Exercise 4

Gengcong Yan - 1009903
ELEC-E8125 - Reinforcement Learning

October 22, 2021

1 Task 1

```
def featurize(self, state):
        if len(state.shape) == 1:
            state = state.reshape(1, -1)
        # Task 1a: TODO: Implement handcrafted features as (s, abs(s))
        #(x1, x2, x3, x4, abs(x1), abs(x2), ...)
        # return np.concatenate((state,abs(state)),axis=1)

        # # Task 1b: Use the RBF features using the featurizer
        return self.featurizer.transform(self.scaler.transform(state))

def get_action(self, state, epsilon=0.0):
    # TODO Task 1: Implement epsilon-greedy
    # Hint: See exercise 3
    if np.random.uniform(0,1)<epsilon:
        # random action
        return env.action_space.sample()
    else:
        # Choose current best action
        feas=self.featurize(state)
        qvalues=[ q.predict(feas)[0]  for q in self.q_functions ]
        re=np.argmax(np.array(qvalues))
        return re
```

```
def single_update(self, state, action, next_state, reward, done
    ):
        # Calculate feature representations of the
        # Task 1: TODO: Set the feature state and feature next
            state
        featurized_state = self.featurize(state)
        featurized_next_state = self.featurize(next_state)

        # Task 1:  TODO Get Q(s', a) for the next state
        qss= np.array([q.predict(featurized_next_state)[0] for
            q in self.q_functions])
        next_qs = np.amax(qss,axis=0)

        # Calculate the updated target Q- values
        # Task 1: TODO: Calculate target based on rewards and
            next_qs
        if done:  # No next state
            target=reward
        else:
            target = reward+self.gamma*next_qs
        self.q_functions[action].partial_fit(featurized_state,
            (target,))
```

The rewards with 2 different representations are shown in Fig.1 and Fig.2.

2 Question 1

NO, Cartpole using linear features failed to learn accurately Q-values. because a linear regressor may be too simple to represent the value approximation. Coaction between different features is not adequately expressed. the position of cartpole may vary differently based on different velocity, which leads to good or bad direction. We need some more complex expressions to reflect the relationship between the different features

