Homework 4:

Reinforcement Learning

Report Template

Please keep the title of each section and delete examples. Note that please keep the questions listed in Part III.

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

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

1.taxi.py

we first use random to determine the action will choosed
if the random number less than self.epsilon, we choose the action by random in the possible action
ifthe random number greater than self.epsilon, we choose the action with maximun value in qtable

```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
        epsilon = np.random.rand()
        if epsilon < self.epsilon:
            action = self.env.action_space.sample()
        else:
            action = np.argmax(self.qtable[state])
        return action
        raise NotImplementedError("Not implemented yet.")
        # End your code
```

from the formula of Q-learning we update the qtable by the equation.

$$\underbrace{newQ_{S,A}}_{\text{基于状态和行动的新Q值}} = \underbrace{Q_{S,A}}_{\text{c当前Q值}} + \underbrace{\alpha}_{\text{学习效率}} \left(\underbrace{R_{S,A}}_{\text{基于状态和行动的奖励}} + \underbrace{\gamma}_{\text{折扣因子}} * \max \underbrace{Q'\left(s', a'\right)}_{\text{在给定新的状态和行动下未来最大的奖励}} - Q_{S,A} \right)$$

```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
    max_val = np.max(self.qtable[next_state])
    self.qtable[state,action] = self.qtable[state,action] + self.learning_rate*(reward+self.gamma*max_val-self.qtable[state,action])
    if done:
        np.save("./Tables/taxi_table.npy", self.qtable)
    # End your code
```

just return the maxvalue in the specific state in q-table.

```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 np.max(self.qtable[state])
    raise NotImplementedError("Not implemented yet.")
    # End your code
```

2.cartpole.py

we use function linspace to separate the upper bound and lower bound into n interval, but

we don’ t the first and last element, so we eliminate the first and last element with

ans[1:-1]

```python
    def init_bins(self, lower_bound, upper_bound, num_bins):
        """
        Slice the interval into #num_bins parts.
        Parameters:
            lower_bound: The lower bound of the interval.
            upper_bound: The upper bound of the interval.
            num_bins: Number of parts to be sliced.
        Returns:
            a numpy array of #num_bins - 1 quantiles.
        Example:
            Let's say that we want to slice [0, 10] into five parts,
            that means we need 4 quantiles that divide [0, 10].
            Thus the return of init_bins(0, 10, 5) should be [2. 4. 6. 8.].
        Hints:
            1. This can be done with a numpy function.
        """
        # Begin your code
        # TODO
        ans = np.linspace(lower_bound,upper_bound,num_bins+1)
        ans = ans[1:-1];
        return ans
        #raise NotImplementedError("Not implemented yet.")
        # End your code
```

we can easily use function digitize in numpy to calaulate the target value in which interval.

```python
    def discretize_value(self, value, bins):
        """
        Discretize the value with given bins.
        Parameters:
            value: The value to be discretized.
            bins: A numpy array of quantiles
        returns:
            The discretized value.
        Example:
            With given bins [2. 4. 6. 8.] and "5" being the value we're going to discretize.
            The return value of discretize_value(5, [2. 4. 6. 8.]) should be 2, since 4 <= 5 < 6 where [4, 6) is the 3rd bin.
        Hints:
            1. This can be done with a numpy function.
        """
        # Begin your code
        # TODO
        ans = np.digitize(value,bins)
        return ans
        raise NotImplementedError("Not implemented yet.")
        # End your code
```

when we get the observation value, we need to turn it into discrete value. So for every value in observation, we use the discretize function we write before to turn it into discrete value. 1

```python
    def discretize_observation(self, observation):
        """
        Discretize the observation which we observed from a continuous state space.
        Parameters:
            observation: The observation to be discretized, which is a list of 4 features:
                1. cart position.
                2. cart velocity.
                3. pole angle.
                4. tip velocity.
        Returns:
            state: A list of 4 discretized features which represents the state.
        Hints:
            1. All 4 features are in continuous space.
            2. You need to implement discretize_value() and init_bins() first
            3. You might find something useful in Agent.__init__()
        """
        # Begin your code
        # TODO
        state = []
        n = len(observation)
        #print(n)
        for i in range(0,n):
            state.append(self.discretize_value(observation[i],self.bins[i]))
        return state
        #raise NotImplementedError("Not implemented yet.")
        # End your code
```

we first use random to determine the action will choosed

if the random number less than self.epsilon, we choose the action by random in the possible action
ifthe random number greater than self.epsilon, we choose the action with maximun value in qtable

```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
        #print(state)
        epsilon = np.random.rand()
        if epsilon<self.epsilon:
            action = self.env.action_space.sample()
        else:
            action = np.argmax(self.qtable[state[0],state[1],state[2],state[3]])
        return action
        raise NotImplementedError("Not implemented yet.")
        # End your code
```

from the formula of Q-learning we update the qtable by the equation.

