交大資工 朱致伶
110550062

Homework 4:Reinforcement Learning

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

1. taxi.py (Q learning in Taxi-v3):

- **choose_action:**

```
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
    # Perform Epsilon greedy to decide whtether to explore or exploit.
    # Explore
    if random.uniform(0, 1) < self.epsilon:
        action = self.env.action_space.sample()
    # Exploit
    else:
        action = np.argmax(self.qtable[state, :])
    return action
    # End your code
```

- **learn:**

```
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
    # Renew the Q-value of the current state-action pair
    cur_qvalue = self.qtable[state, action]
    next_max = np.max(self.qtable[next_state, :])
    new_qvalue = (1 - self.learning_rate) * cur_qvalue + self.learning_rate * (reward + self.gamma * next_max)
    self.qtable[state, action] = new_qvalue
    # End your code
    np.save("./Tables/taxi_table.npy", self.qtable)
```

- **check_max_Q:**

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

2. Cartpole.py (Q learning in CartPole-v0):

- **init_bins:**

```
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.
    """
    # Begin your code
    # Compute the quantiles and return them as an array
    return np.linspace(lower_bound, upper_bound, num_bins, endpoint=False)[1:]
    # Since the first and last quantile are the lower and upper bound,
    # we only need to return the quantiles from the second to the second last.
    # End your code
```

- **discretize_value:**

```
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.
    """
    # Begin your code
    # Find the index of the quantile that the value belongs to
    return np.digitize([value], bins)[0]
    # End your code
```

- **discretize_observation:**

```
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.
    """
    # Begin your code
    state = []
    for i in range(len(observation)):
        s = self.discretize_value(observation[i], self.bins[i]) # discretize each feature in observation
        state.append(s)
    return tuple(state) # return the discretized state in tuple form
    # End your code
```

- **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
    if np.random.uniform(0, 1) < self.epsilon: # Explore
        action = self.env.action_space.sample()
    else:  # Exploit
        action = np.argmax(self.qtable[state])
    return 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
    # 2 cases(done / not doen) to update the qtable
    if not done:
        cur_qvalue = self.qtable[state+(action,)]
        next_max = np.max(self.qtable[next_state])
        # If the episode is not done, we need to consider both the current and future reward
        new_qvalue = (1 - self.learning_rate) * cur_qvalue + self.learning_rate * (reward + self.gamma * next_max)
        self.qtable[state+(action,)] = new_qvalue
    else:
        cur_qvalue = self.qtable[state+(action,)]
        next_max = np.max(self.qtable[next_state])
        # If the episode is done, we only need to consider the current reward
        new_qvalue = (1 - self.learning_rate) * cur_qvalue + self.learning_rate * (reward)
        self.qtable[state+(action,)] = new_qvalue
    # End your code
    if done:
        np.save("./Tables/cartpole_table.npy", self.qtable)
```

- **Check_max_Q:**

```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
    state = self.discretize_observation(self.env.reset())
    max_q = np.max(self.qtable[state])
    return max_q
    # End your code
```

3. DQN.py (DQN in CartPole-v0)

- **learn:**

```python
def learn(self):
    '''
    - Implement the learning 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)
    '''
    # 1. Update target net by current net every 100 times.
    if self.count % 100 == 0:
        self.target_net.load_state_dict(self.evaluate_net.state_dict())

    # Begin your code
    # 2. Sample trajectories of batch size from the replay buffer.
    b_state, b_action, b_reward, b_next_state, b_done = self.buffer.sample(self.batch_size)

    b_state = torch.FloatTensor(np.array(b_state))
    b_action = torch.LongTensor(b_action)
    b_reward = torch.FloatTensor(b_reward)
    b_next_state = torch.FloatTensor(np.array(b_next_state))
    b_done = torch.BoolTensor(b_done)

    # 3. Forward the data to the evaluate net and the target net.
    q_eval = self.evaluate_net(b_state).gather(1, b_action.reshape(self.batch_size, 1))
    q_next = self.target_net(b_next_state).detach()
    q_target = b_reward.reshape(self.batch_size, 1) + self.gamma * q_next.max(1)[0].view(self.batch_size, 1) * (~b_done).reshape(self.batch_size, 1)

