Homework 4:

Reinforcement Learning

Part l. Implementation:

taxi:

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

In the choose action part, I declare a variable called ‘p’ to store the random number, like possibility. If it is smaller than ‘epsilon’, then action is the one which has the maximum value in ‘qtable’ with this state. If not, then choose one action randomly by using ‘sample()’.

```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
    mx = np.max(self.qtable[next_state])
    new = (1-self.learning_rate)*self.qtable[state,action]+self.learning_rate*(reward
    +self.gamma*mx)

    self.qtable[state, action] = new
    # raise NotImplementedError("Not implemented yet.")
    # End your code
    np.save("./Tables/taxi_table.npy", self.qtable)
```

In learn part, I implement the formula of new value of the state

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

which alpha is learning rate, and old value is from ‘qtable’ with this state and action, and r_t is reward, and discount factor is ‘self.gamma’, and maximum value of next state is from ‘qtable’ but using ‘next_state’ to search.

After this calculation, updating the value in the ‘qtable’ with this state and action.

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

In check max Q part, What I do is just find the maximum value in ‘qtable’ with the state and return.

cartpole:

```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
    slices = np.linspace(lower_bound, upper_bound, num_bins+1)
    return slices[1:-1]
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

In init bins part, I use function ‘np.linspace’ to implement. Because in this function, the number of slices will equals to the third component value, I add the ‘num_bins’ by 1, and return the result without the first one and the last one.

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

In discretize value part, I use the function ‘np.digitize’ to implement. By this function, I can find which interval includes this value((-inf, first component) is index 0), and return the index generated by this function.

```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 = []
    for i in range(len(observation)):
        state.append(self.discretize_value(observation[i], self.bins[i]))
    return state
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

In discretize observation part, I implement by the function I have done called ‘discretize_vale’. According to the constructor, there is a object called ‘bins’ which has done the ‘init_bins’, I use it to find the interval index directly. I set a list called ‘state’

to store the four features, and use a for loop to find the interval index of each value in ‘observation’, and put them into state, and return state.

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

In this part, it is similar to the same part in taxi.py. However, because there are more dimension in ‘qtable’ than taxi.py, I use the ‘tuple’ to change the form of ‘state’, making it can be the index in ‘qtable’, and find the maximum value’s index.

```
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
    mx = np.max(self.qtable[tuple(next_state)])
    if done:
        mx = 0
    new = (1-self.learning_rate)*self.qtable[tuple(state) + (action, )] + self.
    learning_rate * (reward + self.gamma * mx)
    self.qtable[tuple(state) + (action, )] = new
    # raise NotImplementedError("Not implemented yet.")
    # End your code
    np.save("./Tables/cartpole_table.npy", self.qtable)
```

In this part, it's similar to the same part in taxi.py, and use the same way to deal with the 'qtable' index problem, and if want to find the value of specific state with specific action, use 'tuple(state) + (action,)'. A little different is that if 'done' is one, I set the maximum q value of the next state = 0 to calculate the new q value of this state because the episode ends.

```
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[tuple(state)])
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

In this part, it's similar to the same part in taxi.py, but it can't use state directly because there is no this object in the function component. Therefore, because it need the initial state's maximum q value, I use the function has been done called 'discretize_observation' with 'self.env.reset()' to get the indexes of each feature data in the initial state, and search the maximum value of this state in 'qtable', and return it.

DQN:

```
# Begin your code
# TODO
sample = self.buffer.sample(self.batch_size)
state = torch.tensor(np.array(sample[0]), dtype=torch.float)
action = torch.tensor(sample[1], dtype=torch.long)
reward = torch.tensor(sample[2], dtype=torch.float)
next_state = torch.tensor(np.array(sample[3]), dtype=torch.float)
done = sample[4]

