Homework 4:

Reinforcement Learning

Report Template

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

Part 1. All details after # TODO

```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
    '''
    if a random number is less than epsilon, return random action (expore)
    else return the action with maximum Q value
    '''
    if np.random.random() < self.epsilon:
        return np.random.randint(self.env.action_space.n)
    action = np.argmax(self.qtable[state])

    return action

    # 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
    '''
    update the qtable according to the formula give in class
    '''
    self.qtable[state][action] = (1 - self.learning_rate) * self.qtable[state][action] + \
        self.learning_rate * (reward + self.gamma * np.max(self.qtable[next_state]))
    # 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

    value = np.max(self.qtable[state])
    return value
    # End your code
```

Part 2.

```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
    '''
    since the np.linspace() will have points including the first and last point,
    so delete the first and last element
    '''
    points = np.linspace(lower_bound, upper_bound, num_bins+1)  # call function to devide
    points = np.delete(np.delete(points, num_bins), 0)          # delete first & last
    return points
    # 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
    '''
    use numpy.digitize to do the task described above
    '''
    temp = np.array([value])
    index = np.digitize(temp, bins)
    return index[0]
    # 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
    '''
    declare a empty list, and iterate through each element in observation and
    use discretize_value to find the value with corresponding bins
    '''
    discretized = []
    for i in range(4):
        discretized.append(self.discretize_value(observation[i], self.bins[i]))
    return discretized
    # 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
    '''
    if a random number is less than epsilon, return random action (expore)
    else return the action with maximum Q value
    '''
    if np.random.random() < self.epsilon:
        return np.random.randint(self.env.action_space.n)
    return np.argmax(self.qtable[state[0]][state[1]][state[2]][state[3]])
    # 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
    '''
    update the qtable as the fuction given in class
    '''
    self.qtable[state[0], state[1], state[2], state[3], action] = \
        self.qtable[state[0], state[1], state[2], state[3], action] * \
        (1-self.learning_rate) + self.learning_rate * (reward + self.gamma * \
        np.max(self.qtable[next_state[0], next_state[1], next_state[2], next_state[3]]))
    # 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
    state = self.env.reset()
    state = self.discretize_observation(state)
    return np.max(self.qtable[state[0]][state[1]][state[2]][state[3]])
    # End your code
```

Part 3.

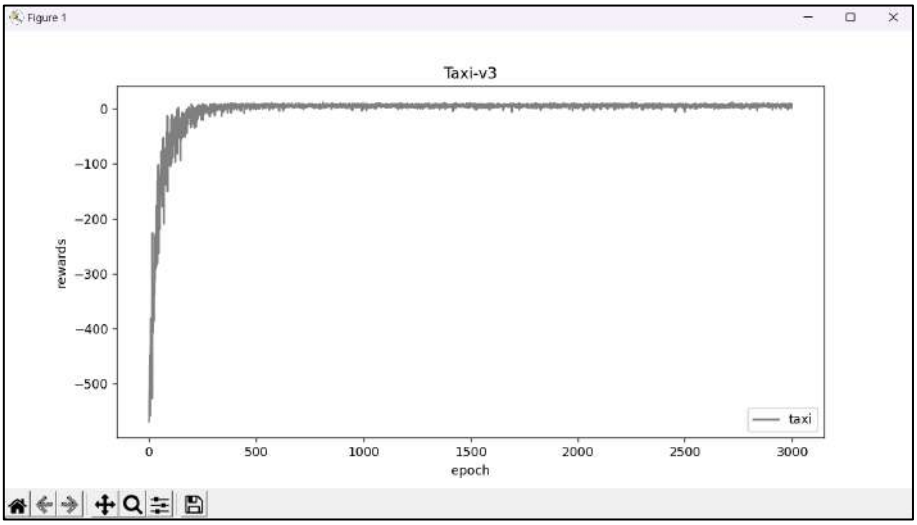
```python
def learn(self):
    '''
    - Implement the learning function.
    - Here are the hints to implement.
    Steps:
    -----
    1. Update target net by current net every 100 times. (we have done this for you)
    2. Sample trajectories of batch size from the replay buffer.
    3. Forward the 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
    # Step2: Sample the data stored in the buffer and store them into data type Tensor
    states, actions, rewards, next_states, dones = self.buffer.sample(self.batch_size)
    states = torch.tensor(np.array(states), dtype=torch.float)
    actions = torch.tensor(np.array(actions), dtype=torch.int64).unsqueeze(-1)
    rewards = torch.tensor(np.array(rewards), dtype=torch.float)
    next_states = torch.tensor(np.array(next_states), dtype=torch.float)
    dones = torch.tensor(np.array(dones), dtype=torch.float)

    # Step3: Forward the data to the evaluate net and the target net with a few adjustment of the size
    q_values = torch.gather(self.evaluate_net(states), 1, actions)
    next_q_values = self.target_net(next_states).detach()
    next_q_values, _ = torch.max(next_q_values, dim=1)
    target_q_values = (rewards + self.gamma * (1 - dones) * next_q_values).unsqueeze(1)

    # Step4: Compute the loss with MSE.
    loss = F.mse_loss(q_values, target_q_values)

    # Step5: Zero-out the gradients.
    self.optimizer.zero_grad()

    # Step6: Backpropagation.
    loss.backward()

    # Step7: Optimize the loss function.
    self.optimizer.step()

    # End your code
    torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")
```

```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
        if np.random.random() < self.epsilon:
            return np.random.randint(self.n_actions)
        # forward the state to nn and find the argmax of the actions
        action = torch.argmax(self.evaluate_net(Tensor(state))).item()
        # End your code
    return action
```