    # 4. Compute the loss with MSE.
    MSE = nn.MSELoss()
    loss = MSE(q_eval, q_target)
    # 5. Zero-out the gradients.
    self.optimizer.zero_grad()
    # 6. Backpropagation.
    loss.backward()
    # 7. Optimize the loss function.
    self.optimizer.step()

    self.count += 1
    # End your code
    torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")
```

- **choose_action:**

```
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
        if random.uniform(0,1) < self.epsilon: # explore in epsilon probability
            action = self.env.action_space.sample()
        else: # exploit the best action
            action = torch.argmax(self.evaluate_net.forward(torch.FloatTensor(state))).item()

        # End your code
    return action
```

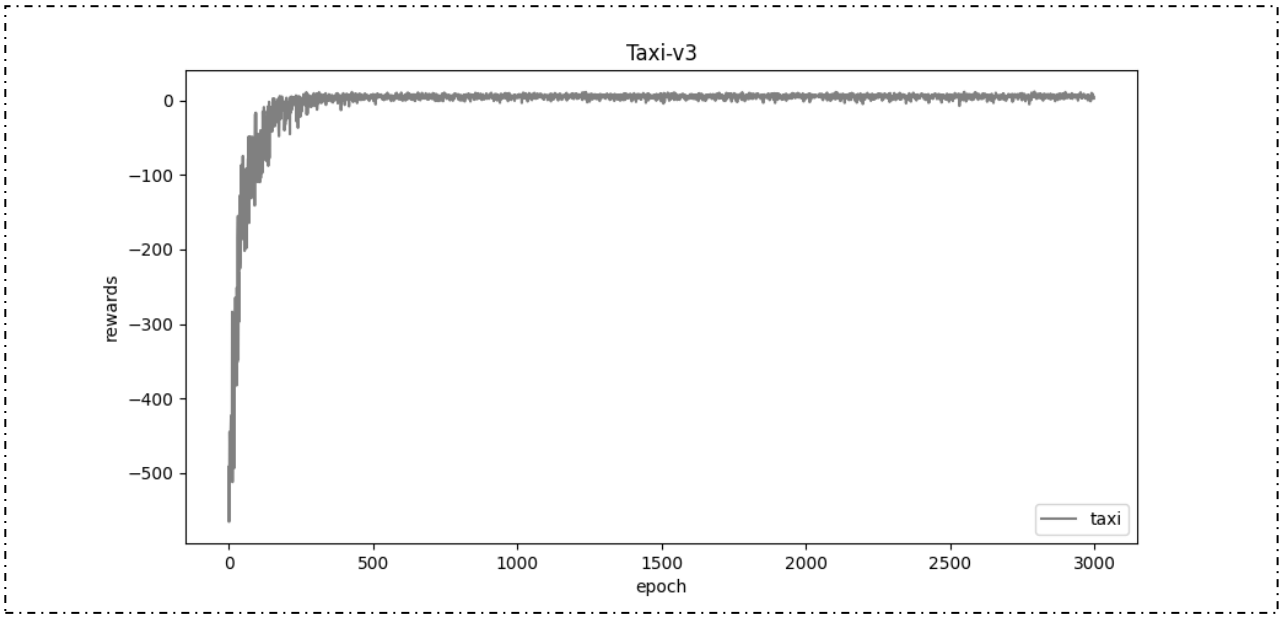
- **check_max_Q:**

```
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
    initial_state = self.env.reset()
    with torch.no_grad(): # no need to calculate the gradient or update the parameters