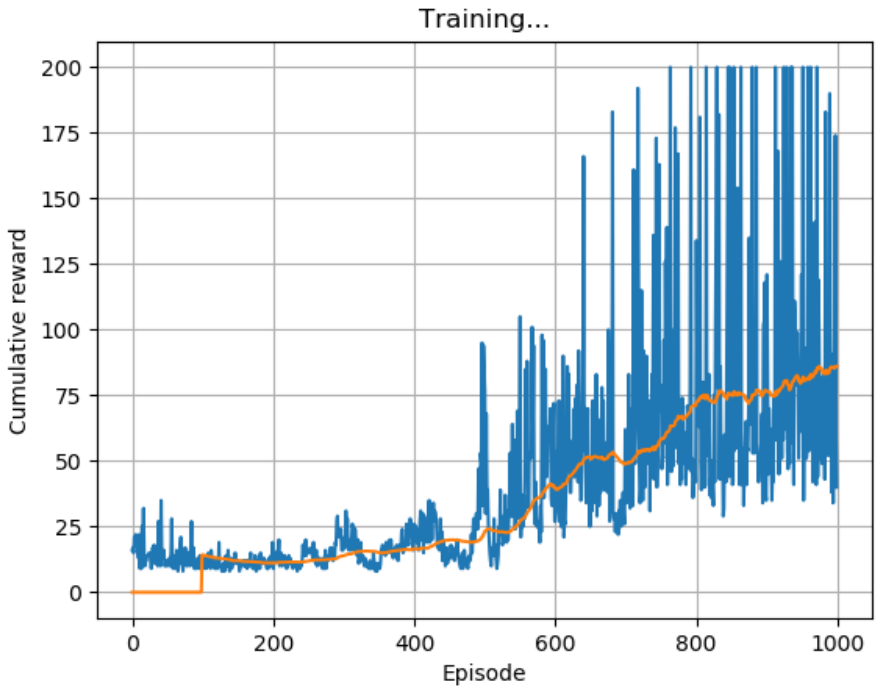

Figure 1: Rewards with handcrafted feature

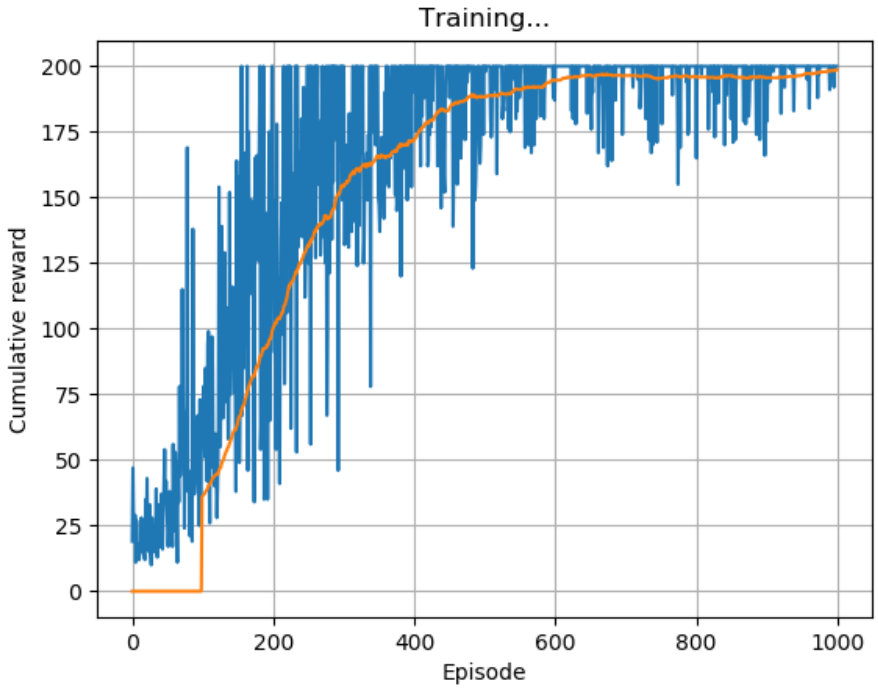

Figure 2: Rewards with RBF

3 Task 2

```
#### Code in function update_estimator, not all
 # Task 2: TODO: Reformat data in the minibatch
    states = []
    action = []
    next_states = []
    rewards = []
    dones = []
    for i in range(len(samples)):
        states.append(samples[i].state)
        action.append(samples[i].action)
        next_states.append(samples[i].next_state)
        rewards.append(samples[i].reward)
        dones.append(samples[i].done)
    states = np.array(states)
    action = np.array(action)
    next_states = np.array(next_states)
    rewards = np.array(rewards)
    dones = np.array(dones)

    # Task 2: TODO: Calculate Q(s', a)
    featurized_next_states = self.featurize(next_states)
    next_qs = []
    for fns in featurized_next_states:
        fns=fns.reshape(1,-1)
        next_qs.append(np.amax(np.array([q.predict(fns)[0] for
           q in self.q_functions]),axis=0))
    next_qs=np.array(next_qs)

    # Calculate the updated target values
    # Task 2: TODO: Calculate target based on rewards and
       next_qs

    targets = rewards+self.gamma*next_qs* (1-dones)
```

The rewards using minibatch update and experience replay are shown in Fig.3 and Fig.4.

4 Question 2

4.1 Question2.1

With experience replay, the training process became faster,the rewards increased a lot just in the beginning. This strategy will save computation power and converge faster, but ultimately the final performance won't get any better reward.

