Exercise X

Eugeniu Vezeteu - 886240
ELEC-E8125 - Reinforcement Learning

October 15, 2020

1 Task 1

On this task Q-learning with function approximation was used. Figure. 1 show the result of training using handcrafted feature vector $\phi(s) = [s, |s|]^T$

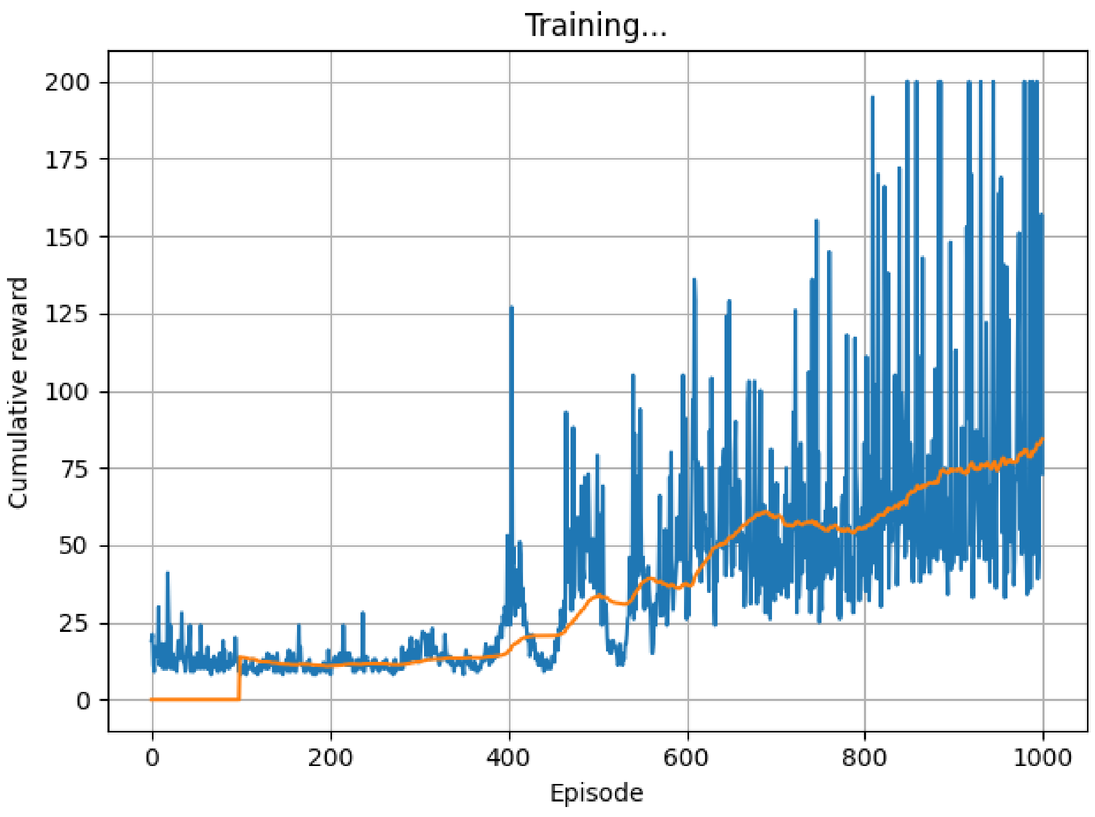

Figure 1: Task 1.a Q-learning with function approximation with handcrafted feature vector

The next code shows the implementation of Task 1.a

```
# Task 1a: TODO: Use (s, abs(s)) as features
featurized_state = np.concatenate((state, abs(state)), axis=1)
.
.
.
# Task 1: TODO: Set the feature state and feature next state
featurized_state, featurized_next_state = self.featurize(state)
    , self.featurize(next_state)
.
.
.
# Task 1:  TODO Get Q(s', a) for the next state
pred = np.array([q.predict(featurized_next_state)[0] for q in
    self.q_functions])
next_qs = np.amax(pred, axis=0)
.
.
.
# Task 1: TODO: Calculate target based on rewards and next_qs
if done:
    target = reward
else:
    target = reward+self.gamma*next_qs
```