eval = self.evaluate_net(state).gather(1, action.unsqueeze(1))
m = torch.LongTensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
eval = eval + m
next = self.target_net(next_state).detach()
for i in range(self.batch_size):
    if(done[i]==1):
        next[i]= 0
target = reward+ self.gamma * next.max(1).values.unsqueeze(-1)
loss_func = nn.MSELoss()
loss = loss_func(eval.float(), target.float())
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
# raise NotImplementedError("Not implemented yet.")
# End your code
torch.save(self.target_net.state_dict(), "./Tables/DQN.pt")
```

First, I do the ‘sample trajectories of batch size from the replay buffer’. I select some sample randomly from ‘buffer’, which number depends on ‘batch size’. After that, I use ‘torch.tensor’ and ‘np.array’ to change first item of sample into numpy array, and then into pytorch tensor, and components’ type is float, and then put into ‘state’. Next, use the similar way to change components in sample into ‘action’, ‘reward’, and ‘next_state’ respectively(‘action’ and ‘reward’ don’t need ‘np.array’), and ‘done’ equals to fourth item of sample.

Second, I do ‘forward the data to the evaluate net and target net’. I use the function ‘evaluate_net’ with ‘state’ declared before to get the q value of evaluation network now, and according to the value of ‘action.unsqueeze(1)’, select the corresponding components of each column and put the results into ‘eval’, and plus ‘eval’ with ‘m’ which length is 32 and all zero to specify ‘eval’ length.

Next, I do the ‘compute the loss with MSE’. I use ‘target_net’ with ‘next_state’ to evaluate the q value in target network, and separate components by ‘detach()’ and put them into ‘next’. Depending on corresponding index value of ‘done’, if it equals to one, then this index value in ‘next’ equals to zero. I declare ‘target’ which equals to ‘reward’ declared before + ‘self.gamma’(discount factor of this agent) * maximum value in ‘next’, and ‘unsqueeze(-1)’ is used to keep the same dimension in order to calculate with reward easily. I create a function ‘loss_func’ by ‘nn.MSELoss()’, and use

it to calculate the loss between q value of ‘eval’ and ‘target’.

Next, I do the ‘Zero-out the gradients’. I use the function ‘zero_grad()’ to empty ‘optimizer’’s gradients which store in ‘grad’ attribute of tensor.

I do the ‘Backpropagation’ by ‘loss.backward()’ which do the back propagation with the gradient of ‘loss’, which can be used to update the model parameters.

I do the ‘Optimize the loss function’ by function ‘step()’ which updates the parameters of ‘optimizer’ according to calculation.

```
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
        x = torch.unsqueeze(torch.tensor(state, dtype=torch.float), 0)
        p = np.random.uniform(0, 1)
        if p <= self.epsilon:
            val = self.evaluate_net(x)
            action = torch.argmax(val).item()
        else:
            action = env.action_space.sample()
        # raise NotImplementedError("Not implemented yet.")
        # End your code
    return action
```

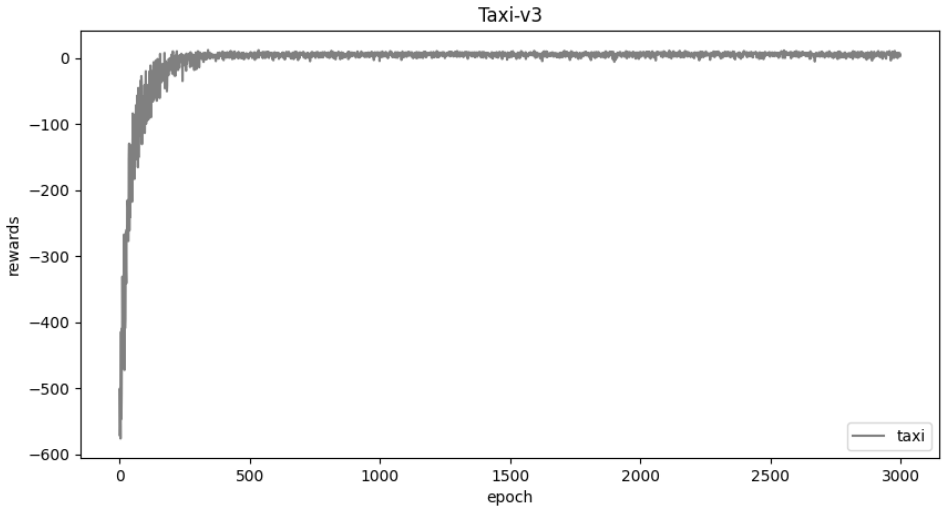
I change ‘state’ into pytorch tensor by function ‘torch.tensor’ and type is torch.float, and use function ‘torch.unsqueeze’ to add one dimension on it(n→(1, n)), and put it into ‘x’. the codes below are the epsilon greedy algorithm which has been implemented in the previous code. If p is smaller than epsilon, then calculate the q value of ‘x’ by ‘evaluate_net’, and select the ‘action’ according to the index of maximum value of ‘val’ after changing ‘val’ into scalar by ‘item()’. If not, just select one action in ‘action_space’ randomly.