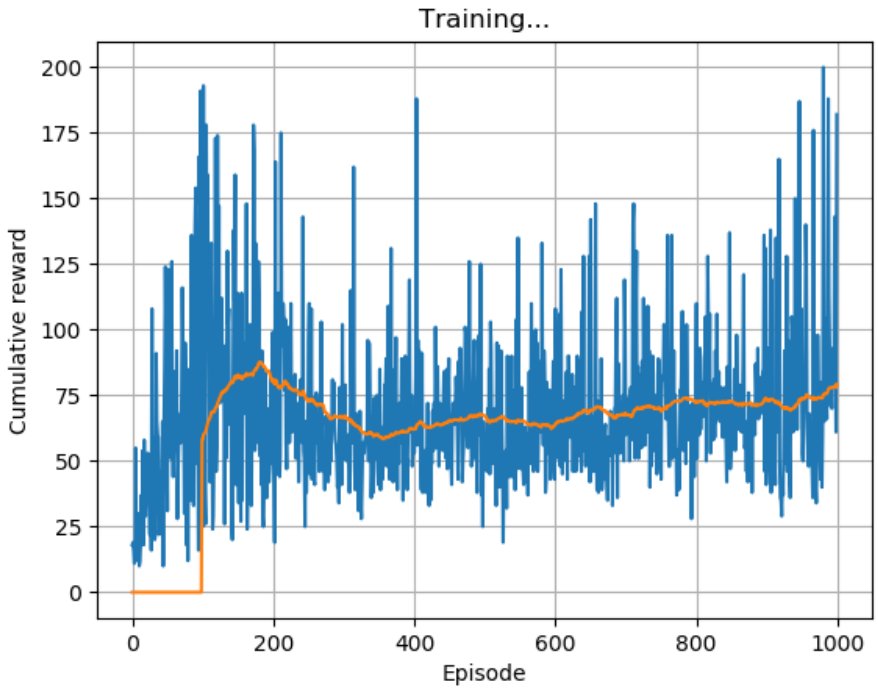

Figure 3: Rewards with handcrafted feature (experience replay)

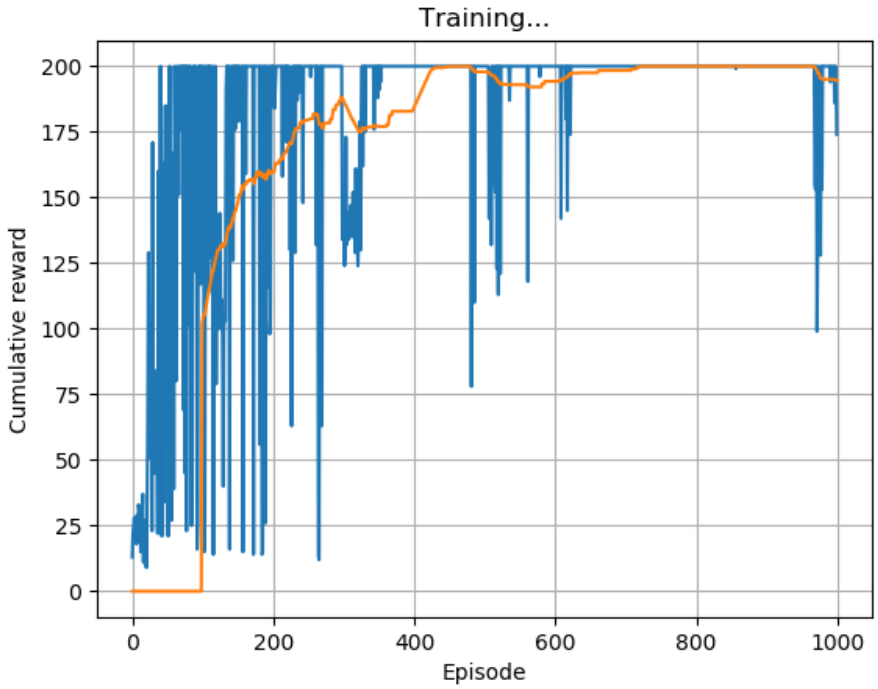

Figure 4: Rewards with RBF (experience replay)

4.2 Question2.2

- **Pros**: It saves computing power and train fast. It will work well if you can effectively construct representations of the important features in problem. But this is generally very difficult.
- **Cons**: It's too simple in some complex problems, resulting in bad performance. There is a lack of interaction between the different features.Their interconnection with each other can influence the training effect.

4.3 Question2.3

No, the grid based methods need much more episodes to train for convergence. grid base methods have to learn each behavior with huge episodes. The function approximation methods can generate different behaviors more efficiently.

