Deep Reinforcement Learning - Assignment 2

Yoel Bokobza - 312190481, Shahaf Yamin - 311530943

July 23, 2021

1 Introduction

We were asked to solve a control problem with continuous state and discrete action spaces using Deep Q-Network(DQN) in this exercise. The environment is the CartPole environment. The goal of CartPole is to balance a pole connected with one joint on top of a moving cart. We define the state space $\mathcal{S} \subset \mathbb{R}^4$ where each state $s \in \mathcal{S}$ contains the following information $s =$ [cart position, cart velocity, pole angle, pole angular velocity]. where each dimension belongs to the following sets:

- $\textit{cart position} \in [-4.8, 4.8]$
- $\textit{cart velocity} \in [-\infty, \infty]$
- $\textit{pole angle} \in [-0.418, 0.418]_{radians}$
- $\textit{pole angular velocity} \in [-\infty, \infty]$

The action space $\mathcal{A}$ is discrete $\mathcal{A} = \{push\ cart\ to\ the\ left, push\ cart\ to\ the\ right\} = \{0, 1\}$.
The reward is $+1$ for every step taken. An episode is terminated if:

- The pole angle goes beyond ±12 degrees or ±0.2094 rad.
- The cart position is larger than 2.4.

We define the score as the total reward (= total number of steps made) received in one episode. The overall performance of the agent is evaluated by using a task score. The task score is defined as the maximum of the minimum score over five consecutive episodes.

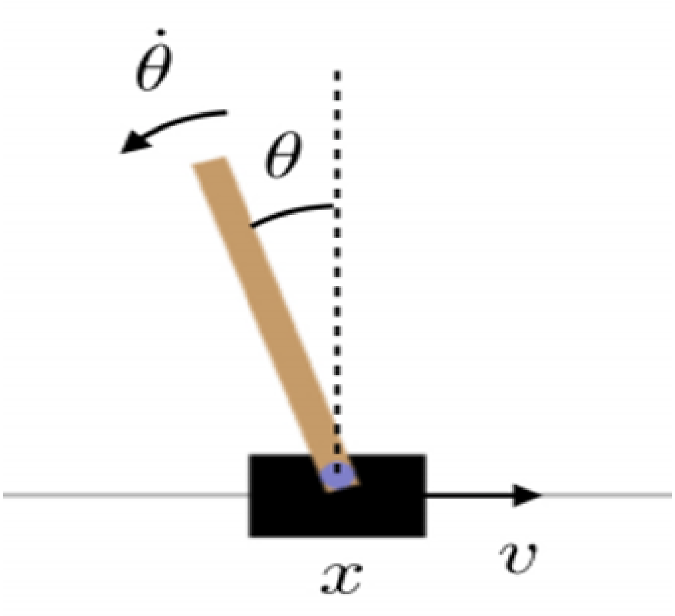

Figure 1: CartPole Illustration.

2 DQN Background

DQN is based on the Q-learning algorithm with a Neural Network function approximation. Reinforcement learning may suffer from instability or even diverge when a nonlinear function approximator such as a neural network represents the action-value function. DeepMind illustrated two new features to overcome this issue – The replay buffer and the target network (which we will describe in the following sections). The training procedure of a neural network aims to minimize a loss function. In other words, we try to find parameters θ that minimizes the loss function $J(\theta)$. For example, if we choose to use the mean squared error (MSE) as our loss, we define $J(\theta)$ as followed:

$$J_{MSE}(\theta) = \frac{1}{2}\mathbb{E}_\pi[(q_\pi(S_t, A_t) - \hat{q}(S_t, A_t; \theta))^2] \tag{1}$$

Gradient descent method can be used to find local optima.

$$\nabla_\theta J_{MSE}(\theta) = -\mathbb{E}_\pi[q_\pi(S_t, A_t) - \hat{q}(S_t, A_t; \theta)]\nabla_\theta \hat{q}(S_t, A_t; \theta) \tag{2}$$

Because we don't know the expectation, we can estimate it using a batch of samples (according to the law of large numbers, and the assumption of an i.i.d samples),

$$\nabla_\theta J_{MSE}(\theta) \approx -\frac{1}{|B|}\sum_{(s,a)\in B}[q_\pi(s, a) - \hat{q}(s, a; \theta)]\nabla_\theta \hat{q}(s, a; \theta) \tag{3}$$