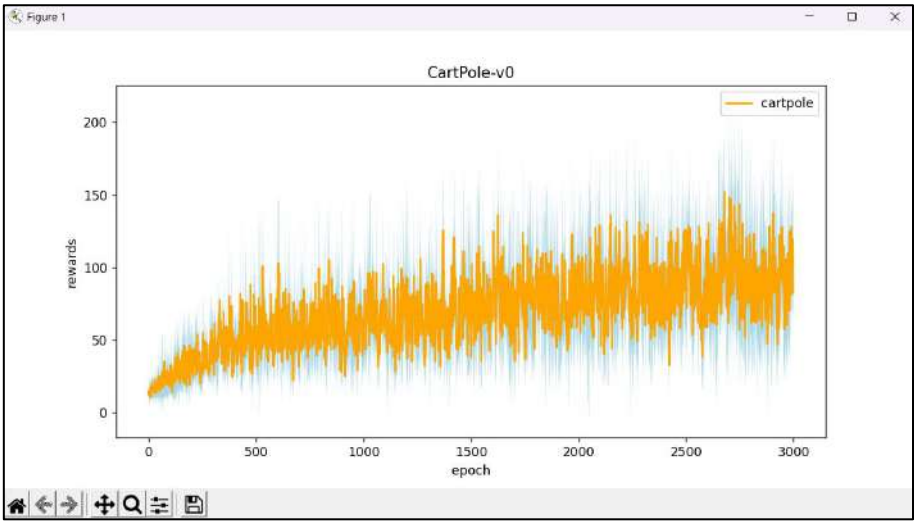
```python
def check_max_Q(self):
    """
    - Implement the function calculating the max Q value of initial state(self.env.reset()).
    - Check the max Q value of initial state
    Parameter:
        self: the agent itself.
        (Don't pass additional parameters to the function.)
        (All you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

Part II. Experiment Results:

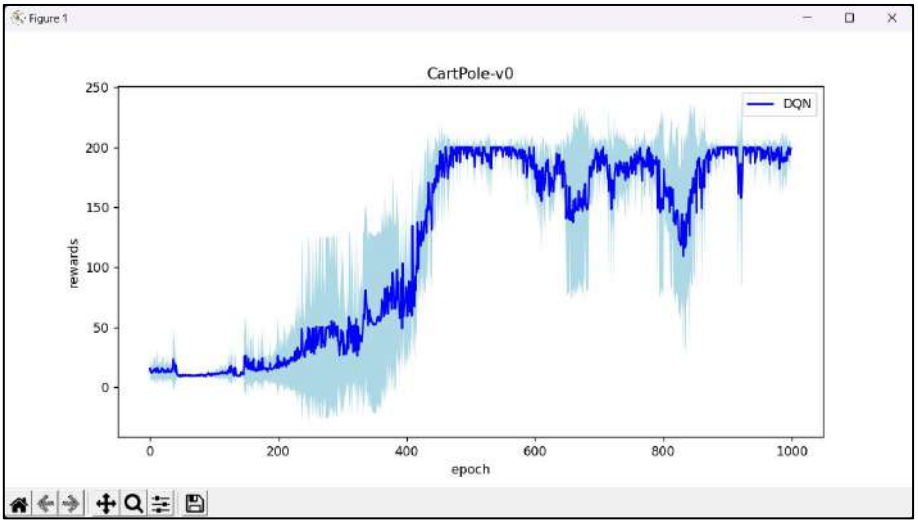
1. taxi.png:



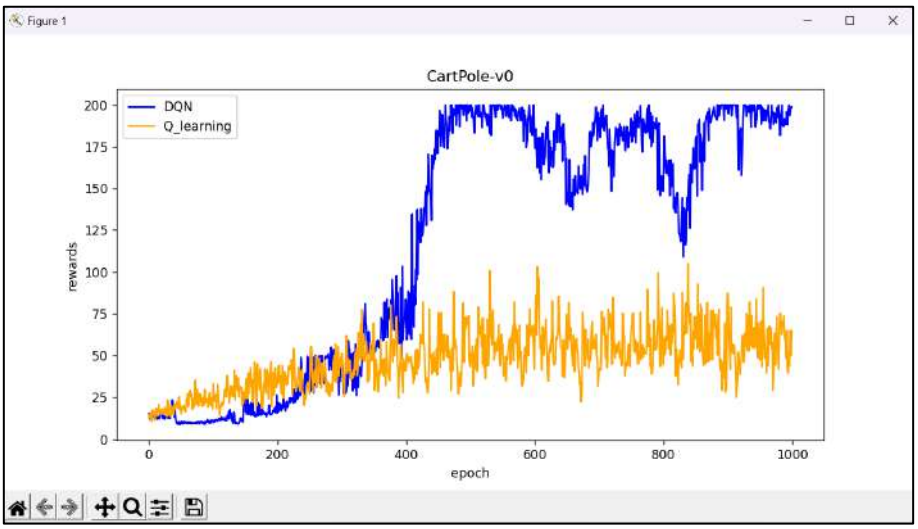
2. cartpole.png



3. DQN.png



4. compare.png



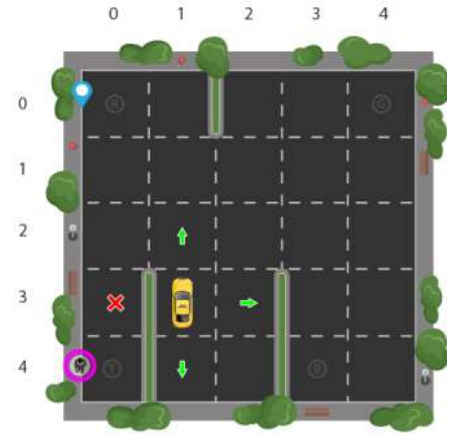
Part III. Question Answering (50%):

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%)**

   Using the reward and map:

Rewards

- -1 per step unless other reward is triggered.
- +20 delivering passenger.
- -10 executing "pickup" and "drop-off" actions illegally.




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

Optimal Q-value = -8 + 20 = 12

The Q value I learned is 1.6226… , which is way smaller than the optimal Q value

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%)**

Episode End

The episode ends if any one of the following occurs:

1. Termination: Pole Angle is greater than ±12°
2. Termination: Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
3. Truncation: Episode length is greater than 500 (200 for v0)

The max Q of initial state is 200 because the episode end when length>200 and reward of each action is 1.

