Exercise 3

Amr Alkhashab - 905833
ELEC-E8125 - Reinforcement Learning

October 17, 2020

1 Task 1

1.1 (a)

To use the handcrafted feature vector all what is need is to add the following code to featurizer;

```
featurized = np.concatenate((state,np.abs(state)), axis=1)
```

The result for such feature:

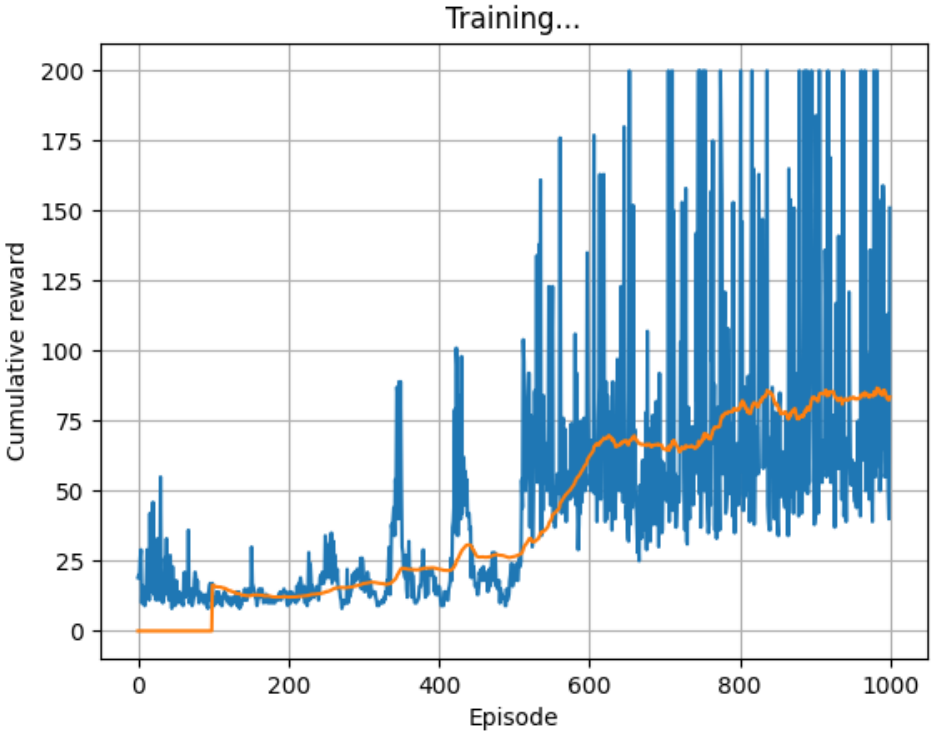

Figure 1: Handcrafted feature vector

1.2 (b)

For RBF

```
featurized = self.featurizer.transform(self.scaler.transform(
    state))
```

The results:

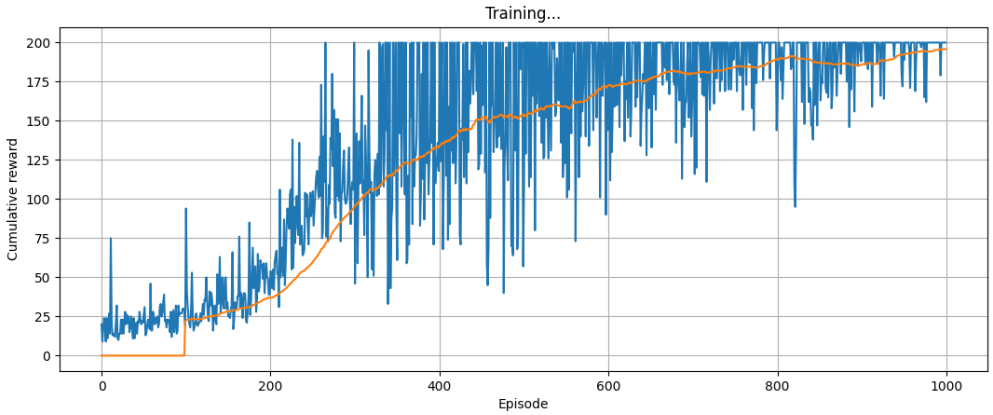

Figure 2: RBF feature vector

2 Question 1

No, it is not possible to learn Q-values accurately using linear features, and that can be shown in Figure. 3, because cartipole is not linear and Qvalues are mostly scattered with high variance across different states and approximating it will result in huge error and in most cases wrong representation for the values. The main objective for the selection of a function is to not only to be able to generilize but also to discriminize between states. It means low number of weight and less variation in the feature vector, which is then happens due to using state as feature vector, willl make discrimination between states qvalues more difficult and even impossible for some cases. A minimum number of weight and suitable representation for feature vector is needed to avoid that.

