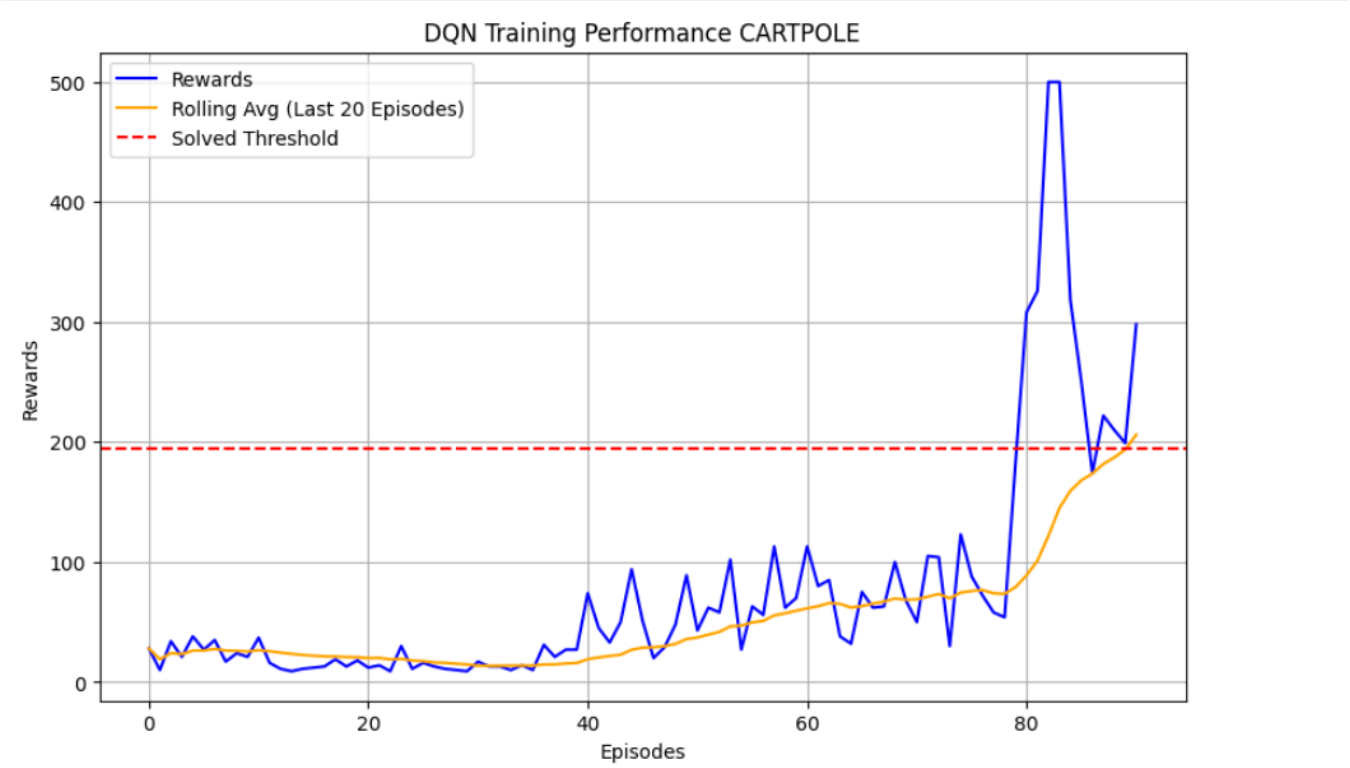

Figure 3: DQN post-mortem training analysis

Another practical add-on to the program is to test the Q-learned function 10 times. You can do it with a code like this one (after the DQN function has been trained and threshold has been reached)

```python
# Testing for 10 episodes
start_time = time.time()

for e_test in range(10): # Run 10 test episodes
    state, _ = env.reset()
    state = np.reshape(state, [1, state_size])
    total_reward = 0

    steps = 0
    while True:
        # Use the trained model for testing
        action_vals = DQN.predict(state, verbose=0) # Predict action values
        action = np.argmax(action_vals[0]) # Choose the action with the highest Q-value

        next_state, reward, done, truncated, _ = env.step(action)
        next_state = np.reshape(next_state, [1, state_size])
        total_reward += reward
        state = next_state
        steps = steps + 1
        if done or truncated:
            print(f"Test Episode: {e_test + 1:2}/10, Reward: {total_reward:.2f}, Steps: {
                steps:3}")
            break

end_time = time.time()
testing_duration = (end_time - start_time) / 60 # Convert to minutes
print(f"Testing completed in {testing_duration:.2f} minutes")
```

Listing 1: Testing Network 10 episodes

4.4 Some comments

1. Complete the implementation of the DQN using examples to get ideas (the most complete is third)
2. Don't use classes programming. I prefer traditional programming as it is easier to understand
3. Add a function to test you passed the objective (195/100 episodes)
4. Try the simple network. If you have time increase the network size to make it more intelligent and see what happens.

5 DDQN

If you have some time you may be interested in trying an implementation of DDQN. First Finish your DQN implementation and then try to add the second network. In the list of examples, you have 2 DDQN notebooks you can use as an example.

Again, as we are tight on time, if you feel you are overstretched, don't do it. Is better to focus on DQN

DQN is far from being an optimal approach to learning. As we are optimizing the Agent by training the network by using a single network that selects and evaluates the actions. This leads to a problem, is like measuring the distance between you and an object that is moving as well, you maybe think are close to the object but the next time you measure the object has moved away and the distance has grown.