Figure. 2 show the result of training using radial basis function.
The next code shows the changes made for Task 1.b

```
# Task 1b: RBF features
featurized_state =  self.featurizer.transform(self.scaler.
    transform(state))
```

2 Question 1 - Would it be possible to learn accurately Q-values

Well, here it depends how we define accurately, if we specify some tolerance parameter, the agent still can earn some reward (for example the agent has learned to balance in the cart pole environment). However, the trick here is to find the good kernel that will accumulate more information from the state space.
Now let's assume that we want the agent to learn the environment perfectly, it would not be possible to learn accurately Q-values of cart pole problem, using linear features directly, the agent will fail, because each feature from the state space will be treated equally (will have the same weight when taking decision), for example a high/low cart velocity may or may not be desired, depending on the other features values. Unexplored states may persist.

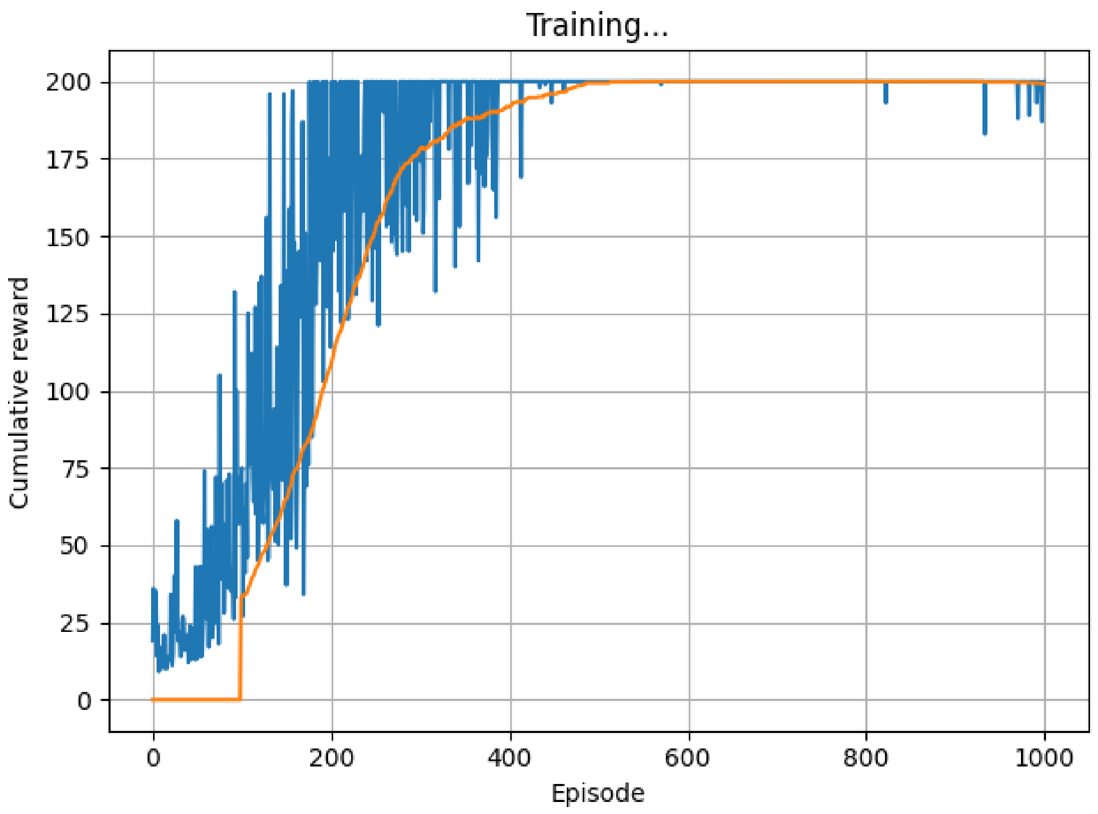

Figure 2: Task 1.b Q-learning with function approximation with radial basis function

3 Task2

On this task the algorithm was modified to perform minibatch updates.