```
average reward: 91.71
max Q:30.3191821427827
```

The maximum Q value I got from the program is also a lot smaller than 200.

3.

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

If we do not dicretize the observation, the observations would have to be regarded as a particular state, and the number of state will be in some way similar to infinite.

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

The performance might be better if we increase the num_bins because it will be more accrate to have more state being catogorized.

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

We might need more data and more time to train because the size of states increases a lot (the size of states is O(n^4))

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

DQN performs better in Cartpole-v0 (as shown in compare.png)
DQN tends to perform better than discretized Q-learning in Cartpole-v0 due to its ability to handle continuous state and action spaces and learn more complex policies through its neural network approximation of the Q-function.

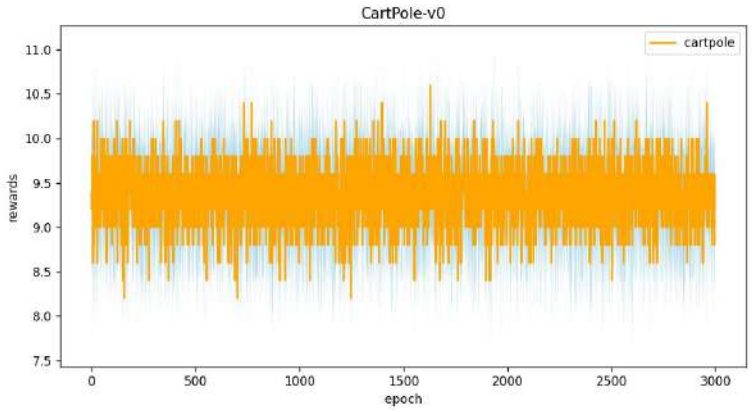
5.

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

The epsilon is to choose whether you should explore (choose random action) or expoit (choose action with max Q)

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

If epsilon greedy algorithm is not used, the action can not be explored, and the reward will be lower than using greedy algorithm.
Graph of cardpole.py without epsilon greedy algorithm:



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

   It is possible to have the same performance if you use all-explore, it just need more time to converge.

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

   During the testing section, what we should do is to evalurate how well are Qtable is, so we should choose the best action.

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

   Using with torch.no_grad() ensures that the computational graph is not built and PyTorch does not track operations during the forward pass. This helps to reduce memory consumption and improve the efficiency of the function.