```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
    x = torch.unsqueeze(torch.tensor(self.env.reset(), dtype=torch.float), 0)
    with torch.no_grad():
        val = self.target_net(x)
        mx = torch.max(val).item()
    return mx
    # raise NotImplementedError("Not implemented yet.")
    # End your code
```

The way to get ‘x’ is the same to the way in the ‘choose_action’ part. There is only one difference is that because I want the check the initial state max q value, therefore, I use ‘env.reset()’ the get the initial state. After getting ‘x’, calculate the q value by ‘target_net’, and select the maximum value by ‘item()’ and ‘torch.max’ to return. Because we don’t need to consider the gradient, I use ‘torch.no_grad()’ to make the calculation ignore the gradient.

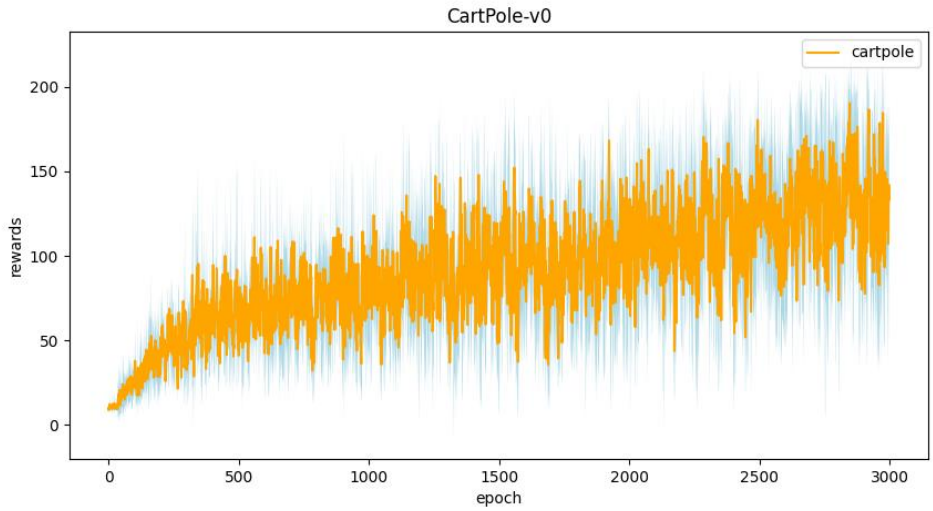
Part ll. Experiment Results:

1. taxi.png



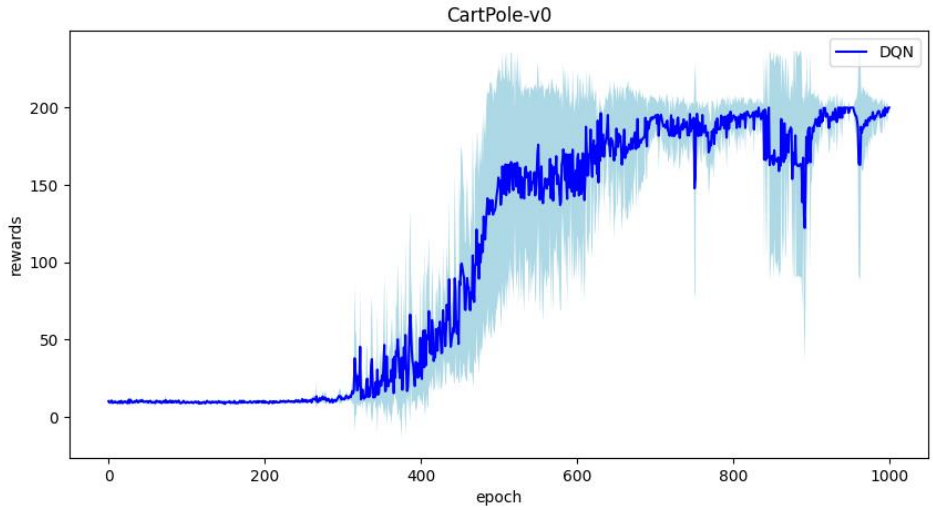
According to this figure, the rewards converge to 0 from -600 when epoch gets larger.

2. cartpole.png



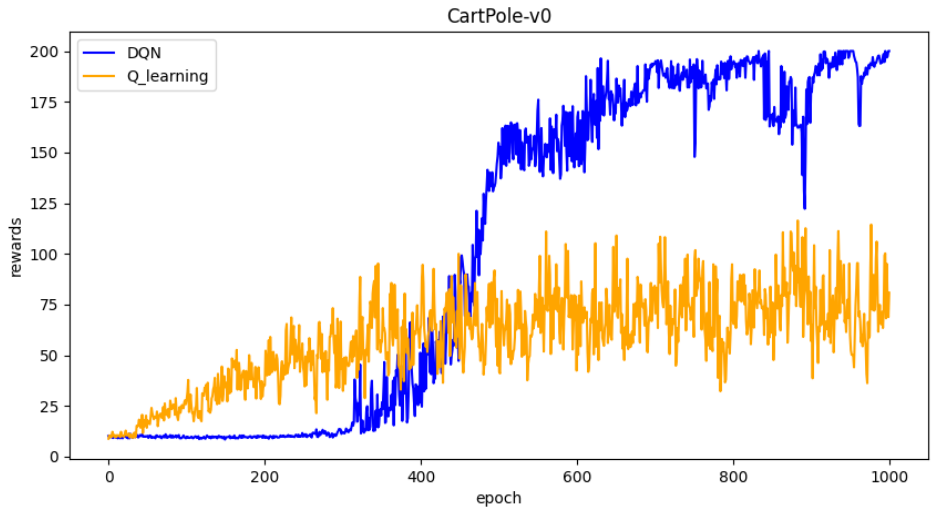
According to this figure, although it has large oscillation, the general trend is increase from near 0 to about 134.

3. DQN.png



According to this figure, the rewards converge from 0 to about 200.

4. compare.png



According to this figure, the increase of rewards of DQN is more than Q learning.

Part lll. Question Answering:

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).

