Homework 3: Multi-Agent Search

110550168

Part I. Implementation (-5 if not explain in detail):

Please screenshot your code snippets of Part 1 ~ Part 3, and explain your implementation.

Part 1:

choose action

```python
def choose_action(self, state):
    """
    Choose the best action with given state and epsilon.

    Parameters:
        state: A representation of the current state of the enviornment.
        epsilon: Determines the explore/expliot rate of the agent.

    Returns:
        action: The action to be evaluated.
    """
    # Begin your code
    # TODO
    # raise NotImplementedError("Not implemented yet.")
    rand_num = np.random.uniform(1) # create a random number
    max_q = self.check_max_Q(state) # get the max Q of the state
    if(rand_num > self.epsilon): # if the random num > epsilon
        for i in range(6): # find the max Q's action and return the action
            if(self.qtable[state][i] == max_q):
                return i
    else: # if the random num < epsilon
        return np.random.randint(6,size = 1) # return random action
    # End your code
```

learn

```python
def learn(self, state, action, reward, next_state, done):
    """
    Calculate the new q-value base on the reward and state transformation observered after taking the action.

    Parameters:
        state: The state of the enviornment before taking the action.
        action: The exacuted action.
        reward: Obtained from the enviornment after taking the action.
        next_state: The state of the enviornment after taking the action.
        done: A boolean indicates whether the episode is done.

    Returns:
        None (Don't need to return anything)
    """
    # Begin your code
    # TODO
    self.qtable[state][action] = self.qtable[state][action] - self.learning_rate * (self.qtable[state][action] - (reward + self.gamma * self.check_max_Q(next_state)))
    # the q learning
    # End your code
```

check max q

```python
def check_max_Q(self, state):
    """
    - Implement the function calculating the max Q value of given state.
    - Check the max Q value of initial state

    Parameter:
        state: the state to be check.
    Return:
        max_q: the max Q value of given state
    """
    # Begin your code
    # TODO
    return max(self.qtable[state]) # return the max value of the state
    # End your code
```

Part 2:

init bins

```python
        # Begin your code
        # TODO
        # raise NotImplementedError("Not implemented yet.")
        nparr = np.linspace(lower_bound,upper_bound,num_bins,endpoint=False) # create bin with lower bound
        nparr = np.delete(nparr,0) # delete lower bound
        return nparr # return bin
        # End your code
```

discretize value

```python
        # Begin your code
        # TODO
        return np.digitize(value,bins,right=True) # get the discretize value of the value
        # End your code
```

discretize observation

```python
        # Begin your code
        # TODO
        re = []
        for i in range(len(self.bins)): # for all bins, discretize the value by a given observation
            re.append(self.discretize_value(observation[i],self.bins[i]))
        return re # return a list of discretize values
        # End your code
```

choose action

```python
        # Begin your code
        # TODO
        rand_num = np.random.uniform(0,1) # a random probability
        if(rand_num > self.epsilon): # if is larger then epsilon
            return np.argmax(self.qtable[tuple(state)]) # return the max q value's action
        else:
            return env.action_space.sample() # random choose a action
        # End your code
```

learn

```
# Begin your code
# TODO
if done: # if done, set max of next action's q to 0
    max_val = 0
else: # else, get the max of next action's q
    max_val = np.max(self.qtable[tuple(next_state)])

tmp = reward + self.gamma * max_val
self.qtable[(tuple(state)+(action,))] = self.qtable[(tuple(state)+(action,))] - self.learning_rate * (self.qtable[(tuple(state)+(action,))] - tmp)
# do q learning, with tuple as index
# End your code
```

check max q

```
# Begin your code
# TODO
state = self.discretize_observation(self.env.reset()) # get the initial condition's discretize value
return np.max(self.qtable[state[0]][state[1]][state[2]][state[3]]) # return the max q of the given observation
# End your code
```

