Exercise 4

Tommaso Brumani - 100481325
ELEC-E8125 - Reinforcement Learning

October 10, 2022

Task 1

To implement Q learning using function approximation, the following code was used in the **'update()'** and **'get_action()'** functions of the **'rbf_agent.py'** file:

- get_action():

```
########## Your code starts here ##########

# determine whether to choose greedy policy
choose_rand = np.random.random() <= epsilon

if choose_rand:
    # take random action
    return np.random.randint(0, self.num_actions)
else:
    # featurize
    f_state = self.featurize(state)

    # take greedy action
    q_values = [func.predict(f_state) for func in self.
       q_functions]
    return np.argmax(q_values)

########## Your code ends here #########
```

- update():

```
########## You code starts here #########

# featurize the state and next_state
f_state = self.featurize(batch.state)
f_next_state = self.featurize(batch.next_state)
q_tar = np.zeros(self.batch_size)

# variable for storing maximum q for each batch element
max_q = np.zeros(self.batch_size)

# extract next state batch elements in which the
   terminating state is not reached
fns_not_done = f_next_state[batch.not_done.astype(bool), :]

# predict q value for each possible action
q_actions = np.array([func.predict(fns_not_done) for func
   in self.q_functions])

# set maximum q for non terminating batch elements
max_q[batch.not_done.astype(bool)] = np.max(q_actions, axis
   =0)

# calculate q_target
q_tar = batch.reward + self.gamma*max_q

########## You code ends here #########
```

The model was trained using two different features for state representation:

(a) A handcrafted feature vector $\Phi(s) = [s, |s|]^T$, the training performance plot for which is reported in Figure 1.

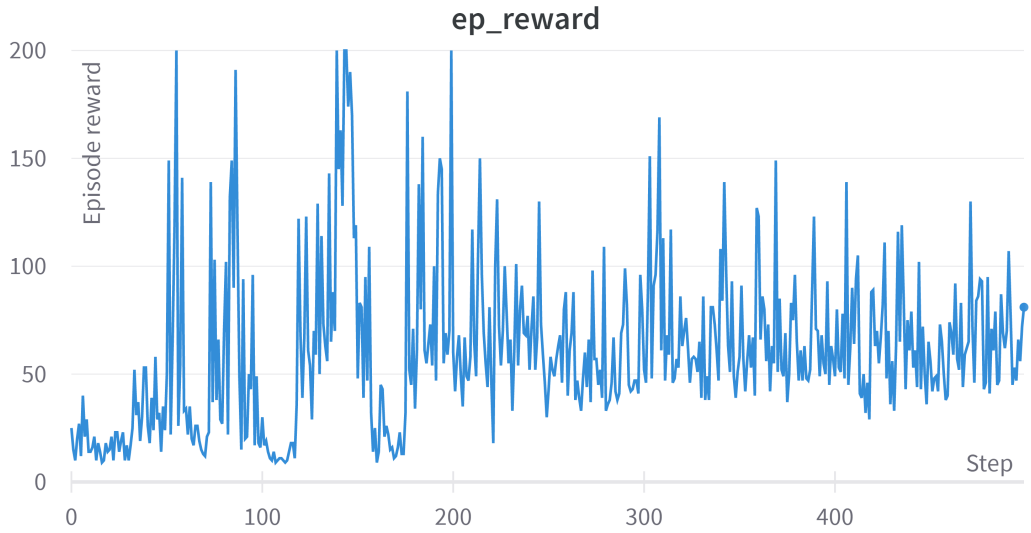

Figure 1: training performance using handcrafted feature vector.

(b) A radial basis function representation, the training performance plot of which is reported in Figure 2.

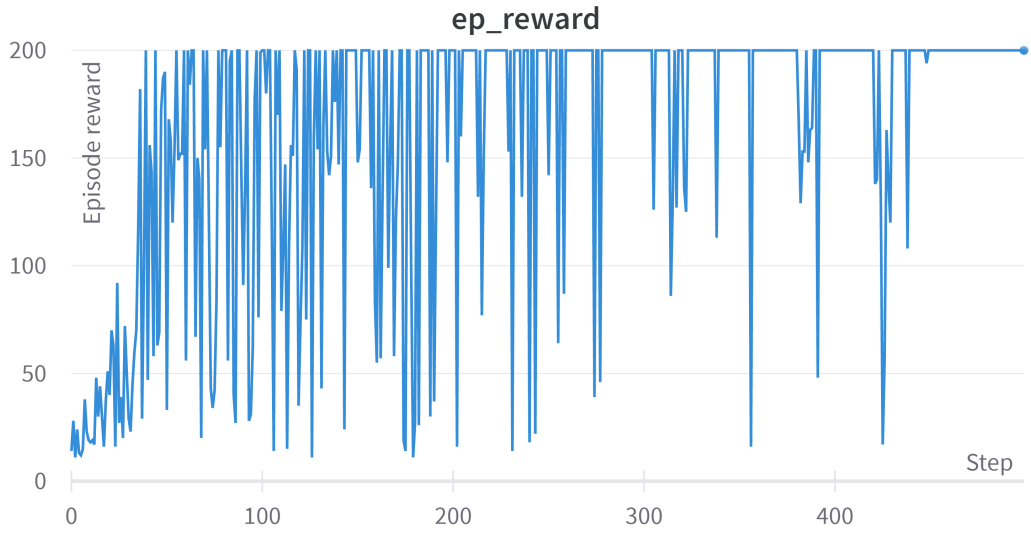

Figure 2: training performance using radial basis function representation.