$$\underbrace{newQ_{S,A}}_{\text{基于状态和行动的新Q值}} = \underbrace{Q_{S,A}}_{\text{c当前Q值}} + \underbrace{\alpha}_{\text{学习效率}} \left(\underbrace{R_{S,A}}_{\text{基于状态和行动的奖励}} + \underbrace{\gamma}_{\text{折扣因子}} * \max \underbrace{Q'(s', a')}_{\text{在给定新的状态和行动下未来最大的奖励}} - Q_{S,A} \right)$$

```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
    max_val = np.max(self.qtable[next_state[0],next_state[1],next_state[2],next_state[3]])

    self.qtable[state[0],state[1],state[2],state[3],action] = self.qtable[state[0],state[1],state[2],state[3],action] + self.learning_rate*(reward+sel
    if done:
        np.save("./Tables/cartpole_table.npy", self.qtable)
    #raise NotImplementedError("Not implemented yet.")
    # End your code
```

just return the maxvalue in the specific state in q-table.

```python
    def check_max_Q(self):
        """
        - Implement the function calculating the max Q value of initial state(self.env.reset()).
        - Check the max Q value of initial state
        Parameter:
            self: the agent itself.
            (Don't pass additional parameters to the function.)
            (All you need have been initialized in the constructor.)
        Return:
            max_q: the max Q value of initial state(self.env.reset())
        """
        # Begin your code
        # TODO
        state =  self.discretize_observation(self.env.reset())
        return np.max(self.qtable[state[0],state[1],state[2],state[3]])
        #raise NotImplementedError("Not implemented yet.")
        # End your code
```

3.DQN.py

from the steps in spec:

1.every 100 time we load(update) the target_net (write by TA)

2.we first get the state,action reqard next_state and done from self.buffer and we turn this values into nparray and turn into tensor matrix and ensure they have the same batch size

4.we put the state and next state into the network to get the Q value and next Q value note thate we only consider the next Q value with haven’t done. Last we use the formula to calculate the target Q value

5.we call the loss function from network and put Qvalue and target Q value into it.

6.we use loss.backward() to reach backprogation

7.we use self.optimizer.step() to optimize the loss function

```python
    def learn(self):
        '''
        - Implement the learning function.
        - Here are the hints to implement.
        Steps:
        -----
        1. Update target net by current net every 100 times. (we have done this for you)
        2. Sample trajectories of batch size from the replay buffer.
        3. Forward the data to the evaluate net and the target net.
        4. Compute the loss with MSE.
        5. Zero-out the gradients.
        6. Backpropagation.
        7. Optimize the loss function.
        -----
        Parameters:
            self: the agent itself.
            (Don't pass additional parameters to the function.)
            (All you need have been initialized in the constructor.)
        Returns:
            None (Don't need to return anything)
        '''
        if self.count % 100 == 0:
            self.target_net.load_state_dict(self.evaluate_net.state_dict())

```

```python
        # Begin your code
        # TODO
        state,action,reward,next_state,done = self.buffer.sample(self.batch_size)
        #print(state)
        #memory = self.buffer.sample(self.batch_size)
        state = torch.FloatTensor(np.asarray(state))
        action = torch.LongTensor(np.asarray(action)).view(self.batch_size, 1)
        reward = torch.IntTensor(np.asarray(reward)).view(self.batch_size, 1)
        next_state = torch.FloatTensor(np.asarray(next_state))
        done = torch.IntTensor(np.asarray(done)).view(self.batch_size, 1)

        q_value  = self.evaluate_net(state).gather(1,action)

        q_next_value = self.target_net.forward(next_state).detach()

        for i in range(self.batch_size):
            if done[i]:
                q_next_value[i] = 0
        q_next_value = q_next_value.max(1)[0].view(self.batch_size, 1)
        target_q_value = reward + self.gamma * q_next_value

        loss_function= nn.MSELoss()
        loss = loss_function(q_value, target_q_value)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")
        # End your code
```