Where B is the batch set. The problem with this method is that we do not know $q_\pi(s, a)$, so we substitute this value with an estimated $q_\pi(s, a)$. We can use, for example, a Monte Carlo estimate or TD(0). In the DQN, we are estimating this value according to the following equation:

$$q_\pi(S_t, A_t) \approx R_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a'; \theta) \tag{4}$$

using this estimation, we get:

$$\nabla_\theta J_{MSE}(\theta) \approx -\frac{1}{|B|}\sum_{(s,a,r,s')\in B}[r + \gamma \max_{a'} \hat{q}(s', a'; \theta) - \hat{q}(s, a; \theta)]\nabla_\theta \hat{q}(s, a; \theta) \tag{5}$$

We calculate the gradient under the assumption that we have the actual value of $q_\pi(s, a)$, and then, we are replacing this value with our estimation of $q_\pi(s, a)$. This procedure is called "Semi-Gradient" because we are not calculating the actual gradient. In addition, we update the parameters using the gradient descent method. The following formula is the parameters update rule of the gradient descent algorithm:

$$\theta \leftarrow \theta - \alpha\nabla_\theta J(\theta) \tag{6}$$

for some $\alpha > 0$. In this exercise, we are using the Huber loss function. Which is a mixture of the MSE and MAE losses where,

$$J_{MAE}(\theta) = \mathbb{E}_\pi[|q_\pi(S_t, A_t) - \hat{q}(S_t, A_t; \theta)|] \tag{7}$$

The Huber loss is defined as

$$J_{Huber}(\theta) \approx \frac{1}{|B|}\sum_{(s,a)\in B} L(s, a) \tag{8}$$

Where B is the batch set and $L(s, a)$ sustains:

$$L(s, a) = \begin{cases} \frac{1}{2}(q_\pi(s, a) - \hat{q}(s, a; \theta))^2, & \text{if } |q_\pi(s, a) - \hat{q}(s, a; \theta)| < 1. \\ |q_\pi(s, a) - \hat{q}(s, a; \theta)| - 0.5 & \text{otherwise.} \end{cases} \tag{9}$$

Now, similarly to the MSE case, we can substitute the value of $q_\pi(s,a)$ with an estimation of it (see Equation 4.).

$$L(s,a,r,s') = \begin{cases} \frac{1}{2}(r + \gamma \max_{a'} \hat{q}(s',a';\theta) - \hat{q}(s,a;\theta))^2, & \text{if } |r + \gamma \max_{a'} \hat{q}(s',a';\theta) - \hat{q}(s,a;\theta)| < 1. \\ |r + \gamma \max_{a'} \hat{q}(s',a';\theta) - \hat{q}(s,a;\theta)| - 0.5 & \text{otherwise.} \end{cases} \tag{10}$$

We define that $\hat{q}(s',\cdot;\theta) = 0$ if s' is a terminal state.
Finally we get:

$$J_{Huber}(\theta) \approx \frac{1}{|B|} \sum_{(s,a,r,s') \in B} L(s,a,r,s') \tag{11}$$

The general purpose of using the Huber loss is to avoid enormous "punishment" for outliers. Moreover, as part of the program, PyTorch is calculating the estimated gradient, e.g. $\nabla_\theta J_{Huber}(\theta)$ using the back-propagation procedure as we learned in class.

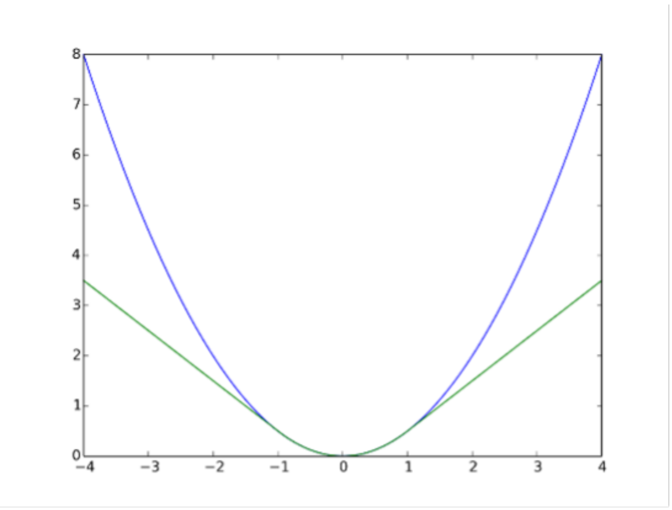

Figure 2: Example for Huber versus MSE, Huber in green, MSE in blue.