Figure. 3 show the result of training using handcrafted feature vector $\phi(s) = [s, |s|]^T$ and minibatch updates

The next code shows the modification made to perform minibatch updates.

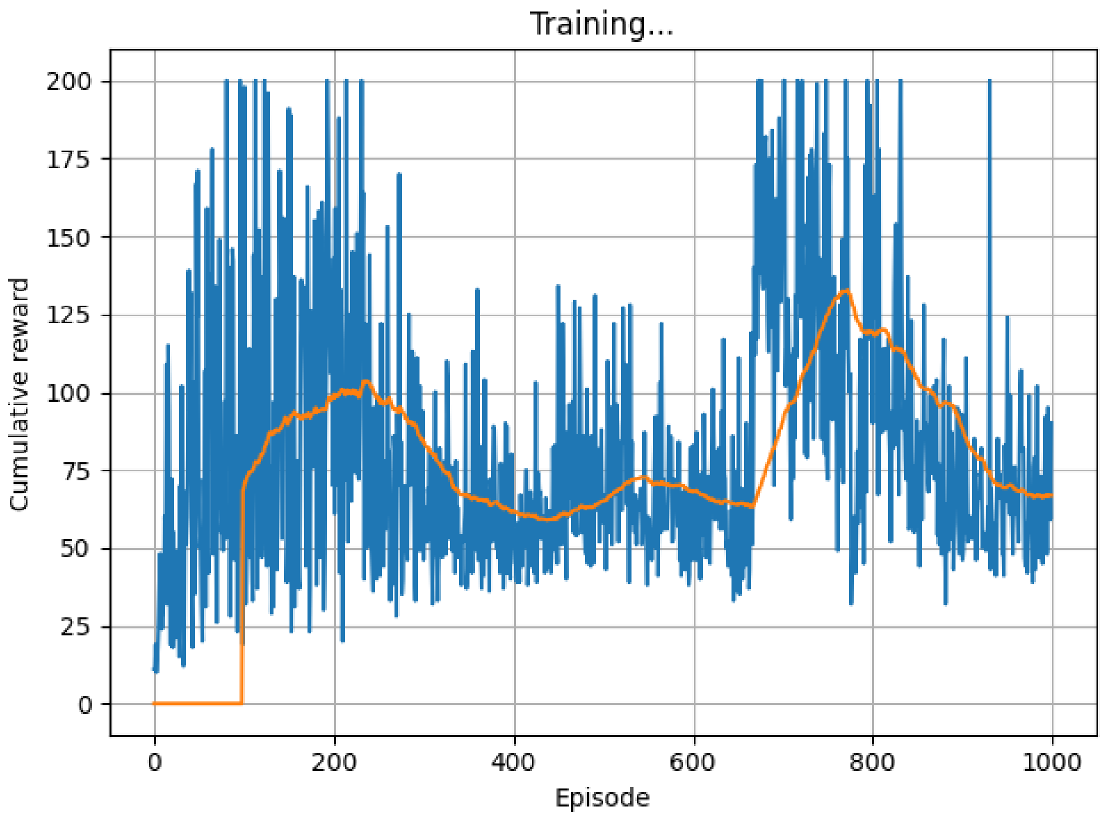

Figure 3: Task 2.a Q-learning with function approximation with handcrafted feature vector and minibatch updates

```python
# Task 2: TODO: Store transition and batch-update Q-values
agent.store_transition(state,action,next_state,reward,done)
agent.update_estimator()
.
.
.
# Task 2: TODO: Reformat data in the minibatch
states = np.array([sample.state for sample in samples])
action = np.array([sample.action for sample in samples])
next_states = np.array([sample.next_state for sample in samples
   ])
rewards = np.array([sample.reward for sample in samples])
dones = np.array([sample.done for sample in samples])
.
.
.
# Task 2: TODO: Calculate Q(s', a)
featurized_next_states = self.featurize(next_states)
next_qs = np.array([np.amax(np.array([q.predict(s_.reshape
    (1,-1))[0] for q in self.q_functions]),axis=0) for s_ in
    featurized_next_states])
```

```

# Task 2: TODO: Calculate target based on rewards and next_qs
#(1 - dones) will be zero when done is True
targets = rewards + self.gamma * next_qs * (1 - dones)
```