we first use random to determine the action will choosed
if the random number less than self.epsilon, we choose the action by random in the possible action
if the random number greater than self.epsilon, we first turn the state into tensor matrix and use unsqeeze to make it to higher dimension. Then we can put it into the nueral network to see what may the value get after action, we choose the action with max value.

```python
    def choose_action(self, state):
        """
        - Implement the action-choosing function.
        - Choose the best action with given state and epsilon
        Parameters:
            self: the agent itself.
            state: the current state of the enviornment.
            (Don't pass additional parameters to the function.)
            (All you need have been initialized in the constructor.)
        Returns:
            action: the chosen action.
        """

        with torch.no_grad():
            # Begin your code
            # TODO
            epsilon = np.random.rand()
            if epsilon < self.epsilon:
                action = self.env.action_space.sample()
            else:
                print(state)
                tensor = torch.Tensor(state)
                temp = torch.unsqueeze(tensor,0)
                value = self.evaluate_net.forward(temp)
                action  = torch.argmax(value).item()
        return action
```

we first turn the initial state(self.env.reset) into tensor matrix and use unsqeeze to make it to higher dimension. and use the neural network target_net to calculate the values we get after the calculation and we return the max Q value.

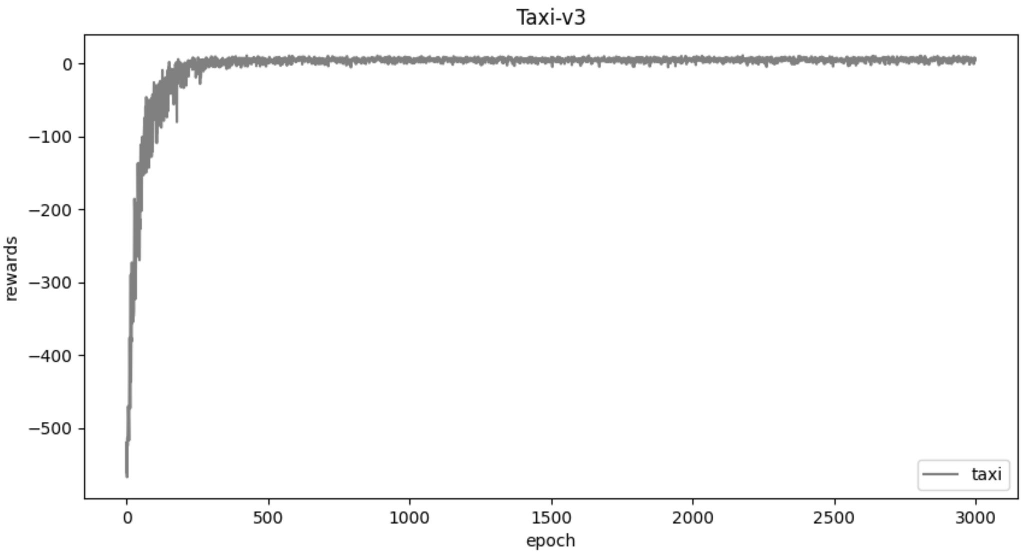
```python
    def check_max_Q(self):
        """
        - Implement the function calculating the max Q value of initial state(self.env.reset()).
        - Check the max Q value of initial state
        Parameter:
            self: the agent itself.
            (Don't pass additional parameters to the function.)
            (All you need have been initialized in the constructor.)
        Return:
            max_q: the max Q value of initial state(self.env.reset())
        """
        # Begin your code
        # TODO

        tensor = torch.FloatTensor(self.env.reset())
        temp = torch.unsqueeze(tensor,0)
        action_values = self.target_net(temp)
        max_Q = torch.max(action_values).item()
        return max_Q
```