3 Question A

In this section, we completed the function *train_model*() in DQN.py by implementing the barebone of DQN.

3.1 Action selection

we've implemented the *select_action*(s) function which get as input a state and return an action according to the ϵ-greedy policy. This function can be described mathematically, as followed

$$\pi(s) = \begin{cases} \text{uniformly random action}, & \text{w.p. } \epsilon. \\ \text{argmax}_a\, \hat{q}(s, a; \theta), & \text{w.p. } 1 - \epsilon. \end{cases} \tag{12}$$

Here's the code implementation of this functionality:

```
def select_action(s,policy_net,epsilon):
    ’’’
    This function gets a state and returns an action.
    The function uses an epsilon-greedy policy.
    :param s: the current state of the environment
    :return: a tensor of size [1,1] (use ’return torch.tensor([[action]], device=device,
    dtype=torch.long)’)
    ’’’

    #with epsilon probability
    if np.random.uniform() < epsilon:
    # This case we choose random action
        # Pick action randomly
        action = np.random.choice(network_params.action_dim)
         # return the action as [1,1] tensor
        return torch.tensor([[action]], device=device, dtype=torch.long)
    else:
    # this case we choose action according to the greedy policy
        with torch.no_grad(): #To reduce memory usage
            return policy_net(s).max(1)[1].view(1, 1)
```

3.2 Current Action Value calculation

The policy_net function returns the q values for each state in the batch and, respectively, the possible actions. The PyTorch method 'gather' picks the q value for each state in the batch according to the corresponding action.

```
    # Compute curr_Q = Q(s, a) - the model computes Q(s), then we select the columns of the
    taken actions.
    # Pros tips: First pass all s_batch through the network
    #            and then choose the relevant action for each state using the method ’gather’
    curr_Q = policy_net(state_batch).gather(1, action_batch)
```

3.3 Expected Action Value

The expected_Q calculates the target network value for each element in the batch. for the i^{th} tuple in the batch – $(s_i, a_i, r_i, s'_i, done)$ the target is

$$\hat{y}_i = \begin{cases} r_i, & \text{if done==True} \\ r_i + \gamma \max_{a'} \hat{q}(s'_i, a'; \theta^-), & \text{otherwise} \end{cases} \tag{13}$$

Where in DQN $\hat{q}(s'_i, a'; \theta^-)$ is the target network (which we will describe later on).

```
# Compute expected_Q (target value) for all states.
#
# calculate the values for all next states ( Q_(s', argmax_a(Q_(s')) )
next_state_value_function = torch.max(target_policy_net(next_states_batch),
dim=1).values.detach()
# masking next state's value with 0, where not_done is False (i.e., done) and calculate the
target value.
expected_Q = reward_batch + torch.mul(next_state_value_function *
torch.tensor(not_done_batch, device=device), params.gamma)
#expand the dimension of expected_Q such that it will fit to curr_Q dimensions
expected_Q = expected_Q.unsqueeze(1)
```

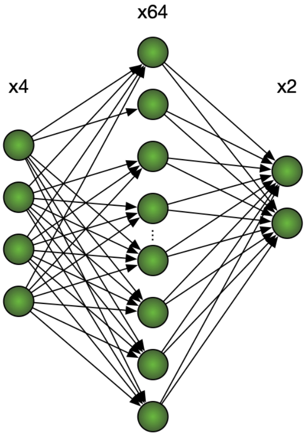

Figure 3: Q-network architecture, the input is the state, and the output is the estimated Q function for this state and each action.