The same code was used for the second case, with radial basis function, result is presented in Figure. 4

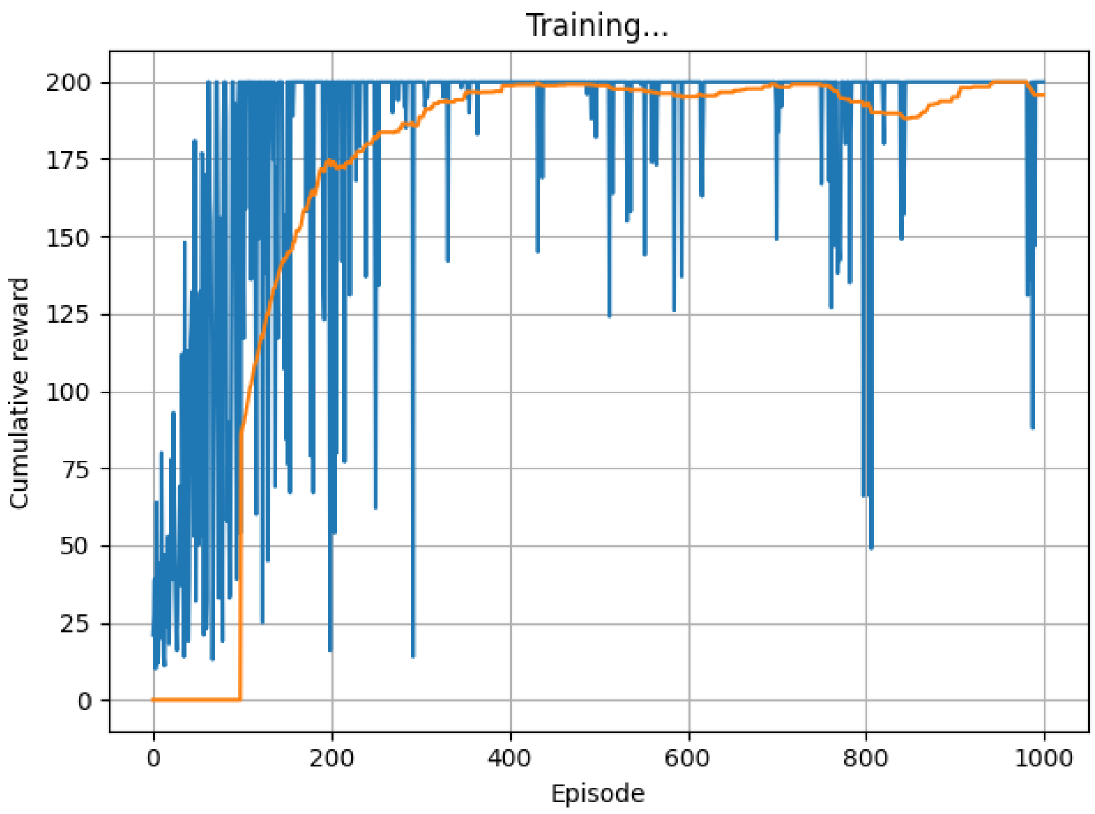

Figure 4: Task 2.b Q-learning with function approximation with radial basis function and minibatch updates

4 Question 2.1 - How does the experience replay affect the learning performance

Experience replay speeds up the training process, the policy under experience replay converges faster, due to that fact that the agent uses past (transitions stored in the memory) to learn,[1]. Learning from past batches is increasing the variance and decrease the bias, using memory ensures a better exploration.

5 Question 2.2 - benefits and cons of using hand-crafted feature

benefits

- *Cost* - reduced learning time, for some easier problems we may achieve the same result with handcrafted features, while saving time and memory.
- Choosing the neural network architecture or RBF kernels for some problems may be harder, then using a handcrafted features.
- We have control on what features the model may use.

cons

- *Performance* - the performance is lower, due to the fact that handcrafted features are limited to the of human choice, while some function approximation functions may find some hidden environmental behaviour (hidden relationship between feature states).
- For example using the handcrafted feature $\phi(s) = [s_x, s_{\dot{x}}, cos(s_\theta), sin(s_\theta), s_{\dot{\theta}}]$ may cause some problems doe to cosine and sin periodicity.
- Feature extraction done by NN or RBF outperform the handcrafted, due to the fact that their are learned/adapted to a specific task.

