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.

Part 1: taxi.py

```
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
    """
    Get a random number(rnd) to decide what action should do.
    if rnd > epsilon, then find the max Q and get its action.
    else, get the random action.
    """
    rnd = random.uniform(0, 1)
    if rnd > self.epsilon:
        action = np.argmax(self.qtable[state])
    else:
        action = self.env.action_space.sample()
    return action
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

```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
    """
    Follow the formula below to get the qtable value.
    Q(s,a) = (1-lr)Q(s,a) + lr*(reward + gamma*maxQ(s',a'))
    If done, save the npy.
    """
    self.qtable[state, action] = (1-self.learning_rate)*self.qtable[state, action]      \
                                        + self.learning_rate*(reward + self.gamma*np.max(self.qtable[next_state]))
    if not done:    return
    # raise NotImplementedError("Not implemented yet.")
    # End your code
    np.save("./Tables/taxi_table.npy", self.qtable)
```

```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
    """
    Find the max_q in given state according to the qtable.
    """
    max_q = np.max(self.qtable[state])
    return max_q
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

Part 2: cartpole.py

```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
    """
    Get the quantiles by np.arange().
    ex. [0, 10] into five parts will get [0. 2. 4. 6. 8.].
    Return the np array after ignoring the first term.
    """
    arr = np.arange(lower_bound, upper_bound, (upper_bound-lower_bound)/num_bins)
    return arr[1:]
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

```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
    """
    Get the index by np.digitize().
    """
    return np.digitize(value, bins)
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

```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
    """
    The input observation includes 4 parts.
    Do the discretize_value() respectively, and we can get the discretize state.
    """
    state = [0, 0, 0, 0]
    for i, ob in enumerate(observation):
        state[i] = self.discretize_value(ob, self.bins[i])
    return state
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

```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
    """
    Get a random number(rnd) to decide what action should do.
    if rnd > epsilon, then find the max Q and get its action.
    else, get the random action.
    """
    rnd = random.uniform(0, 1)
    if rnd > self.epsilon:
        action = np.argmax(self.qtable[tuple(state)])
    else:
        action = self.env.action_space.sample()
    return action
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

```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
    """
    Follow the formula below to get the qtable value.
    Q(s,a) = (1-lr)Q(s,a) + lr*(reward + gamma*maxQ(s',a'))
    The type of state is list, but we should put tuple in qtable, so change the type of state.
    If done, save the npy.
    """
    self.qtable[tuple(state) + (action, )] = (1-self.learning_rate)*self.qtable[tuple(state) + (action, )]     \
                                + self.learning_rate*(reward + self.gamma*np.max(self.qtable[tuple(next_state)]))
    if not done:    return
    # raise NotImplementedError("Not implemented yet.")
    # End your code
    np.save("./Tables/cartpole_table.npy", self.qtable)
```

```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
    """
    Get the initial state(env.reset()) after discretize it.
    Find the max_q of the state according to the qtable.
    """
    state = self.discretize_observation(self.env.reset())
    max_q = np.max(self.qtable[state])
    return max_q
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

Part 3: DQN.py

```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
        """
        Record the state from np array to torch and unsqueeze it.
        Get a random number(rnd) to decide what action should do.
        if rnd > epsilon, predict the Q_value by evaluate_net(), find the max Q and get its action.
        else, get the random action.
        """
        x = torch.unsqueeze(torch.tensor(state, dtype=torch.float), 0)
        rnd = np.random.uniform(0, 1)
        if rnd > self.epsilon:
            action_value = self.evaluate_net(x)
            action = torch.argmax(action_value).item()
        else:
            action = self.env.action_space.sample()
        # raise NotImplementedError("Not implemented yet.")
        # End your code
    return action
```

```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())