4 Question B

Replay Buffer

The i.i.d assumption (that is crucial for Neural Network training) does not hold because there is a high correlation between consecutive samples in a given trajectory. The solution for this problem is to use a Replay Buffer. The replay buffer with a capacity N, stores transitions $(S_t, A_t, R_{t+1}, S_{t+1})$ and in each training session, a batch of size $|B|$ is uniformly sampled from the buffer. In contrast to consuming samples online and discarding them after that, uniform sampling from the stored experiences means they are less heavily correlated and can be re-used for learning. Usually, the replay buffer has a fixed capacity of size N, and it's stores transitions using First-In-First-Out(FIFO) queue management policy.

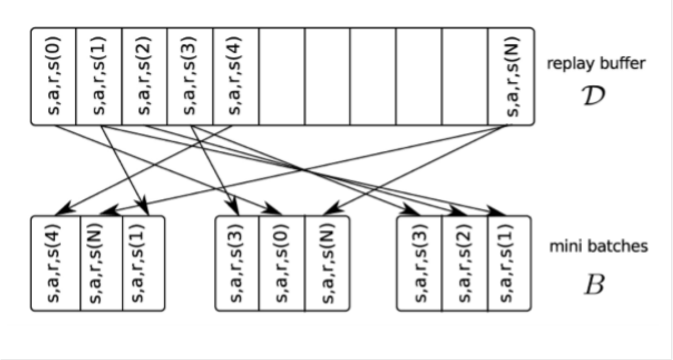

Figure 4: Replay Buffer with capacity N, and mini batches the sampled from it.

In this work $N = 100000$, we've implemented the buffer via python dequeue, an ordered collection of items like the queue. It has two ends, a front and rear, and the items remain positioned in the collection. We had to implement two methods –

- Push – pushing experience tuple into the buffer.

```
def push(self, *args):
    self.memory.append(Transition(*args))
```

- Sample – sample mini batches uniformly random from the buffer.

```
def sample(self, batch_size):
    batch = random.sample(self.memory,batch_size)
    return batch
```

5 Question C

Hard Target Network-Update

In the naive implementation, we use a single network for both target and policy networks, i.e. $\theta^{-} = \theta$, which may cause instability issues. when using a single network, we get an estimation of $q_{\pi}(S_t, A_t)$ which depends on the same parameters θ as the parameters of $\hat{q}(S_t, A_t; \theta)$. On the contrary, in supervised learning, the target should be independent of the network parameters, and in our case, the target (the estimation of $q_{\pi}(S_t, A_t)$) depends on the parameters θ. The problem is that the network aims to modify its parameters to push the estimated q values into their actual values. In the naive implementation of a single network, we adjust the parameters to push curr_Q towards the target $\hat{y}$. Still, because the target $\hat{y}$ depends on θ, as we adjust θ, we are also adjusting the target. This case usually causes instability because it is like "chasing our own tail." The solution suggested by DeepMind is to define two networks; the policy network $\hat{q}(s, a; \theta)$, and the target network - $\hat{q}(s, a; \theta^{-})$. The key idea is to create a "periods of supervised learning" by using the target network $\hat{q}(s, a; \theta^-)$ for the target calculation

$$\hat{y}_i = \begin{cases} r_i, & \text{if done==True} \\ r_i + \gamma \max_{a'} \hat{q}(s'_i, a'; \theta^-), & \text{otherwise} \end{cases} \tag{14}$$

The target network is a copy of the policy network, and it is updated periodically, i.e., every $C \in \mathbb{N}$ iterations we update the target network parameters with the policy network parameters $\theta^- \leftarrow \theta$. Following this method, the target values are fixed for some period C, and thus during these periods, the policy network training procedure acts like supervised learning.

First, we initialized the target network with the weights of the policy network –

```
# Build neural networks
policy_net = Network(network_params, device).to(device)
# Build a target network
target_policy_net = Network(network_params, device).to(device)
# Initialize the target network with the weights/biases of the policy network
for target_param, local_param in zip(target_policy_net.parameters(), policy_net.parameters()):
    target_param.data.copy_(local_param.data)
target_policy_net.eval()
```

Then, we’ve implemented the hard target network-update rule:

```
    if params.target_update == ’hard’:
        # update every params.target_update_period episodes the target network
        if i_episode % params.target_update_period == 0:
            for target_param, local_param in zip(target_policy_net.parameters(),
                                                 policy_net.parameters()):
                target_param.data.copy_(local_param.data)
```