6 Question 2.3 - Do grid based methods look sample-efficient

According to the image ”Training process with different methods” given in the exercise sheet 4, the Grid-based method performs the worst, because, in the continuous state space problems, Grid-based method is not able to properly handle the state space behaviour, the state space discretization may be not capture enough the environment.In the same image presented in assignment 4, we notice that for the first 400 episodes, the Grid-based method obtained a higher reward in comparison with the handcrafted method, however, after 400 episodes, the handcrafted method obtained a better reward.
So, the Grid based do not look sample efficient compared to any of the function approximation method.

7 Task 3

On this task the best action in term of states was plotted, with $\dot{x} = \dot{\theta} = 0$. Result is presented in Figure. 5. The state space was discretized into 100 units.

The next code shows the modification made to perform minibatch updates.

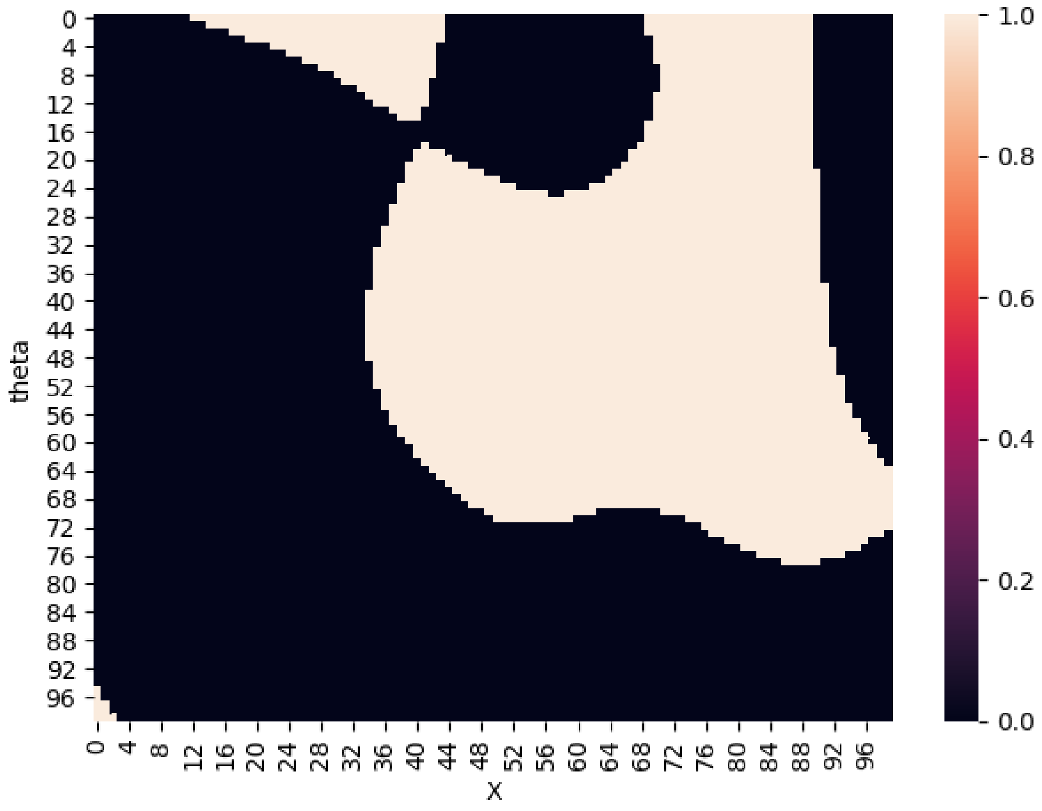

Figure 5: Task 3 - Policy with RBF with experience replay

```
# Task 3 - plot the policy
X = np.linspace(-2.4, 2.4, 100)
theta = np.linspace(-0.3, 0.3, 100)
policy = np.zeros((100, 100))

for i,x in enumerate(X):
    for j,t in enumerate(theta):
        s = np.array([x, 0, t, 0])
        a = agent.get_action(s)
        policy[i,j] = a

seaborn.heatmap(policy)
plt.xlabel("X")
plt.ylabel("theta")
plt.show()
```

Dark regions correspond to 0 (go left) action, 1 correspond to go right.