    # Begin your code
    # TODO
    """
```

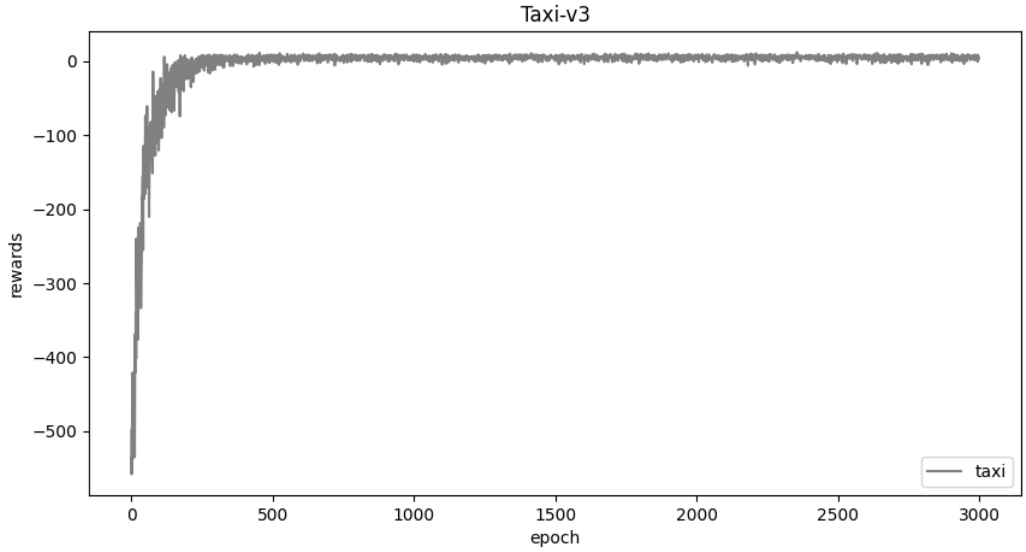
```python
# TODO
"""
Get the samples from buffer by the function "sample" in replay_buffer.
Record s0, a0, r1, s1, _ from samples and turn them to torch.
(state, action, reward, next_state, done) for (s0, a0, r1, s1, _).
Forward the state to evaluate_net and target_net.
Predict the Q value and find the max value.
MSELoss(predict_now, Q_target).
Zero-out, backpropagation, and optimize the loss function.
"""
samples= self.buffer.sample(self.batch_size)

s0 = torch.FloatTensor(np.array(samples[0]))
a0 = torch.LongTensor(samples[1])
r1 = torch.FloatTensor(samples[2])
s1 = torch.FloatTensor(np.array(samples[3]))
_ = torch.FloatTensor(samples[4])
_ = 1 - _
batch_index = np.arange(self.batch_size, dtype=np.int64)

Q_now = self.evaluate_net.forward(s0)
Q_next = self.target_net.forward(s1)

pred_now = Q_now[batch_index, a0]
pred_next = torch.max(Q_next, dim=1)[0]

Q_target = r1 + self.gamma * pred_next * _
loss_fn = nn.MSELoss()
loss = loss_fn(pred_now, Q_target)

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()

if not samples[4]:  return
# raise NotImplementedError("Not implemented yet.")
# End your code
```

```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
    """
    Record the initial state from np array to torch, and unsqueeze it.
    Return the target_net of the state and get its max value.
    """
    initial_state = torch.unsqueeze(torch.FloatTensor(self.env.reset()),0)
    return max(max(self.target_net(initial_state)))
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

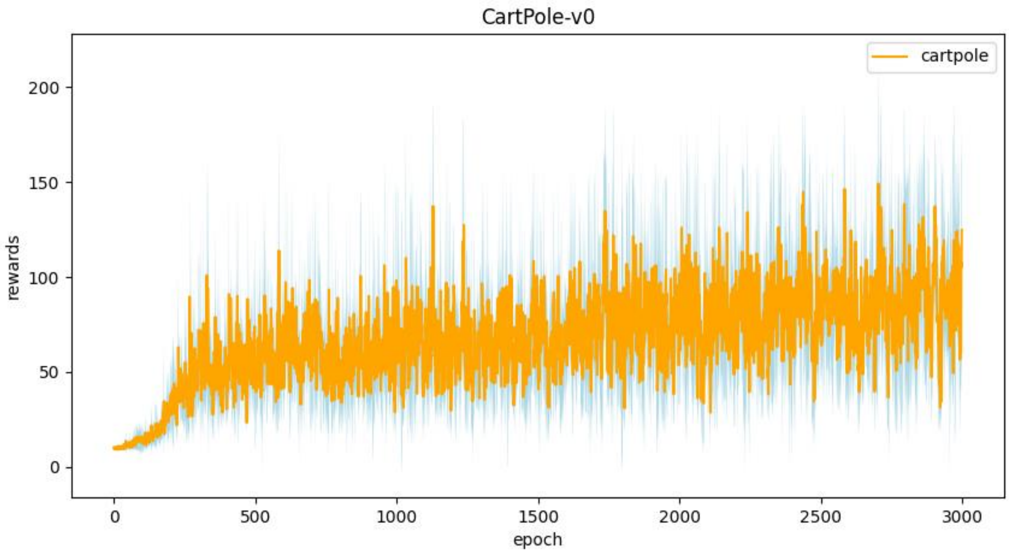
Part II. Experiment Results:

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

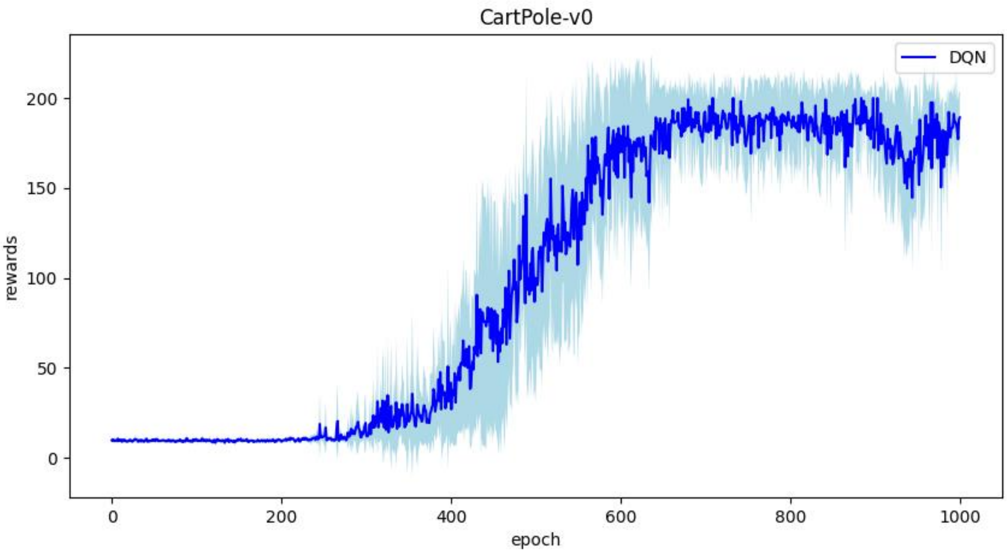
1. taxi.png



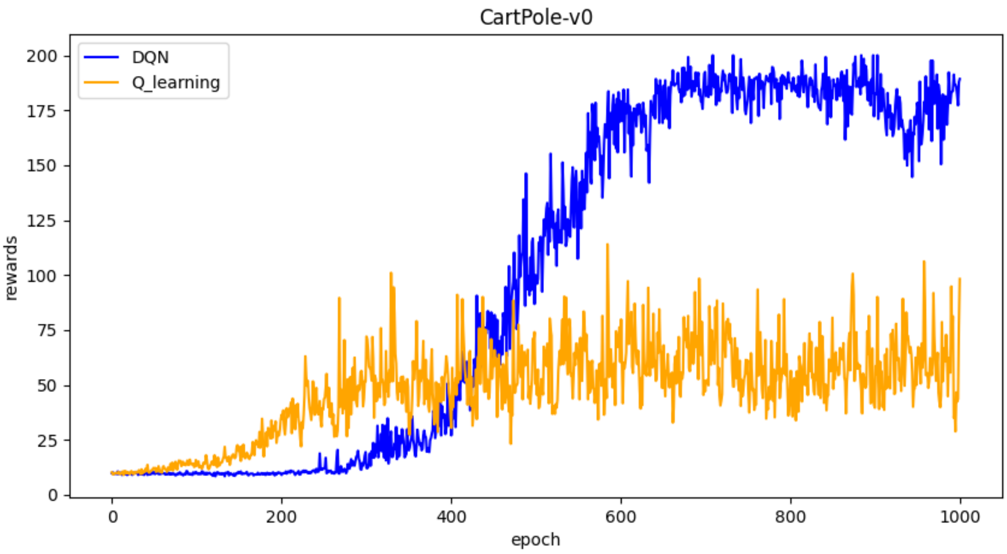
2. cartpole.png



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%)**

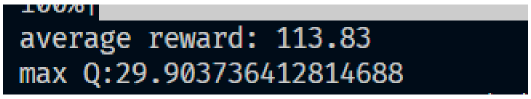
```
average reward: 7.74
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146699999995
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

Q = (-1)*(1-r^num_steps)/(1- r)+(20*r^num_steps)= -1*(1-0.9^9) / 0.1 + 20*0.9^9 = 1.622

optimal Q: L -> L -> D -> D -> pickup -> U -> U -> U -> U -> dropoff

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



```
100%|
average reward: 113.83
max Q:29.903736412814688
```

Q = (1 - r^ave.reward) / (1 - r) = 31.74...

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

      The observed data is continuous, so we discretize it to several parts.

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

      I think it may be better because it can separate to more parts so that it can be more precise.

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

      The efficiency may be slower and it need more cost on qtable.

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

   DQN model.

   From compare.png, we can find that whem we try over about 400 opoches, the reward of DQN will be higher than Q-learning. I think it is because that Q-learning need to discretize to several parts and DQN can use the continuous data.

5.
   a. What is the purpose of using the epsilon greedy algorithm while choosing an actio n? **(3%)**

      It can balance exploration and exploitation by choosing them randomly.

   b. What will happen, if we don' t use the epsilon greedy algorithm in the CartPole-v 0 environment? **(3%)**

      All exploration: (All random) Cannot remain the known best situation.

      All exploitation: Can only work on known situation, may miss some case.

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

      Yes, epsilon greedy alogithm select randomly. I think there might be some ways m ore reliable, maybe use the probability or something else.

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

      We use this alogorithm to train the model and we don' t need to do that while test ing.

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

In “choose_action” function, “with torch.no_grad()” can disable gradient calculation during a block of code. It will set the “requires_grad” to False so that it can save many memory and computation.