6 Question D

Soft Target Network-Update

Using hard updates may cause significant changes after each update and thus induce instability. In the soft update, the target network is slowly updated after each episode towards the policy network by the following update rule:

$$\theta^- \leftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta^- \tag{15}$$

From this update rule, we can see that for $\tau \ll 1$, θ^- is changing very slowly towards θ, in our case $\tau = 0.01$ and therefore, we may deduce that $\tau \ll 1$.

```
    # soft target update
            if params.target_update == ’soft’:
                for target_param, local_param in zip(target_policy_net.parameters(),
    policy_net.parameters()):
                    target_param.data.copy_(params.tau * local_param.data + (1 - params.tau) *
    target_param.data)
```

The problem that may arise with this method is that small changes in the parameters space of θ^-, can cause significant changes in the q-target estimation $\hat{q}(s, a; \theta^-)$, which in turn, can harm the aforementioned “periodically supervised” idea.

7 Question E

Effect of Replay Buffer and Target Network

This section will study how the replay buffer and the target network affects the performance of the DQN. We have evaluated the performance of our DQN agent for different scenarios using the average score of five different experiments. Each experiment consists of 200 episodes, each of maximally 500 steps. We have run the previous experiment for both soft and hard target updates for methods with target networks. The training hyper-parameters are the parameters provided with the code.

```
training_params = {'batch_size': 256 (With replay), 'gamma': 0.95, 'epsilon_start': 1.1,
    'epsilon_end': 0.05, 'epsilon_decay': 0.95, 'tau': 0.01, 'target_update_period': 15,
    'grad_clip': 0.1}
```

target update	with replay with target	with replay without target	without replay with target	without replay without target
hard	169.4±40.76	x	22±7.13	x
none	x	21.4±4.03	x	21.2±7.73
soft	207.8±149.4	x	22.8±6.17	x

Table 1: Experiments results - average and standard deviation.

From the table and figures below, we can deduce that using replay buffer and target network is crucial for improving the agent performance under this environment setup. Those results can be explained by the advantages of both *replay buffer* and *target network* methods, which we already explained in the previous sections. In addition, we can see that although the soft update outperformed the hard update on average, it suffer from high variance. Thus, we can observe that in this problem, there is a trade-off between the hard and soft updates; one is better on average but suffers from high variance, while the other suffers from low average performance but with lesser variance. Moreover, we can see from the following graphs that all the methods which do not include both *replay buffer* and *target network* suffer from divergence in both loss and mean Q error criteria.

We observe exciting phenomena under the hard update target strategy; there are spikes in the Q error and loss graphs in every constant interval. This happens because we update the target network with the policy network every 15 episodes, e.g. we clone the policy network weights to the target network. After that, during the training procedure of the policy network, we aim to minimize the loss function between the target $\hat{y}$ and the policy network's estimation. Which, in turn, leads to a minimization of mean Q error. Thus, we see that the mean Q error is getting closer to 0 between the target network updates. In addition, we observe that those spikes are decreasing as we go further in the training procedure; this suggests that our training procedure tends to converge under this setup.

Under the soft update target strategy, we are slightly updating the target network every time step. Thus, we see that the loss and mean Q error change moderately by observing the soft target and replay buffer figures.