8 Task 4

In this task a basic DQN was used instead of RBF.

The next code shows the modification made to the code

```
# Task 4: Update the DQN
agent.store_transition(state, action, next_state, reward, done)
agent.update_network()
.
.
.
# Task 4: TODO: Compute the expected Q values
expected_state_action_values = reward_batch + self.gamma *
    next_state_values
```

The DQN was trained for both environments:
CartPole Figure. 6
LunarLander Figure. 7

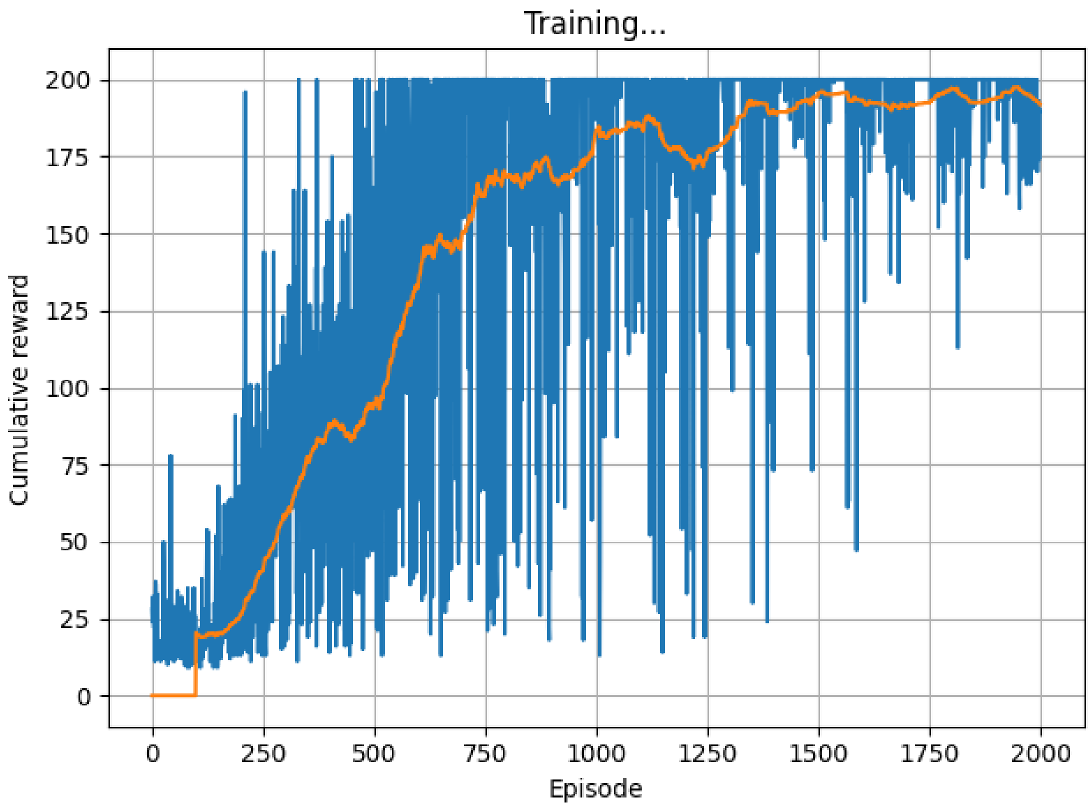

Figure 6: Task 4 DQN CartPole training results

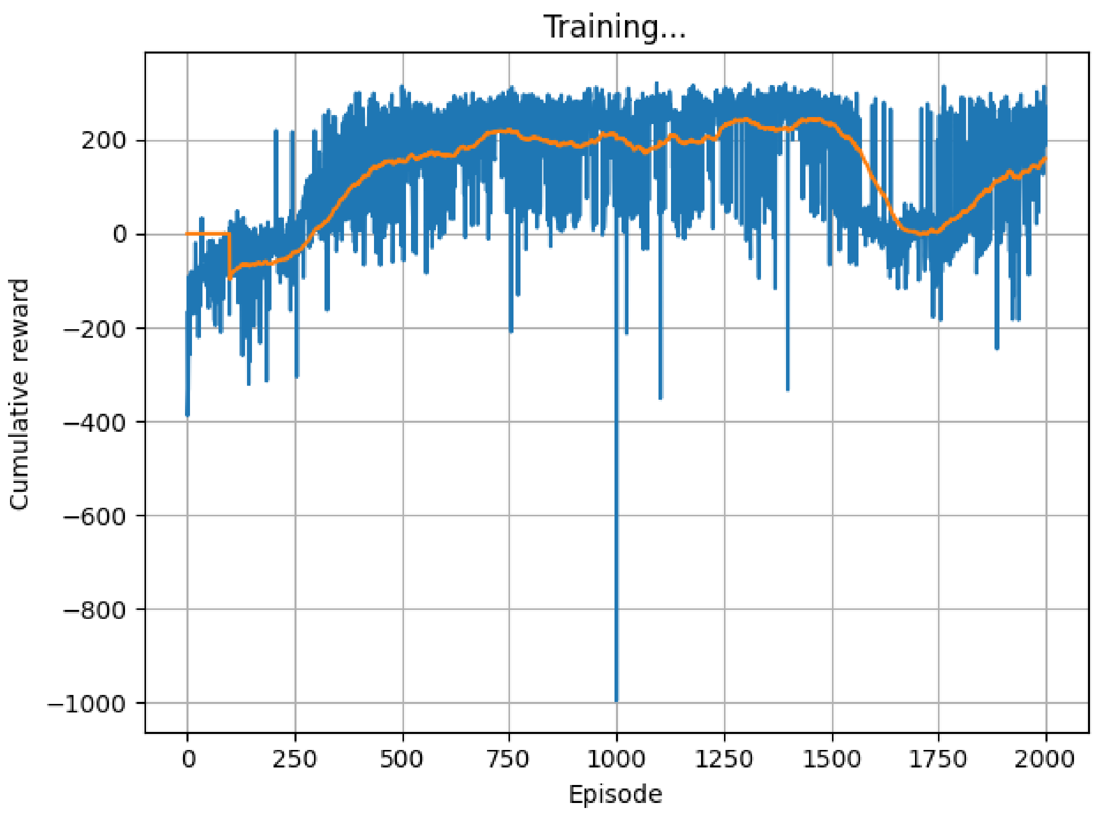

Figure 7: Task 4 DQN LunarLander training results

9 Question 3.1 - Can Q-learning be used directly in environments with continuous action spaces

YES, (a discretized version of Q-learning can be used in continuous action spaces environment). The discretization of the action space must be done under some assumptions. If the assumptions aren't realistic or the discretization is poor, Q-learning applied directly to continuous action space will fail.

10 Question 3.2 - Which steps of the algorithm would be difficult to compute, how to solve them?

The target state-action values are computed as a maximum state-action value over all actions.
This step is easy done with discrete action space, so the difficult step here is to choose the right continuous action space discretization.
To solve this problem we may try to search for a different action space discretization parameters, lets say the banal Line Search of a discretization parameter, based on agent performance. (This solution has very high time and memory complexity - since the agent must be trained

from scratch for each new parameter), also by discretization of the action space the number of actions increases exponentially with the number of degrees of freedom.

Also, maximizing the state-action value over a set of continuous actions requires computational time and power [2].
DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces, it cannot be straightforwardly applied to continuous domains since it relies on a finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step. [3],

To improve it we may use Actor-Critic, Deep Deterministic Policy Gradient.

References

[1] R. Liu and J. Zou, "The effects of memory replay in reinforcement learning," pp. 478–485, 10 2018.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[3] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2016.