Question 1.1

It would not be possible to accurately learn Q values for the Cartpole problem using linear features, since we can expect the real value function to take a symmetric shape around the center of the space (the cart should try to keep the pole going away from the two sides of the

screen), and such a function cannot be achieved using linear features with a linear regressor, which could only produce values proportional to the state.
It then becomes necessary to map the state to a space with a higher number of features, so that a linear regressor may find a fitting linear function over a higher number of dimensions, resulting in a non-linear function in the original state space.
From the fact that using the handcrafted feature vector $\Phi(s) = [s, |s|]^T$ still fails to converge to a good policy, we can deduce that the number of features in that instance is still insufficient to learn the problem.

Question 1.2

The use of an experience replay buffer makes it possible to learn from past experiences, which achieves a number of positive effects.
For one, it allows to increase sample efficiency by making it possible to reuse samples multiple times, and the ability to store samples in memory allows to take mini-batches to speed up computation.
Furthermore, it helps stabilize the learning process by breaking the temporal correlations of subsequent updates, reducing bias: by sampling the batches randomly from memory, the algorithm learns from samples that have low correlation and are thus more close to the assumption of being i.i.d.

Question 1.3

Tabular methods are generally sample-inefficient compared to RBF function approximation, as the number of states is usually greater than that of parameters used in function approximation.
Since modifying a single parameter has a cascading effect on the value of multiple states, function approximation makes it possible to train a model using a lower number of samples compared to tabular methods, as the variables to be learned are fewer.

Task 2

The 2D plot of policy learned with RBF in terms of x and θ for $\dot{x} = 0$ and $\dot{\theta} = 0$ is reported in Figure 3.

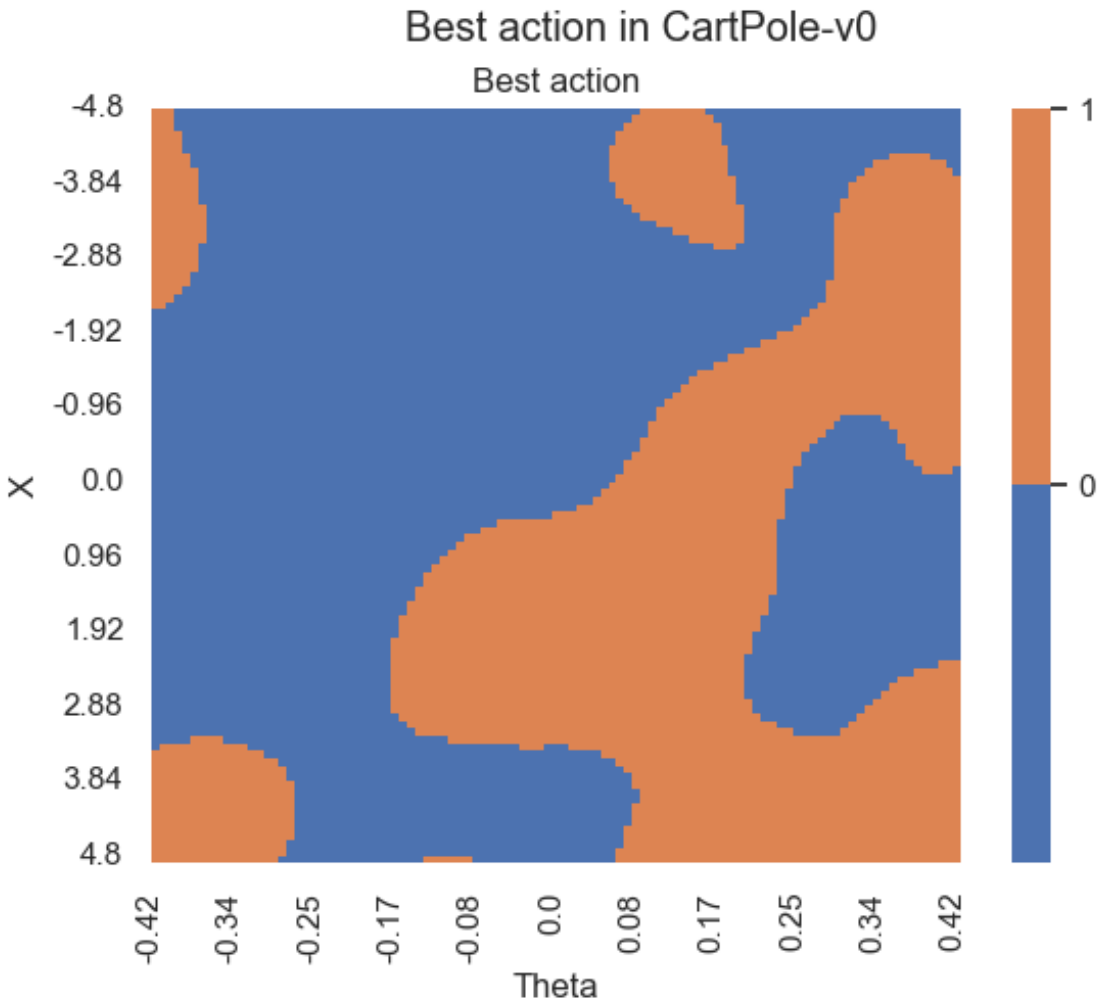

Figure 3: learned policy with RBF.