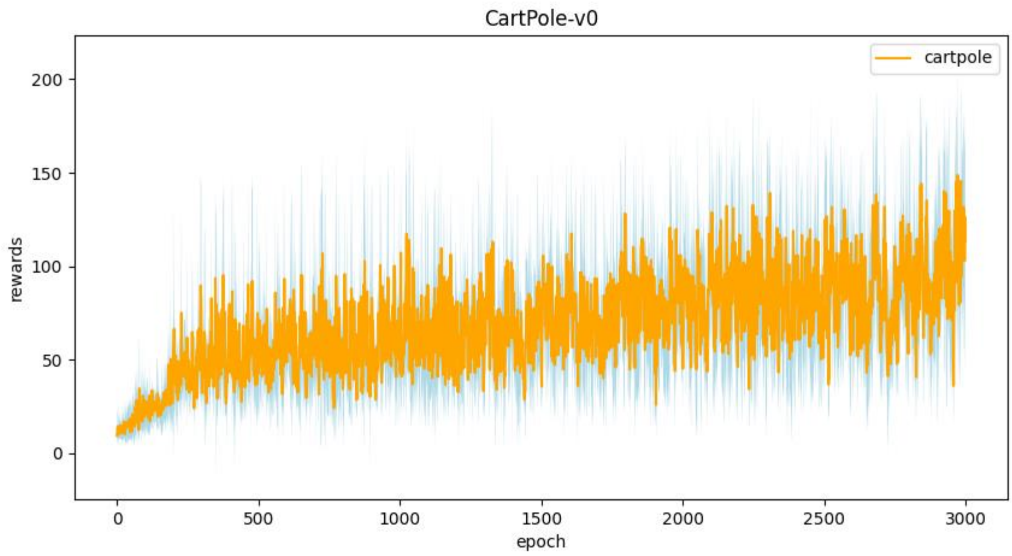
Part II. Experiment Results:

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

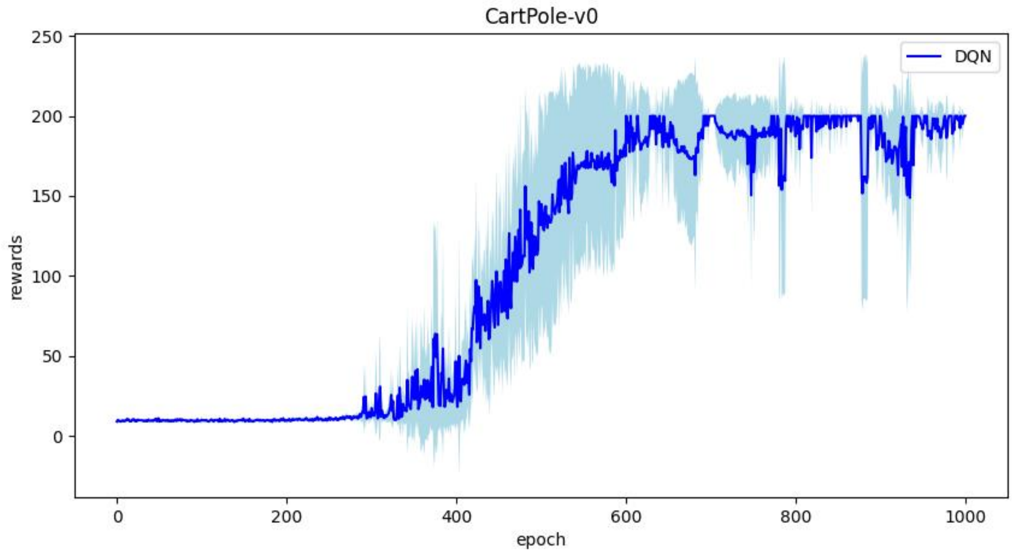
1. taxi.png:



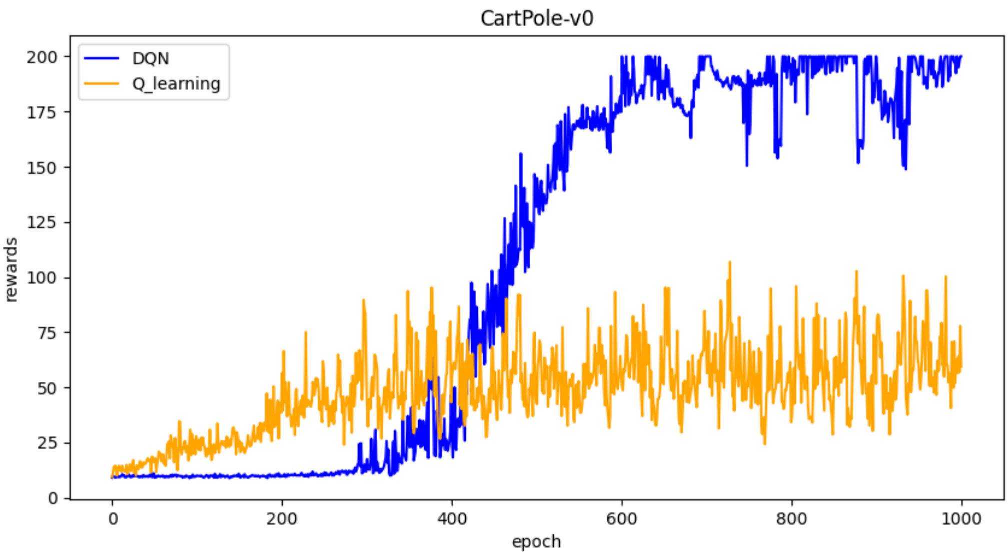
2. cartpole.png



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 [00:17<00:00, 174.99it/s]
#2 training progress
100%|████████████████████████████████████| 3000/3000 [00:17<00:00, 174.60it/s]
#3 training progress
100%|████████████████████████████████████| 3000/3000 [00:17<00:00, 175.47it/s]
#4 training progress
100%|████████████████████████████████████| 3000/3000 [00:16<00:00, 178.34it/s]
#5 training progress
100%|████████████████████████████████████| 3000/3000 [00:16<00:00, 179.15it/s]
average reward: 8.11
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146700000021
```

$\gamma = 0.9$.
taxi at (2,2) passanger at Y desitination at R.

$\Rightarrow L \to L \to D \to D \to$ pick $\to U \to U \to U \to U \to$ drop

(L: left, D: down, U: up) (9 step −1, 1 step 20)

$$Q_{opt} = -1\,(1 + 0.9 + 0.9^2 + \dots + 0.9^8) + 20 \times (0.9)^9$$

$$= -1 \times \frac{1-0.9^9}{1-0.9} + 20 \times 0.9^9$$

$$= 1.622614670000017$$

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%)**

```
#1 training progress
100%|████████████████████████████████████████| 3000/3000 [00:19<00:00, 153.55it/s]
#2 training progress
100%|████████████████████████████████████████| 3000/3000 [00:20<00:00, 149.49it/s]
#3 training progress
100%|████████████████████████████████████████| 3000/3000 [00:18<00:00, 158.16it/s]
#4 training progress
100%|████████████████████████████████████████| 3000/3000 [00:19<00:00, 154.98it/s]
#5 training progress
100%|████████████████████████████████████████| 3000/3000 [00:22<00:00, 135.40it/s]
average reward: 177.2
max Q:31.72239218300627
```

$\gamma = 0.97$, reward $= 177.2$

$\Rightarrow Q_{opt} = 1 + 0.97 + 0.97^2 + \dots + 0.97^{176}$

$= \dfrac{1-(0.97)^{177}}{1-0.97}$ ← make it more precise

replace 177 with 177.2

$\approx \dfrac{1-(0.97)^{177.2}}{1-0.97} \approx 33.1823843 9194456$

3.

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

Because the data we observe iscontinuous, but we need to make it into discrete
states, so we have to discretize it to get the data in which interval.

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

The performance will become greater, because we will have more states, and the
data after discretize will be more closer to the true data.

c. Is there any concern if we increase “num_bins”? **(3%)**

If we increase “num_bins” the process speed may become slower, because we increase the number of states and also make the size of Q-table bigger and need more space to calculate.

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

DQN perform better in cartpole-v0,in Qlearning we should devided the observation to limited discrete data and DQN can directly use the continuous data, which is more familier to real data. so DQN can have higher performance.

5.

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

the purpose is to balance exploration and exploitation, it can prevent the extreme cases, always choose the rangom action to

get more information, or choose the best action in with max Q value,(there may be other better action we havn' t explore).

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

   if we always use explore, choose action randomly, the episode will be easily ended and since we don' t choose the best action in qtable , we may not construct a good Q table.
   if we always use exploit, we only choose the action that is known, this may cause us can' t not explore the better performance action that we haven' t explore and can' t get a good Q table .

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

   Yes it is possible. Since the CartPole-v0 environment is easily, if we can find another method to balance the explore and exploit rate, we can get the same performance in the CartPole-v0 environment.

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

   Because in the training section we need to find new action that we haven' t explore. But in the testing section we only want to test the q table we got after training, so we not need the epsilon greedy algorithm during the testing section.

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

   we don' t need to calculate the gradient and back paropagation in choose action, so we use with torch.no_grad() to disable gradient tracking.