        q_values = self.target_net(torch.FloatTensor(initial_state))
        max_q = torch.max(q_values)
    return max_q
    # End your code
```

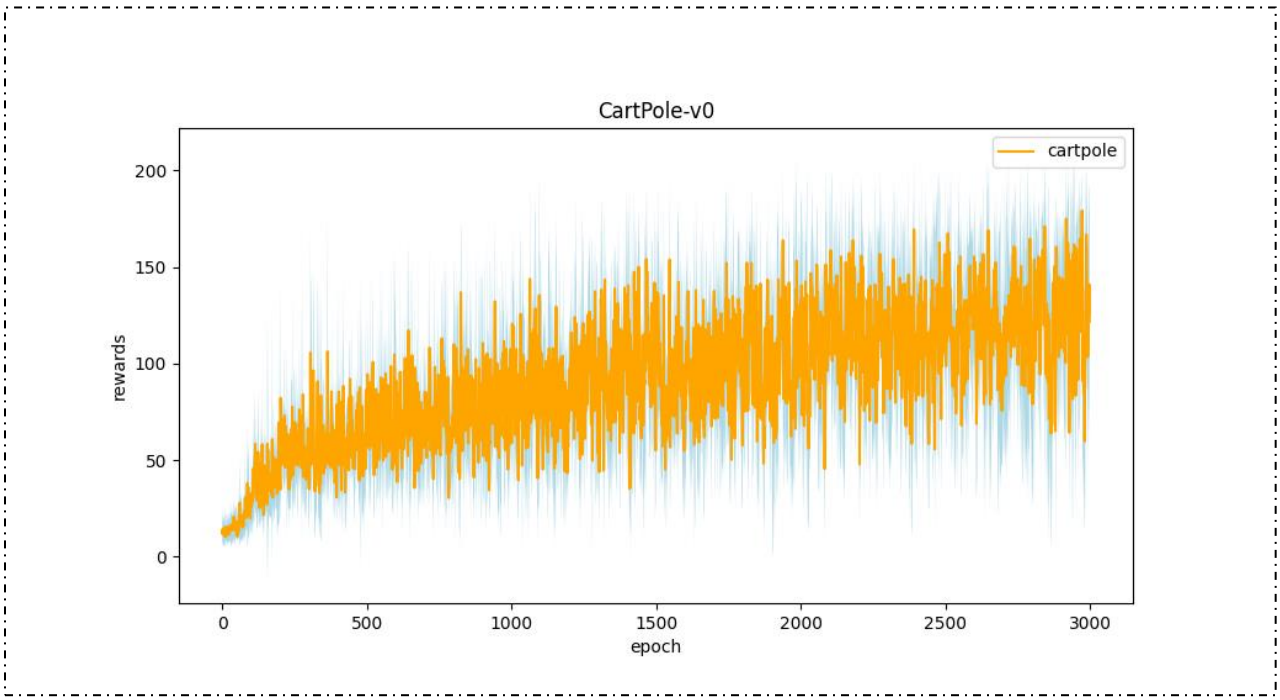
交大資工 朱致伶
110550062

Part II. Experiment Results: Please paste taxi.png, cartpole.png, DQN.png and compare.png.

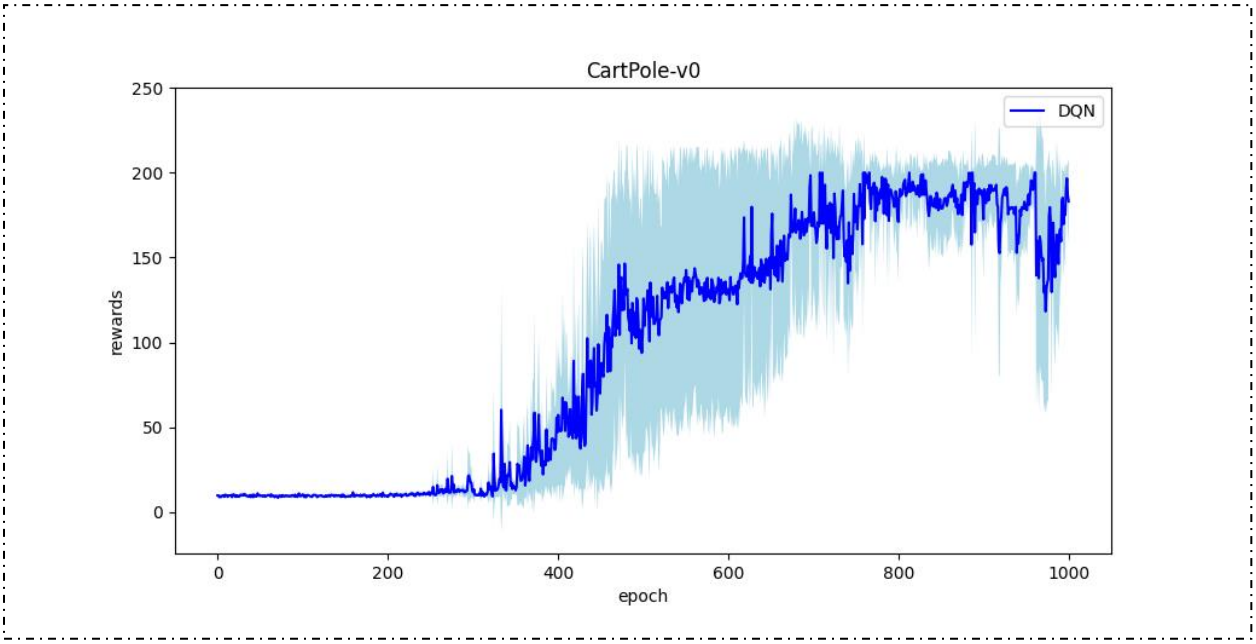
1. taxi.png:



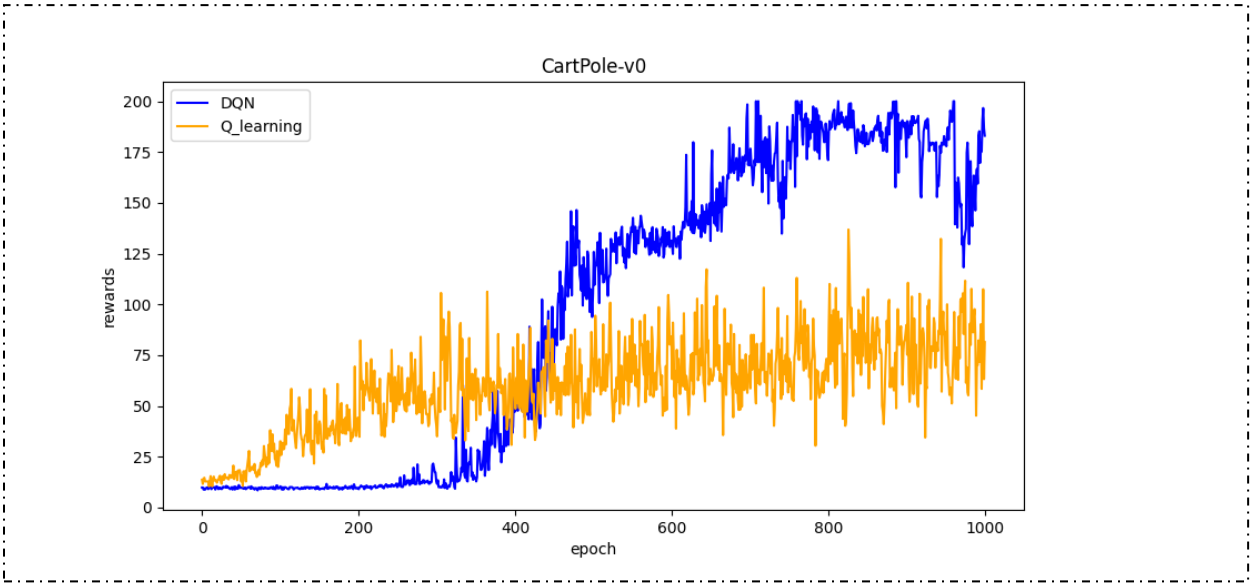
2. cartpole.png:



3. DQN.png:



4. compare.png



交大資工 朱致伶
110550062

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%)**