Technically it is said the the DQN algorithm overestimates the Q_values, becoming one of the reasons of its slowness.

The DDQN algorithm is designed to solve this issue.

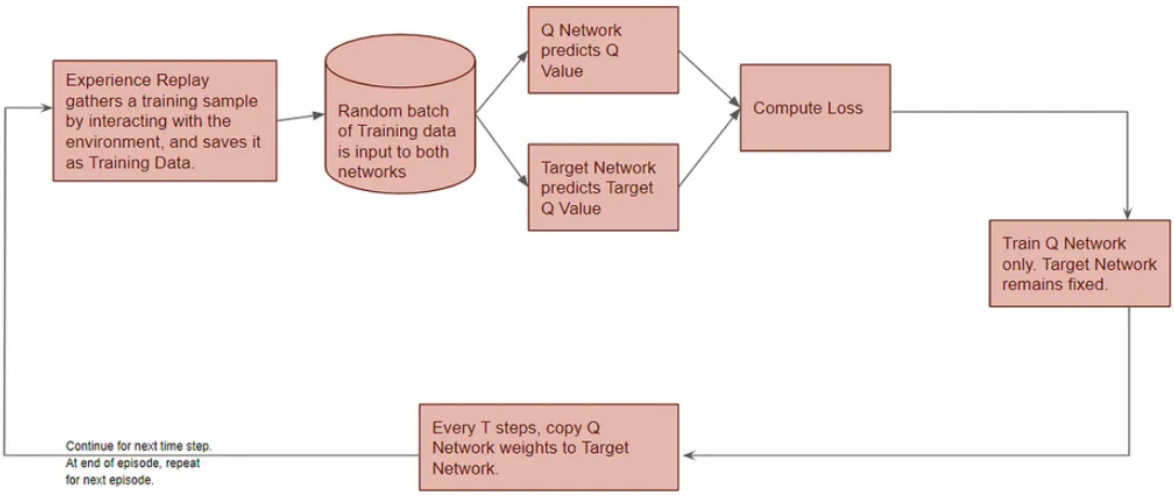

Figure 4: DDQN Flow for a program. It uses two networks

The algorithm works as follows

In DQN, the target for the Q-value update is computed as:

$$y = r + \gamma \max_a Q_{\text{target}}(s', a)$$

Here:

- r is the reward.
- γ is the discount factor.
- Q_{target} is the target Q-network.
- s' is the next state.

The same network is used to both:

1. Select the action a using $\max_a$.
2. Evaluate the value of that action.

This coupling leads to an **overestimation bias**, as the Q-value may be overestimated due to the inherent inaccuracies of the neural network.

Key Idea of DDQN

DDQN addresses this issue by separating the action selection and evaluation processes. The target value in DDQN is computed as:

$$y = r + \gamma Q_{\theta'}(s', a')$$

where:

- $a' = \arg\max_a Q_\theta(s', a)$ is the action selected using the **main network**.
- $Q_{\theta'}(s', a')$ is the value of a' evaluated using the **target network**.

This decoupling reduces overestimation bias and leads to more stable training.

DDQN Algorithm Steps

1. **Initialize Networks:**
   - Main network Q_θ.
   - Target network $Q_{\theta'}$, initialized with the same weights as the main network.
2. **Experience Replay:**
   - Store transitions (s, a, r, s') in a replay buffer.
   - Sample a mini-batch of experiences during training.
3. **Action Selection:**
   $$a = \arg\max_a Q_\theta(s, a)$$
4. **Target Calculation:**
   - Use the main network to select the next action:
     $$a' = \arg\max_a Q_\theta(s', a)$$
   - Use the target network to evaluate the value of a':
     $$y = r + \gamma Q_{\theta'}(s', a')$$
5. **Update Main Network:** Minimize the loss:
   $$L(\theta) = \mathbb{E}\left[(y - Q_\theta(s, a))^2\right]$$
6. **Update Target Network:** Periodically update the target network to match the main network:
   $$\theta' \leftarrow \theta$$
7. **Repeat:** Continue the process until convergence or a stopping criterion is met.

Advantages of DDQN

- **Reduced Overestimation Bias:** Decoupling action selection and evaluation leads to more accurate value estimates.
- **Improved Stability:** Training becomes more stable compared to traditional DQN.
- **Scalability:** Suitable for high-dimensional state and action spaces.

Comparison: DQN vs DDQN

- **DQN Target:**

$$y = r + \gamma \max_a Q_{\text{target}}(s', a)$$

- **DDQN Target:**

$$y = r + \gamma Q_{\text{target}}(s', \arg\max_a Q_{\text{main}}(s', a))$$

6 Questions

With your experience with the programm respond to the following questions

1. What Neural Network structure have you chosen
2. What is your best execution? How long it took to reach the threshold?
3. Does the Replay buffer size has any impact? (try several sizes from small to very large)
4. Does the learning rate has any impact?
5. What is your preferred epsilon decay function?

7 Submission

Submit your Python notebook in one zip

- Don't include all your executions, submit only the best one you had. If you tried DDQN include it in an additional notebook file.
- Include your answers in another file (word) into the zip file
- remember to put your name in the zip name

References

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *Arxiv* (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: `http://arxiv.org/abs/1312.5602`.

[Mor20] Miguel Morales. "Grokking Deep Reinforcement Learning". In: (2020). Ed. by Manning Publications Co. ISBN=9781617295454.