5 Task 3

```
# There are also code in EX3 used here

def get_cell_index2(x,th):
    x = find_nearest(x_grid, x)
    # v = find_nearest(v_grid, state[1])
    th = find_nearest(th_grid, th)
    # av = find_nearest(av_grid, state[3])
    return x, th,

x_grid=x_grid.round(2)
th_grid=th_grid.round(2)

for i,x in enumerate(x_grid):
    for j,th in enumerate(th_grid):

        x,th=get_cell_index2(x,th)
        state=np.array([x,0,th,0])
        action=agent.get_action(state)
        q_grid[(i,j)]=action

# print(q_grid,x_grid,th_grid)

print("Ploting Policy with RBF experience replay")
sb.heatmap(q_grid,xticklabels=x_grid,yticklabels=th_grid,cbar=
   False)
plt.title("Policy with RBF experience replay")
plt.xlabel("X")
plt.ylabel("Theta")
plt.show()
```

Base on the code in Exercise 3, after some change we can plot the heatmap of policy learned with RBF and experience replay in Fig.5. Black and white represent 0 and 1 in plot, showing the current action in different situation. We can see theta of cartpole Basically no impact on action, but the position decides different reactions preventing cartpole falling down.

6 Task 4

The rewards in Cartpole and Lunar landing environment with DQN implementation are shown in Fig.6 and Fig.7.

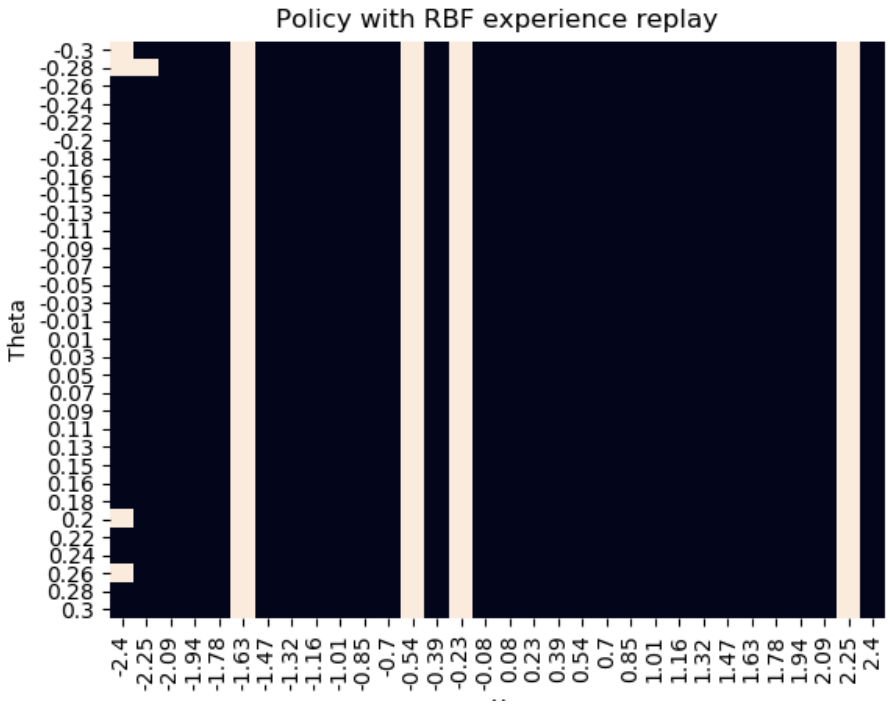

Figure 5: Heatmap of policy learned with RBF and experience replay

```
# get_action in DQN_agent
def get_action(self, state, epsilon=0.05):
        # TODO: Implement epsilon-greedy
        # Hint:
        # If greedy
        #   convert state to torch float tensor
        #   get q values from policy net (consider using with
           torch.no_grad()
        #   return action
        # If not greedy
        #   return random action
        if random.random()< epsilon:
            #random action
            return random.randrange(self.n_actions)
        else:
            # find best action
            with torch.no_grad():
                state = torch.from_numpy(state).float()
                qvalues=self.policy_net(state)
                # print("qvalues",qvalues)
                qmax_action=np.argmax(qvalues).item()
                # print("qmax_action",qmax_action)
                return qmax_action
```