Part 3:

learn

```
# Begin your code
# TODO
# raise NotImplementedError("Not implemented yet.")
observations, actions, rewards, next_observations, done = self.buffer.sample(self.batch_size) # get the sample data
state = torch.FloatTensor(np.array(observations)) # make it become a float tensor
act = torch.LongTensor(actions).unsqueeze(1) # make it become a long tensor, squeeze the data to right size
rwd = torch.FloatTensor(rewards) # make it become a float tensor
next_state = torch.FloatTensor(np.array(next_observations)) # make it become a float tensor
do = torch.BoolTensor(done) # make it become a bool tensor

val = torch.gather(self.evaluate_net(state),1,act) # get the tensor of the act
max_val = self.target_net(next_state).detach() * (~do).unsqueeze(-1) # calculate the next state's q max
tar = rwd.unsqueeze(-1) + self.gamma * max_val.max(1)[0].view(self.batch_size,1)
# calcculate the q at this state
loss = nn.MSELoss()(val,tar) # to calculate the loss, make it get close to real q

# calculate the new eval_net
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()

# End your code
```

choose action

```
# Begin your code
# TODO
rand_num = np.random.uniform(0,1) # a random probability
if(rand_num > self.epsilon): # if is larger then epsilon
    action = torch.argmax(self.evaluate_net(torch.FloatTensor(state))).item()
    # return the max q value's action
    # it is a tensor, change state to tensor and put it in evaluate net, then get the max q's index
else:
    action = env.action_space.sample() # random choose a action
# End your code
```

check max q

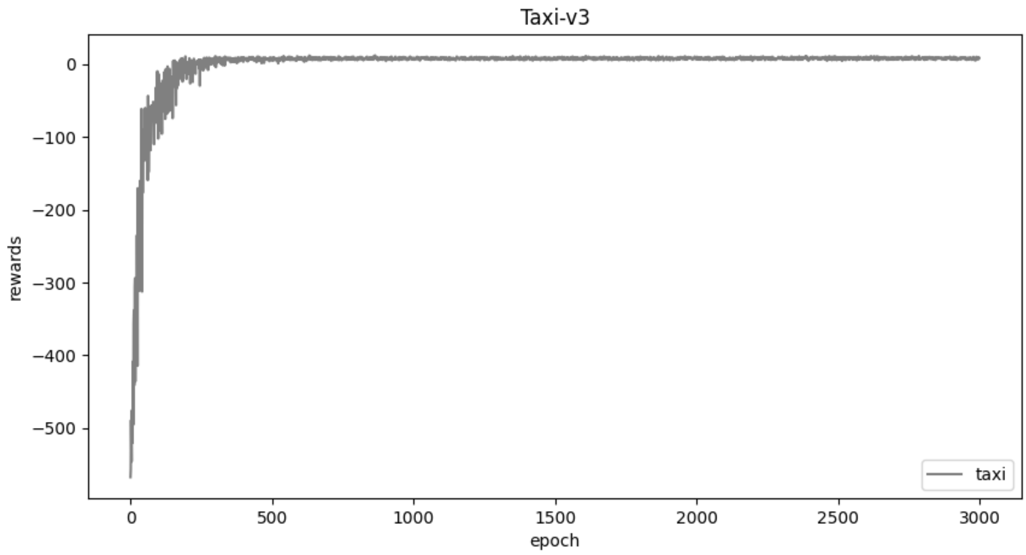
```
# Begin your code
# TODO
ini = torch.FloatTensor(self.env.reset()) # get the initial float tensor
state = ini.unsqueeze(0) # unsqueeze to right size
return torch.max(self.target_net(state)).item() # return the max q of the tensor

# End your code
```

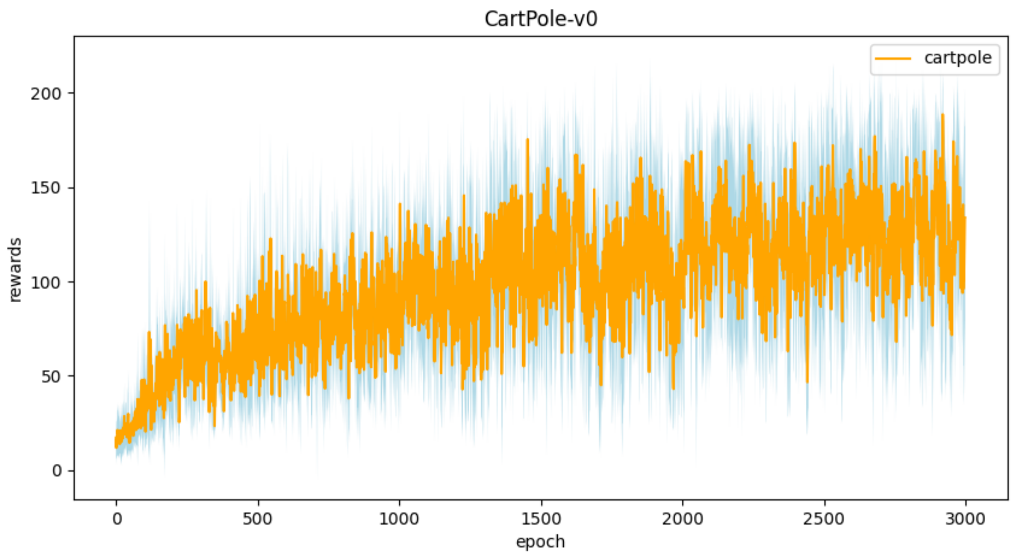
Part II. Experiment Results:

Please paste taxi.png, cartpole.png, DQN.png and compare.png here.