Task 3

To implement the DQN agent in the **'dqn_agent.py'** file, the code in the **'update()'** and **'get_action()'** functions were modified as follows:

- update():

```
########## You code starts here #########

# calculate the q(s,a)
q_actions_policy = self.policy_net(batch.state)
qs = torch.gather(input=q_actions_policy, dim=1, index=
    batch.action.type(torch.int64))

# calculate q target
q_actions_target = self.target_net(batch.next_state)
max_q = torch.max(input=q_actions_target, dim=1).values

# set to zero where final state is reached
max_q = torch.reshape(max_q, batch.not_done.size())*batch.
    not_done

# calculate q target
q_tar = batch.reward + self.gamma*max_q

#calculate the loss
loss = (qs - q_tar.detach())**2 * 0.5
loss = loss.sum()

# clip grad norm and perform the optimization step
self.optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(self.policy_net.parameters()
    , self.grad_clip_norm)
self.optimizer.step()

########## You code ends here #########
```

- get_action():

```
########## You code starts here #########

# determine wether to choose greedy policy
choose_rand = np.random.random() <= epsilon

if choose_rand:
    # take random action
    return np.random.randint(0, self.n_actions)
else:
    # take greedy action
    q_actions_target = self.policy_net(torch.from_numpy(
       state).to(device))
    return torch.argmax(q_actions_target).item()

########## You code ends here #########
```

The agent was trained in both the Cartpole and Lunar Lander environments, and its performance is presented in Figure 4 and Figure 5, respectively.

Figure 4: training performance of DQN in the Cartpole environment.

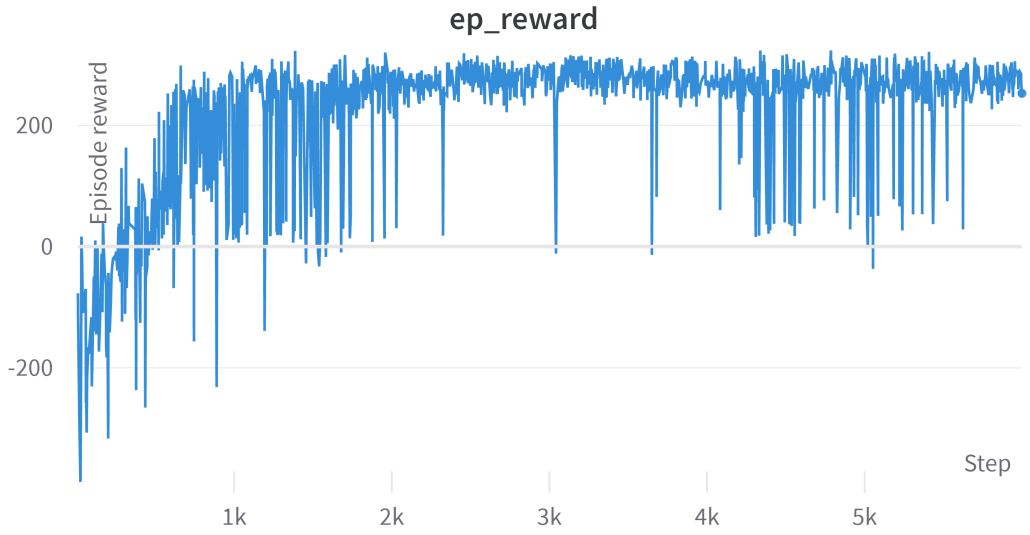

Figure 5: training performance of DQN in the Lunar Lander environment.

Question 3.1

Standard Q-learning cannot be used directly in continuous action space environments because discrete and finite actions and states are necessary for it to calculate the Q function for specific state-action pairs.

Question 3.2

The main difficulty when dealing with continuous action spaces is in calculating the maximum Q value over all actions for the next state. If the action space is continuous this problem becomes very complex, as simply evaluating the value over all possible actions is impossible. A possible (albeit imperfect) solution would be to simply discretize the action space.

Question 3.3

The use of an additional neural network is necessary to stop the bias derived from bootstrapping from causing the estimated Q values to diverge, for example in instances in which an outlier reward has been registered by chance. If a single neural network was used, for example, registering an abnormally high reward value during a transition could cause the algorithm to iterate over that estimate, thus it skewing its estimates.
Using a second network relatively fixed in time, instead, gives the algorithm the opportunity to correct this bias over multiple iterations by identifying its sharp divergence from previous estimates.
In this instance where a single network is used, not stopping the gradient of the target

Q value would result in it being taken into account and propagated when calculating the gradient, which might skew the update.