Deep Reinforcement Learning
Assignment 2: Deep Q-Network

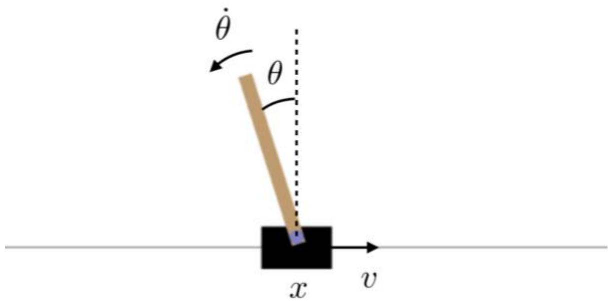
Lecturer: Armin Bies, Ph.D.

Due date: 31.8.21

Omer Luxembourg

<u>Preview</u>

In this exercise we are asked to solve a control problem with continuous state and discrete action space using DQN. The environment is given by the OpenAI Gym CartPole environment (https://gym.openai.com/envs/CartPole-v0/). The goal of CartPole is to balance a pole connected with one joint on top of a moving cart (see figure). The state is continuous and given by four real numbers s = [cart position, cart velocity, pole angle, pole angular velocity] (cart position ∈ [−4.8, 4.8], cart velocity ∈ [−∞, ∞], pole angle ∈ [−0.418, 0.418]rad and pole angular velocity ∈ [−∞, ∞]). The actions are discrete and given by 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 (i) the pole angle goes beyond ±12 degrees (±0.2094 rad), (ii) the cart position is larger than 2.4. The score is 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. For example, if the agent achieves in 10 episodes the following scores: 7, 9, 10, 20, 35, 15, 25, 5, 8, 8, then the task score is max[min[7, 9, 10, 20, 35], . . . , min[10, 20, 35, 15, 25], . . . min[15, 25, 5, 8, 8]]= 10.



a) In this part we completed the `train_model()` function given in `DQN.py`. This is exactly the DQN barebone implementation.

```
# 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'
# TODO: fill curr_Q - part a
# propagating the states through the network, then taking the value of picked actions
curr_Q = policy_net(state_batch).gather(1, action_batch)

# Compute expected_Q (target value) for all states.
# Don't forget that for terminal states we don't add the value of the next state.
# Pros tips: Calculate the values for all next states ( Q_(s', max_a(Q_(s')) )
#           and then mask next state's value with 0, where not_done is False (i.e., done).
# TODO: fill expected_Q - part a
if torch.cuda.is_available():
   mask = torch.cuda.FloatTensor(not_done_batch)
else:
   mask = torch.FloatTensor(not_done_batch)
expected_Q = reward_batch + \
          params.gamma * target_net(next_states_batch).max(dim=1).values * mask
#             gamma     *        (Q_(s', max_a(Q)')                *       Mask of            termination states
```

As we can see, we have two values of the current Q function for given state and action. We take actions by a policy network (which is parametrized by Θ) with ϵ-greedy method. After we take those actions, we evaluate the Q function from the network – saved as a variable in `curr_Q`.
In addition, we use a target network (Θ^-)to learn which is constructed from the policy network in several manners (soft, hard update, or just follows it every episode).
We created the variable `expected_Q` to save our Q update which is as follows:

$$y_j = \begin{cases} r_j \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \Theta^-) \end{cases}$$

Where the first term is for episodes that terminates at $j+1$.
*The mask value is to mask them out of the calculation of the additional term.

b) We created a replay buffer in the `ReplayBuffer` class in `buffer.py`. 2 function were implemented – **push**, to push new experiences into the buffer, which contains state, action, next state and reward (and done signal). **Sample**, to sample batch sized transition like the last randomly from the buffer.

```
def push(self, *args):
   """Saves a transition."""
   # TODO - part b
   # filling the buffer to capacity
   if len(self.memory) < self.capacity:
      self.memory.append(args)
   else:      # full buffer - run over old elements
```

```
        self.memory[self.position] = args
        self.position = (self.position + 1) % self.capacity

def sample(self, batch_size):
    # TODO - part b
    return random.sample(self.memory, min(len(self.memory), batch_size))        # to avoid bigger batch than buffer
```

Notice that inserting is done by a loop. Every time the buffer fills up, we are running over the oldest transition that are in the buffer (using `position` variable).

c) In this section we implemented the *hard target network update*. Each *x* episodes we will update our target network to be as the policy network. This is done by copying the weights of the policy network into the weights of the target network, correspondingly.