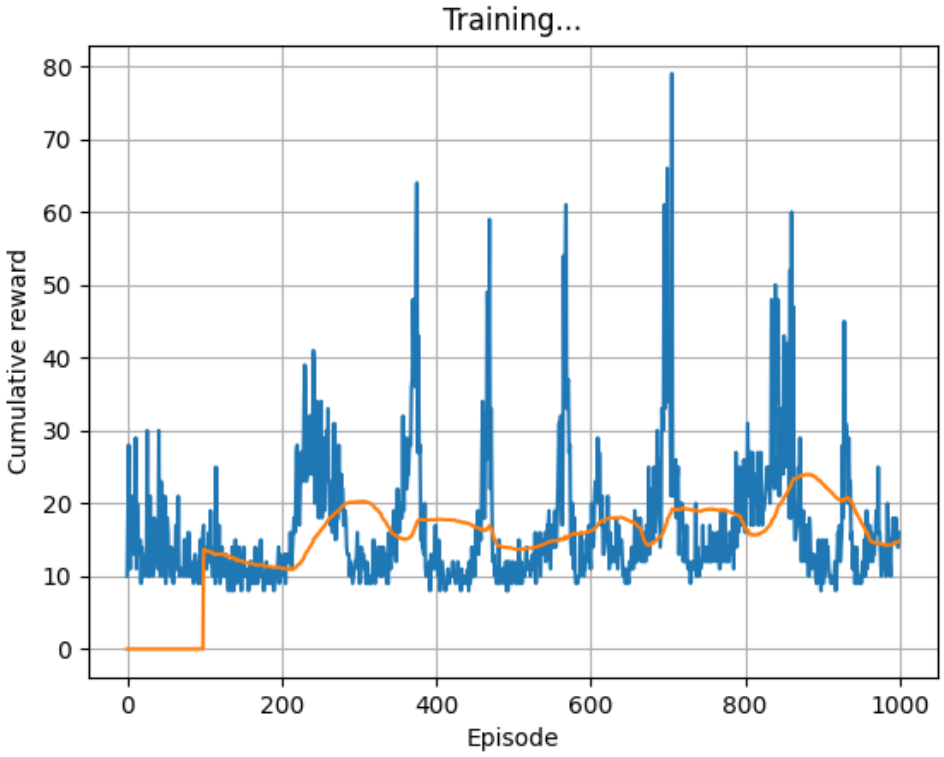

Figure 3: State feature vector

3 Task 2

```
        states = []
        action = []
        next_states = []
        rewards = []
        dones = []

#adding the batch stored data to number of lists
        for i in range(len(samples)):
            states.append(samples[i][0])
            action.append(samples[i][1])
            next_states.append(samples[i][2])
            rewards.append(samples[i][3])
            dones.append(samples[i][4])

        states = np.array(states)
        next_states = np.array(next_states)
        action = np.array(action, dtype=int)
#obtain the featurized vector for all states
        featurized_next_states = self.featurize(next_states)

        targets = []
        next_qs = np.zeros(1, )
 #used for loop to seperate Done (False and True) and all
    result is added to targets
        for i in range(len(samples)):
            # Task 2: TODO: Calculate Q(s', a)
            if dones[i] == False:
                qs = [q.predict([featurized_next_states[i]])[0]
                    for q in self.q_functions]
                qs = np.array(qs)
                next_qs[0] = np.max(qs, axis=0)
            else:
                next_qs[0] = 0
            # Calculate the updated target values
            # Task 2: TODO: Calculate target based on rewards
               and next_qs
            targets.append(rewards[i] + self.gamma * next_qs
               [0])

        # Calculate featurized states
        featurized_states = self.featurize(states)

        # Get new weights for each action separately
#change targets from list to numpy
        targets = np.array(targets)
```

The results:

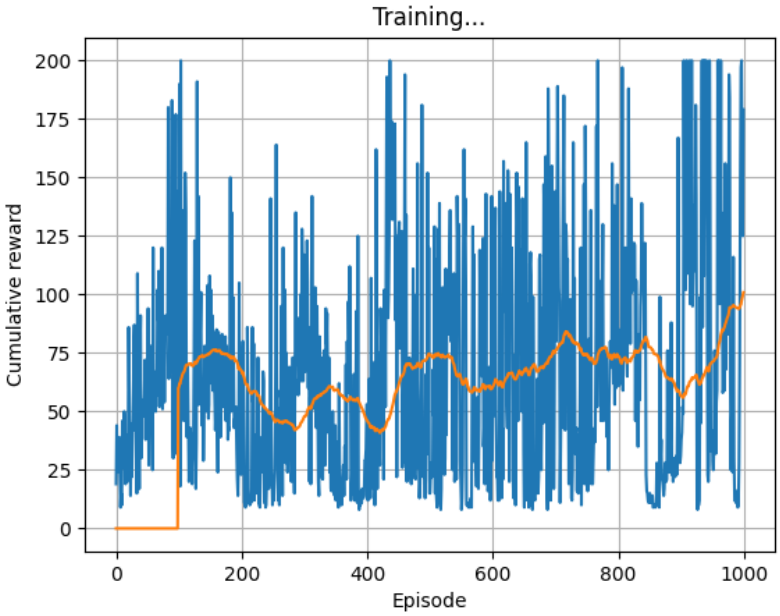

Figure 4: Handcrafted experience replay feature vector

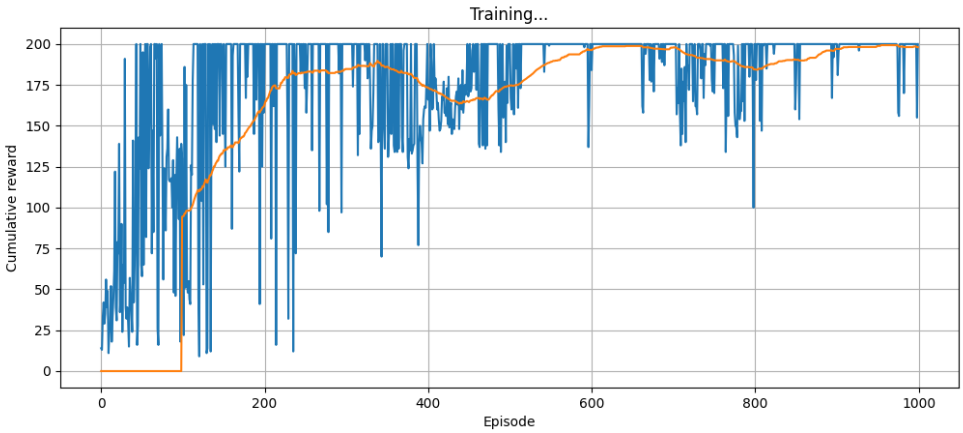

Figure 5: RBF experience replay feature vector

4 Question 2

4.1 Question 2.1

What experience replay does is [1],

1. Each step is store and used several times to update weights, that improves data efficiency and thus less episodes are needed for convergence as shown in figure.

2. Uses random samples rather than consecutive one. This breaks done the strong correlations between the samples and thus reduce variance in results.

3. By talking minibatch for experience replay the update is avearges over many samples, reducing undesired oscillations in the parameter and allow for smooth learning.

4.2 Question 2.2

The basic advantage are the ablility to optimize the performance. Adjusting the hand-crafted feacture so that it produce suitable results in the lowest training time. If the pattern of the results is known, which is not the case for most problems, then a hand-crafted feature vector representation would be suitable, for example if we want our agent to produce something like a sin, then containing a sin in the vector may actually improve and reduce uncertainity in our results, since it will produce more accurate representation of our values. One of cons are it is very difficult to obtain a feature vector that performs will, it needs a long cycle of trials, error and improvment to achieve desired results. This may complicate things since there is unlimited number of representation that is available. This become more and more difficult and even impossible as number of weight and representations increase. Using wrong representation can also degrade the performance, that is incase that the behavior is not sin for example.