```
!python  taxi.py

#1 training progress
100% 3000/3000 [06:13<00:00,  8.04it/s]
#2 training progress
100% 3000/3000 [06:15<00:00,  8.00it/s]
#3 training progress
100% 3000/3000 [06:15<00:00,  8.00it/s]
#4 training progress
100% 3000/3000 [06:11<00:00,  8.08it/s]
#5 training progress
100% 3000/3000 [06:08<00:00,  8.14it/s]
average reward: 8.02
Initail state:
taxi at (2, 2), passenger at Y, destination at
max Q:1.6226146699999995
```

Since taxi at (2, 2), passenger at Y and destination at R, the optimal policy is $\rightarrow, \rightarrow, \downarrow, \downarrow, pick, \uparrow, \uparrow, \uparrow, \uparrow, drop$. Then the cultimative rewards is $9 * (-1) + 20 = 11$.

Optimal Q-value =

$$\frac{(-1)(1-r^9)}{1-r} + 20r^9 = \frac{(-1)(1-0.9^9)}{1-0.9} + 20 * 0.9^9 = 1.6226$$

Same as the computed Q-value from taxi-v3.

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%)**

Since rewards in CartPole-v0 environment is cumulative discounted rewards, R_{t+k+1} is constant and equal to 1, gamma=0.97.

$U_t = R_{t+1} + rR_{t+2} + r^2R_{t+3} + \cdots = \sum_{k=0}^{\infty} r^k = \frac{1}{1-r} = \frac{1}{1-0.97} = 33.334$.

The difference between Q value from DQN.py and max Q value is $35.222 - 33.334 = 1.888$.
The difference between Q value from cartpole.py and max Q value is $33.334 - 30.546 = 2.788$.

DQN has the closer Q-value to max Q-value compared to CartPole.

We can see that difference between Q value from DQN.py is smaller than difference between Q value from cartpole.py. Because cartpole.py discretize observations causing information loss.

```
!python  cartpole.py

#1 training progress
100% 3000/3000 [00:37<00:00, 79.86it/s]
#2 training progress
100% 3000/3000 [00:38<00:00, 78.19it/s]
#3 training progress
100% 3000/3000 [00:37<00:00, 79.03it/s]
#4 training progress
100% 3000/3000 [00:37<00:00, 79.30it/s]
#5 training progress
100% 3000/3000 [00:37<00:00, 80.10it/s]
average reward: 65.1
max Q:30.546204759415097
```

```
!python  DQN.py

#1 training progress
100% 1000/1000 [13:27<00:00,  1.24it/s]
#2 training progress
100% 1000/1000 [08:27<00:00,  1.97it/s]
#3 training progress
100% 1000/1000 [15:34<00:00,  1.07it/s]
#4 training progress
100% 1000/1000 [09:03<00:00,  1.84it/s]
#5 training progress
100% 1000/1000 [14:39<00:00,  1.14it/s]
reward: 143.8
max Q:35.221832275390625
```

交大資工 朱致伶
110550062

3.

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

Because Q-learning create a table to store all state-action pairs, and the continuous observation space may have infinite possible state, the limitation of table size made us unable to store all feature values. Therefore, discretizing the observation space reduces the number of possible states and enables the use of Q-learning.

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

It will improve the model's performance.

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

It may take more time and resource to train the model.

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

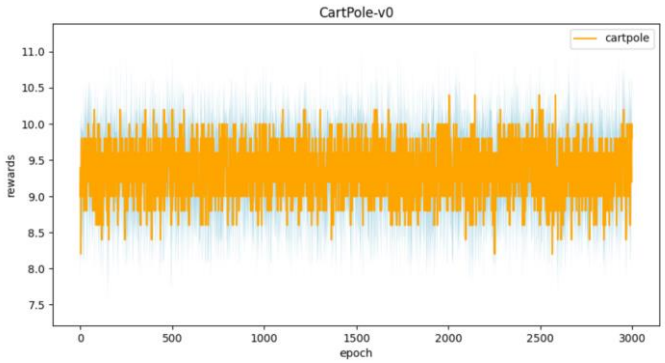
DQN has better performance compared with discretized Q learning for the below 2 reasons.
Reason 1: Discretized Q learning discretizes the observations, and discretization inevitably loses information and precision.
Reason 2: DQN uses a neural network to estimate the Q-values, which can handle complex and nonlinear relationships between the states and actions.

5.

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

At the beginning of the training process, the agent has none or limited knowledge about the environment, so we need to let the agent explore the environment by using the epsilon greedy algorithm.
To explore the environment, the agent has a chance as large as the value of epsilon to take new action. On the other hand, the agent also exploits the current best action if not exploring.

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



The trained result may look like the graph above. Since there is no balance between exploration and exploitation, the agent can't learn new information for the environment or find optimal policy among 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%)**

   Yes. If we can learn new information from the environment, the approach isn't limited to the epsilon greedy algorithm. We can try approaches such as Randomized Probability Matching.

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

   In testing section, we only have to choose the action with the highest q-value over a random action, since the agent has already optimized policy during training process.

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

   Since we only want to choose action in network, don't want to update the network here.