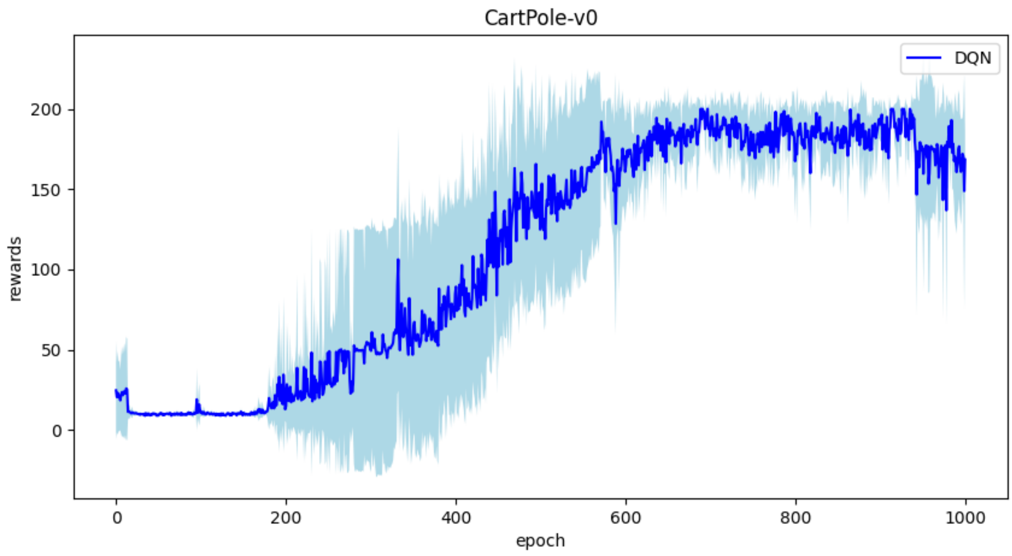
1. taxi.png:



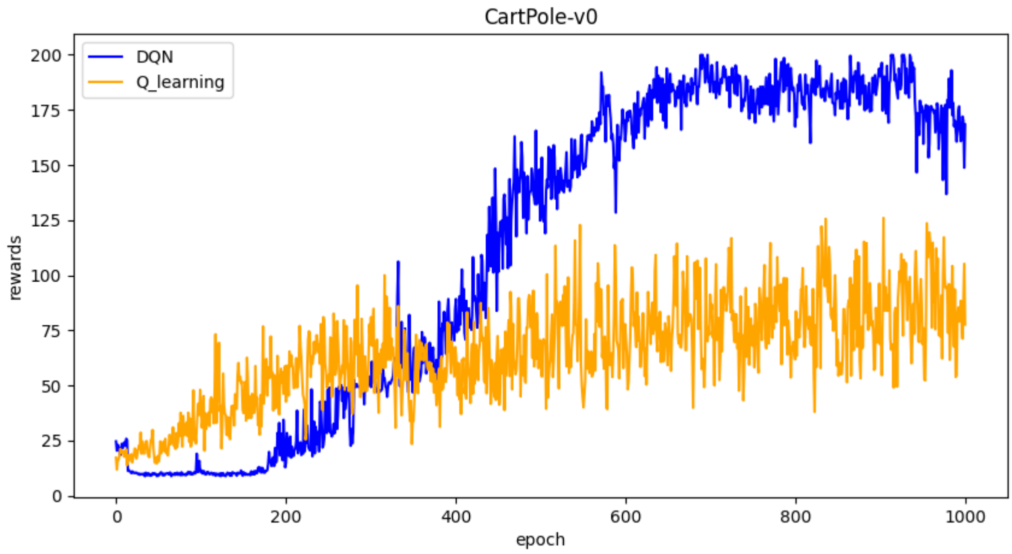
2. cartpole.png



3. DQN.png



4. compare.png



Part III. Question Answering (50%):