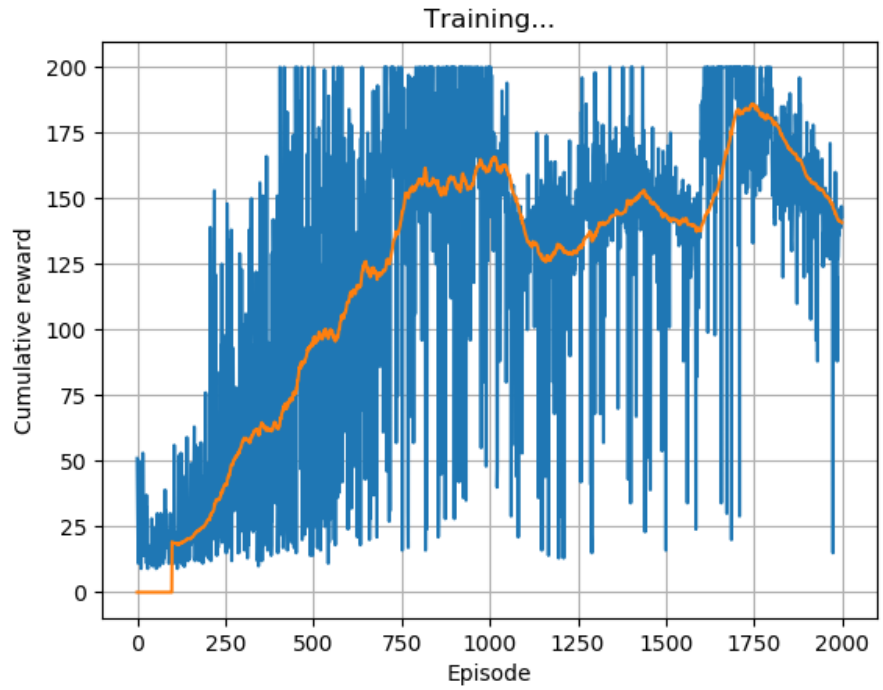

Figure 6: Rewards in Cartpole with DQN

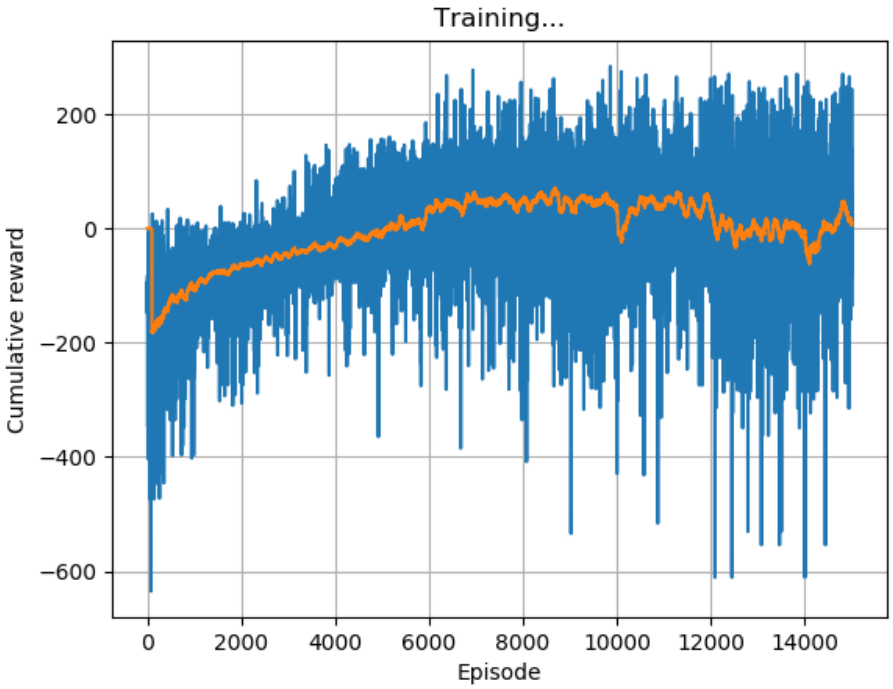

Figure 7: Rewards in Lunar landing with DQN

7 Question 3

7.1 Question 3.1

We **can not** directly use Q-learning in continuous action space. But with some mechanism like discretion and function approximation, we can still apply it in continuous space.

7.2 Question 3.2

In Q-learning, we always need to get $\max Q(S_{T+1}, a)$ for the next state. we need to iterate all actions to obtain maximum value of it. if the environment is a continuous action space, finding the maximum will be time and computation consuming. Even if we discretize the space we may not get a good result. So we can use advanced methods like policy gradient and Actor-Critic.

8 Feedback

1. 10h