$$\Theta^{-} \leftarrow \Theta$$

```
# hard target update. Copying all weights and biases in DQN
if params.target_update == 'hard':
    # TODO: Implement hard target update - part c
    # Copy the weights from policy_net to target_net after every x episodes
    if i_episode % params.target_update_period == 0:    # every 15 episodes: theta' <- theta
        target_net.load_state_dict(policy_net.state_dict())
```

d) Implementing the *soft target network update*. Here after every update of the Q function, we update the target network in a dynamic way using hyperparameter tau (τ):

$$\Theta^{-} \leftarrow \tau\Theta + (1-\tau)\Theta^{-}$$

By updating this way, we get to 'remember' the past weights of the network.

```
# soft target update
if params.target_update == 'soft':
    # TODO: Implement soft target update - part d
    # theta' <- T * theta + (1-T) * theta'
    sd_policy = policy_net.state_dict()     # theta
    sd_target = target_net.state_dict()     # theta'
    # soft update for all the parameters
    for key in sd_policy:
        sd_target[key] = sd_policy[key] * params.tau + sd_target[key] * (1 - params.tau)
```

e) Effects of *Replay Buffer* and *Target Network*:
In this section we will compare DQN algorithm with different combinations of the two methods – replay buffer and target network (soft/hard).

The following table is with the given parameters (no optimization has done).
One experiment consists of 200 episodes, each of maximally 500 steps

```
network_params = {
    'state_dim': env.observation_space.shape[0],
    'action_dim': env.action_space.n,
    'hidden_dim': 64
}
```

```
training_params = {
    'batch_size': 256,
    'gamma': 0.95,
    'epsilon_start': 1.1,
    'epsilon_end': 0.05,
    'epsilon_decay': 0.95,
    'target_update': 'soft/hard',        # use 'soft' or 'hard'
    'tau': 0.01,                      # relevant for soft update
    'target_update_period': 15,         # relevant for hard update
    'grad_clip': 0.1,
}
```

For without target network, we changed the `target_update` to be 'hard' and the `target_update_period` to be 1.
For without replay, we changed `batch_size` to 1 and the replay buffer capacity to 1.

Target Update	**With** replay **With** target	**With** Replay Without target	Without replay **With** target	Without replay Without target
Hard	212.8	-	29.8	-
None	-	25	-	23.6
Soft	30.6	-	22.8	-

When we don't use a target network, we can observe that the mean Q error is increasing and the loss is increasing, no matter if we use replay buffer or not.

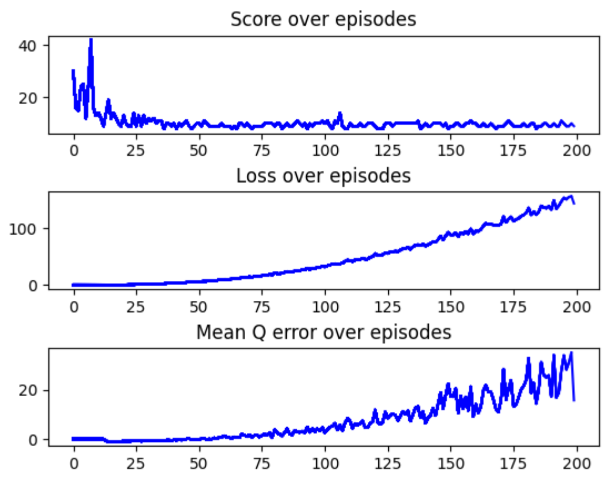


Figure 1 - No target network

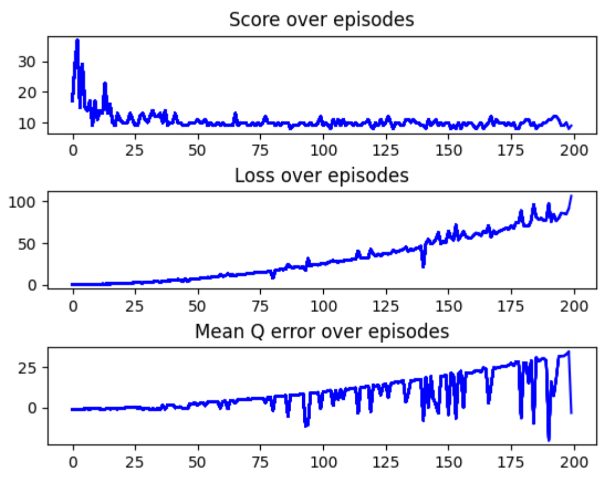


Figure 2 - No target network and no replay buffer

We can induce that when we don't have a constant network, we are practically 'chasing our tail' and can't converge to a certain value. This, as learned in class showing us that neural network needs a constant target to chase on.

When we don't use replay buffer, the loss and the mean Q error is rising, but not drastically as without target network.
For 'hard' target network update we'll get the following:

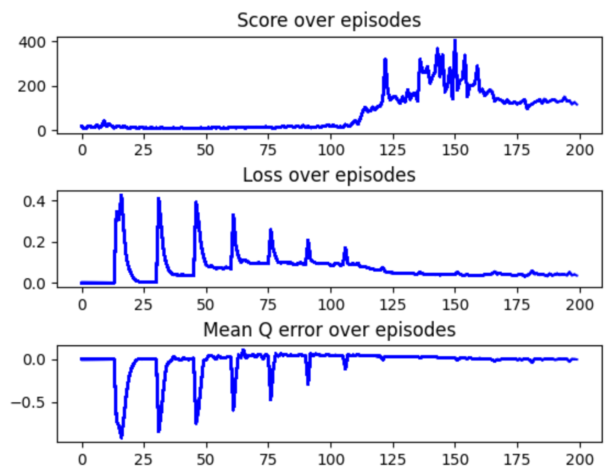


Figure 3 - Hard update

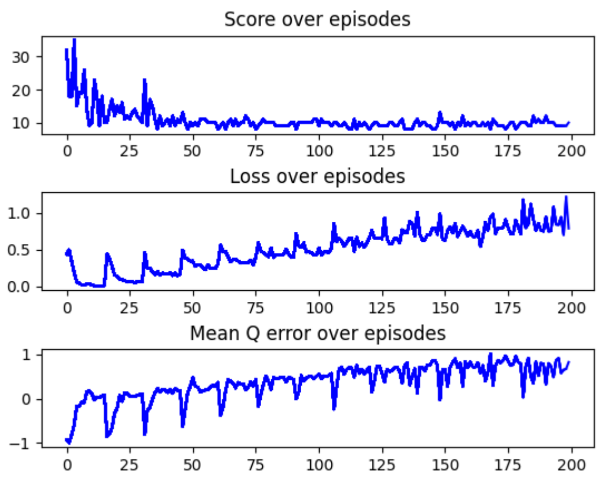


Figure 4 - Hard update and no replay buffer

We can conclude that taking i.i.d. samples from the replay buffer can help the network to learn. Correlated samples my harm our training (each batch the samples are alike

For ‘soft’ target network update, we’ll get a different result than ‘hard’:

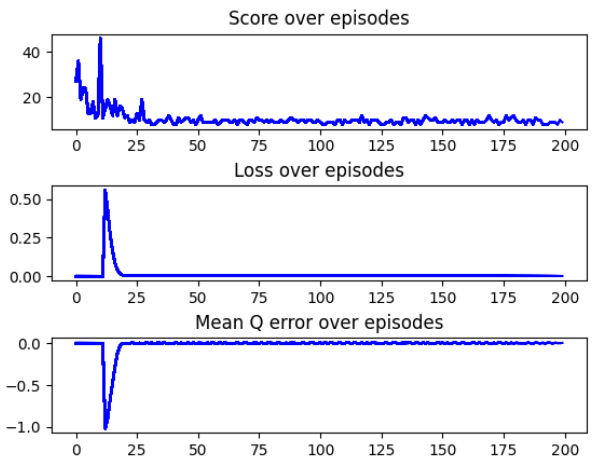


Figure 5 - Soft update

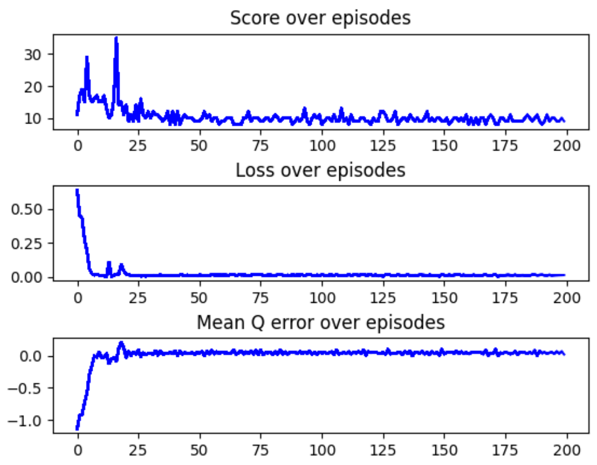


Figure 6 - Soft update no replay buffer

Note that with ‘soft’ target network update, with or without replay buffer is almost the same in terms of the loss over the episodes and the mean Q error. We can notice that the learning is quite jittery for the run without the replay buffer.
We can infer that when using ‘soft’ target update, taking non-i.i.d. samples will not harm the training. It might be because we ‘remember’ our last target network each time and weights that learned from previous correlated batch will take into account in our new network, in contrast to hard update which ‘forgets’ the past and learns from *new* correlated data which won’t be like the previous batches.

Our conclusions are that the replay buffer and the target network are critical for DQN algorithm and without them convergences to the right Q function will be hard or even impossible.

<u>Video of best performance - 501 score (attached with the PDF):</u>
The best task score is 501 from the ‘hard’ update of the target network and with replay buffer (taken from experiment *DQN-HARD-20210717-112844*).

Extra

To find the best parameters we've used the Optuna package in python for hyperparameters smart tunning. The optimization is on the hyperparameters we got in advance (without gamma) but remaining the replay buffer and target network to work ('soft' or 'hard' will be decided by the tunning).
We've run this test for 'hard' update and for 'soft' update separately. Our conclusions from the tunning is the following parameters, which gave the best results:

```
Best trial:
Score: 501.0
Params:
            hidden_dim: 154
            epsilon_end: 0.04333746376187501
            epsilon_decay: 0.2583378644061504
            grad_clip: 0.2445614718796103
            target_update_period: 7
            batch_size: 407
```

Running the DQN with those parameters gave 501 score.