1. **Calculate the optimal Q-value of a given state in Taxi-v3, and compare with the Q-value you learned (Please screenshot the result of the "check_max_Q" function to show the Q-value you learned). (10%)**

```
#1 training progress
100%|████████████████████| 3000/3000 [00:45<00:00, 65.67it/s]
#2 training progress
100%|████████████████████| 3000/3000 [00:53<00:00, 55.57it/s]
#3 training progress
100%|████████████████████| 3000/3000 [00:50<00:00, 59.03it/s]
#4 training progress
100%|████████████████████| 3000/3000 [00:49<00:00, 60.75it/s]
#5 training progress
100%|████████████████████| 3000/3000 [00:45<00:00, 66.36it/s]
average reward: 8.07
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146700000021
```

```
print("opt q:",(-1)*(1-np.power(self.gamma,9))/(1-self.gamma)+20*np.power(self.gamma,9))
```

```
opt q: 1.6226146700000017
```

the optimal q by calculation is almost same to the max q because we just calculate it in the same way but different representation, and there is some error rate

2. **Calculate the max Q-value of the initial state in CartPole-v0, and compare with the Q-value you learned. (Please screenshot the result of the "check_max_Q" function to show the Q-value you learned) (10%)**

Cartpole :

```
#1 training progress
100%|████████████████████| 3000/3000 [03:42<00:00, 13.51it/s]
#2 training progress
100%|████████████████████| 3000/3000 [03:48<00:00, 13.11it/s]
#3 training progress
100%|████████████████████| 3000/3000 [03:33<00:00, 14.03it/s]
#4 training progress
100%|████████████████████| 3000/3000 [04:18<00:00, 11.59it/s]
#5 training progress
100%|████████████████████| 3000/3000 [04:10<00:00, 11.99it/s]
average reward: 108.01
opt q: 33.21430521048692
max Q:30.539265174855622
```

```
print("opt q:",(1-np.power(self.gamma,185))/(1-self.gamma))
```

the optimal q by calculation is slightly higher than the max q because we will explore when doing training, but direct calculation didn't

DQN :

```
#1 training progress
100%|████████████████████| 1000/1000 [08:15<00:00,  2.02it/s]
#2 training progress
100%|████████████████████| 1000/1000 [08:36<00:00,  1.94it/s]
#3 training progress
100%|████████████████████| 1000/1000 [07:36<00:00,  2.19it/s]
#4 training progress
100%|████████████████████| 1000/1000 [08:08<00:00,  2.05it/s]
#5 training progress
100%|████████████████████| 1000/1000 [08:40<00:00,  1.92it/s]
reward: 200.0
opt q: 33.25795863300011
max Q:33.2635612487793
```

```
print("opt q:",(1-np.power(self.gamma,200))/(1-self.gamma))
```

the max q is close to the manipulate calculated q, because the data I input is similar to the reward, and it shows that the DQN is close to the real q value

3.

a. Why do we need to discretize the observation in Part 2? (3%)

ans : We discretize the observation in order to divide continuous state space into a finite number of discrete states, so that we can create a Q table.

b. How do you expect the performance will be if we increase "num_bins"? (3%)

ans : I think that the result will become more accurate, since we have get close to the real data.

c. Is there any concern if we increase "num_bins"? (3%)

ans : The computing time will grow a lot, since the state will grow if we increase the number of bins.

4. Which model (DQN, discretized Q learning) performs better in Cartpole-v0, and what are the reasons? (5%)

ans : DQN performs better in Cartpole-v0, because DQN is better at handling high dimension state space, but if using discretized Q learning, the computing is extremely high since we need to have a super large Q table.

5.

a. What is the purpose of using the epsilon greedy algorithm while choosing an action? (3%)

ans : We use the epsilon greedy algorithm so that we can explore some new actions that are better than the old actions, but at the same time explicit some old good actions.

b. What will happen, if we don't use the epsilon greedy algorithm in the CartPole-v0 environment? (3%)

ans : If we don't use the epsilon greedy algorithm, we may not explore potentially better actions.

c. Is it possible to achieve the same performance without the epsilon greedy algorithm in the CartPole-v0 environment? Why or Why not? (3%)

ans : No, since we may not find potentially better actions, the performance will become less. Unless we use other algorithms to replace the epsilon greedy algorithm.

d. Why don't we need the epsilon greedy algorithm during the testing section? (3%)

ans : The agent has already learned the optimal policy. When testing, we don't need to explore new potentially better actions.

6. Why does "`with torch.no_grad():`" do inside the "choose_action" function in DQN? (4%)

ans : We use it to let the calculation not have gradient, so that we can speed up the computation and have a better efficiency.