4.3 Question 2.3

No, grid based method is not really that efficient when compared to approximation methods expecially compared to experience replay. In our case,that because we can observe that Grid-base was converging faster from hand-crafted one in the start but the exploration reduce rapidly due to small a value, so it preform actuactly worse and havenot converge before the others. Experience replay, however, has very high efficiency that is due to using the same date more than once, and that doesnot occur in the other cases.
From observation each model converged to optimal performace:

1. RBF experience converged at about 500

2. RBF about 700

3. Hand-carafted experience about 100

4. hand-carafted about 800

5. Grid based havnot converged during 1000 (noticable increase in value, even when epsilon was less than 0.1)

   Function approximation methods are said to be an approximation for the grid based method as it's name states. In function approximation, each sample is used to update the wieghts of the function that represents all states, while during grid only values of a single state is updated at a time. And in general, as number of states representation (grids) increase in grid method, so we can have more accurate estimation for states, the less sample-efficient the algorithm becomes, due to the fact the each state is updated individulally.

4.4 Task 3

This plot is different in each trial the model is trained. One conclusion is that the agent rarely achieve velocity and anglular that is of 0, so the weight are differents based on whose samples and data are obtained. The policy is determined based on the model estimation. 2D is drawn as a contour.

```
#grid structuring for contour
fig = plt.figure()
ax = fig.add_subplot(111)
#obs_limit = np.array([4.8, 5, 0.5, 5])
x = np.linspace(-4.8, 4.8, 100)
theta = np.linspace(-0.5, 0.5, 100)
a, b = np.meshgrid(x, theta)

v = np.zeros((1, 10000))
theta_dot = np.zeros((1, 10000))
positions = np.array([a.flatten()])
thetas = np.array([b.flatten()])

#forming array of all possible combination of states
States = np.concatenate((positions, v, thetas, theta_dot)).T
featurized = agent.featurize(States)

#calculating policy based on the discrete state values
#the limit is chosen between [4.8, 0, 0.5, 0] and [-4.8, 0,
   -0.5, 0]
policy = []
for i in range(10000):
    qs = [q.predict([featurized[i]])[0] for q in agent.
       q_functions]
    qs = np.array(qs)
    act = np.argmax(qs, axis=0)
    policy.append(act)

policy = np.array([policy])
policy = policy.reshape(100, 100)
ax.contourf(a,b,policy)
plt.xlabel('Position')
plt.ylabel('angle')
plt.show()
```

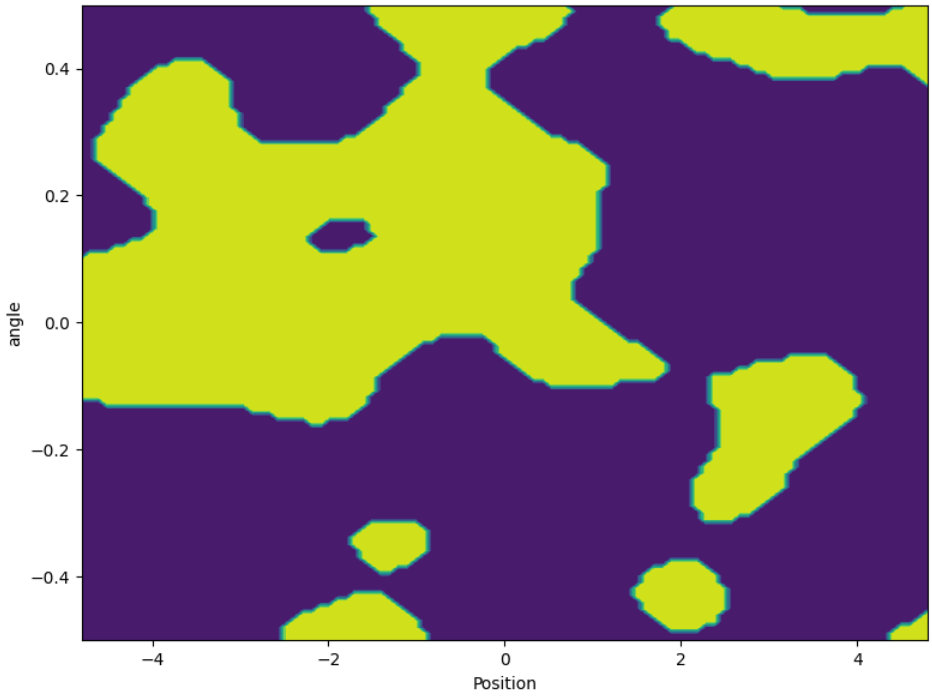

Figure 6: 2D policy

5 Task 4

```
1 expected_state_action_values = reward_batch + self.gamma *
    next_state_values
```

results:

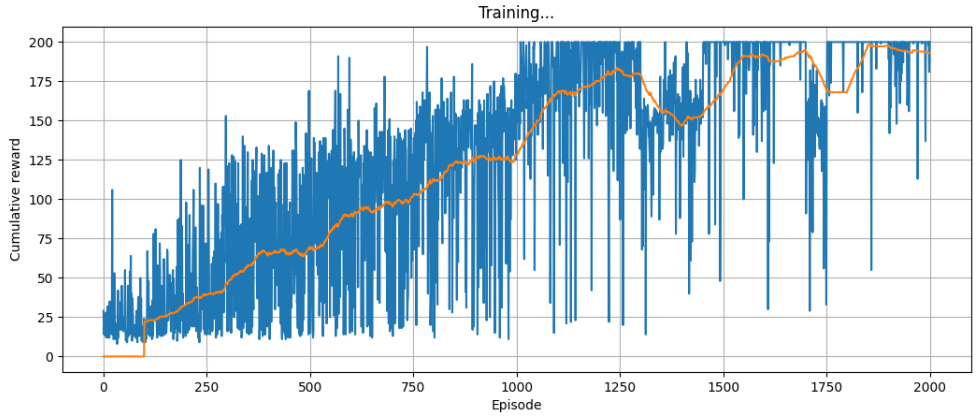

Figure 7: Cartpole

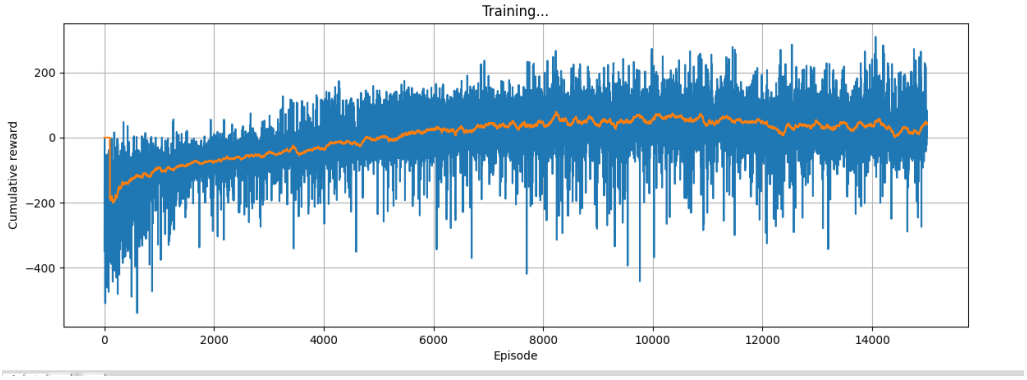

Figure 8: LunarLander

6 Question 3.1

It is possible to use Qlearning directly, provided that the action space is approximated to discrete form, using same method for states in exercise 3. However, in the case where there are two many actions and that actions is represented as function approximation then No.

7 Question 3.2

1. The step to find the action that corresponds to the highest action-state value finction maxQ(s,a) and argmaxQ(s,a). For maxQ, a new function apprioximation like critic for Actor-critic can be used, to evaluate the max action value. Discritizing the space and take small steps to find the argmaxQ can be one simple solution but maybe not so effiecient. Most approaches are headed toward using policy gradient or Actor-critic which doesnot relay on this maximising factor. So the idea is simply obtain a good approximation as we did for the states and thus we can turn it to optimization problem, trying to find the argmax by finding global optimal, a neural network can be used and it is implemented using input convex NN [2].

2. Previously we have an optimizer for each action, an each action had it's own approximation function. Using containous action, we now have a single function which takes a feature vector of not only states but also actions, to calculate the Qvalue.

References

[1] e. a. Mnih, Volodymyr, *Playing atari with deep reinforcement learning.* arXiv preprint-arXiv, 2013.

[2] J. Z. K. Brandon Amos, Lei Xu, *Input Convex Neural Networks.* Cornell university, 2017.