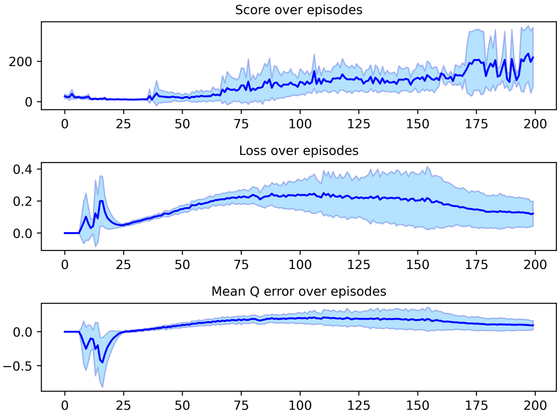

Figure 5: With Replay Buffer and soft update for target network

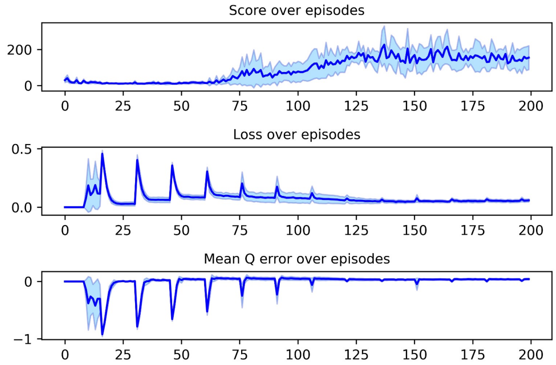

Figure 6: With Replay Buffer and hard update for target network

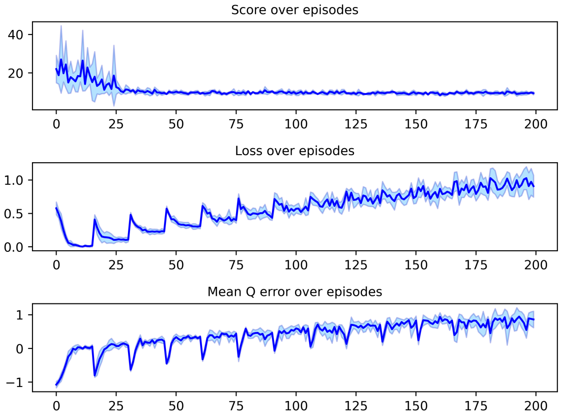

Figure 7: Without Replay Buffer and hard update for target network

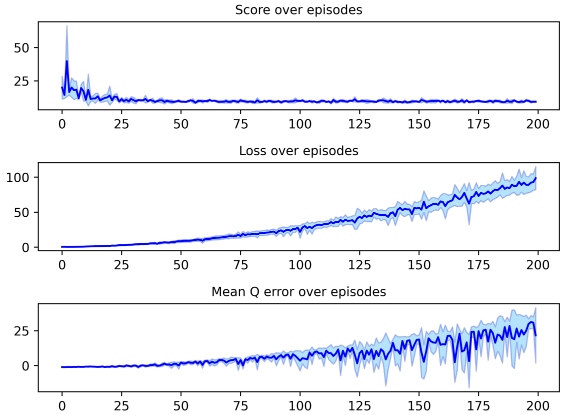

Figure 8: Without Replay Buffer and without update

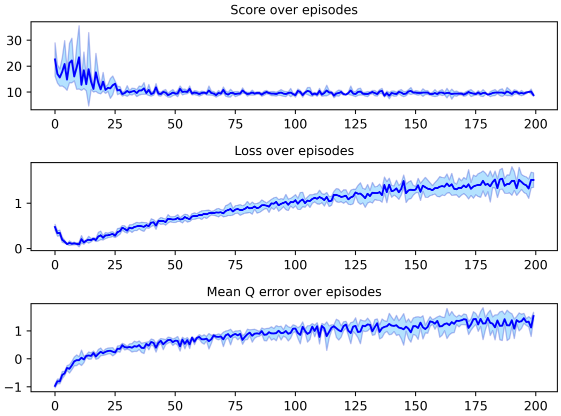

Figure 9: Without Replay Buffer and soft update for target network

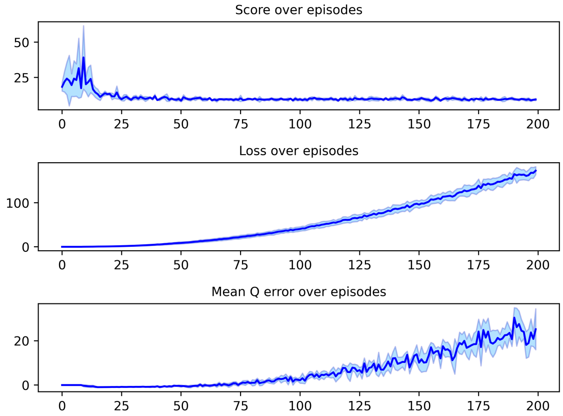

Figure 10: With Replay Buffer and without target network update

8 Question F

In this section, we have been asked to attach a movie of our best performing agent. which we trained with *soft target update* and *experience replay*, and achieved a final task score of **368**. It should be mentioned that the movie stopped recording after 200 steps even though the game is not over.