```
average reward: 7.66
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146699999995
```

The optimal Q-value I calculate is 12(we need 8 steps from original point to passenger and from passenger to destination, and +20 when arriving the destination → 20 – 8 = 12). It’s bigger than both average reward and max Q.

2. Calculate the optimal Q-value of the initial state in CartPole-v0, and compare with the Q-value you learned(both cartpole.py and DQN.py). (Please screenshot the result of the “check_max_Q” function to show the Q-value you learned)

```
average reward: 154.75
max Q:30.199447927788864
```

in cartpole.py

```
reward: 199.91
max Q:31.582843780517578
```

in DQN.py

The optimal Q-value I calculate is 200(+1 in every step and limit of episode is 200 for v0). It’s bigger than both average reward and max Q in cartpole.py, and also bigger than both reward and max Q in DQN.py.

3.

a. Why do we need to discretize the observation in Part 2?

Because we want to represent the states during the movement of the cartpole. However, the movement is continuous, which means that the number of states will be infinite, making it impossible to implement. Therefore, we need to discretize it into specific states and define a finite set of states to work with.

b. How do you expect the performance will be if we increase “num_bins”?

I think that the time the train section takes will be longer, but if we plot the rewards, its oscillation will be smaller.

c. Is there any concern if we increase “num_bins”?

Firstly, it's time-consuming because as the number of bins increases, so does the number of states. Secondly, due to the increased number of states, the difference between two adjacent states may become negligible. Therefore, it would be wasteful to calculate between two states that are almost identical.

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

DQN performs better in Cartpole-v0 because it stores the previous experience and select randomly in them to train. This can use the sample more efficiently. Also, since DQN has two neural networks, which can increase the learning stability. However, discretized Q learning only has one Q table, so it will fluctuate more easily.

5.

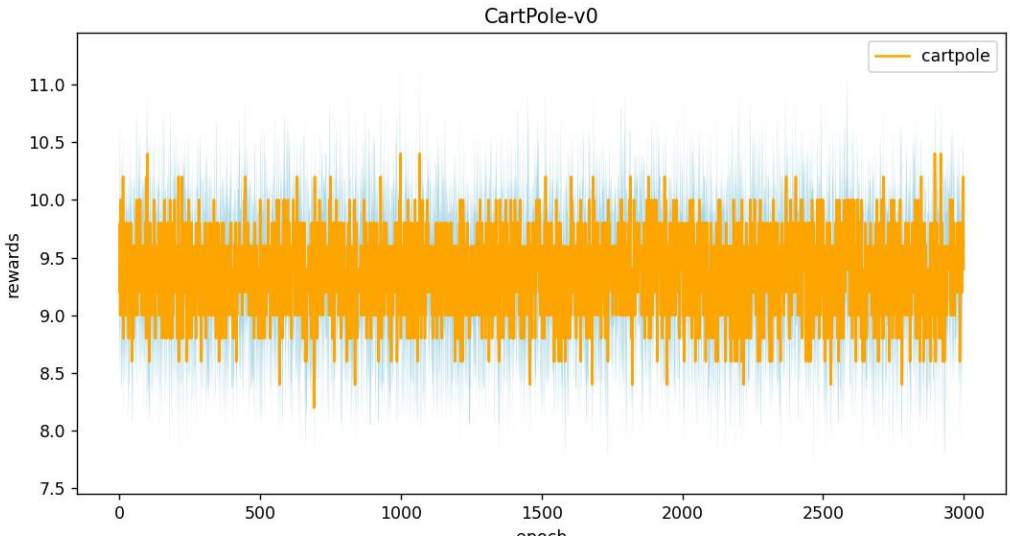
a. What is the purpose of using the epsilon greedy algorithm while choosing an action?

The purpose is to strike a balance between exploitation and exploration. During exploitation, we select the action with the highest reward to maximize the utilization of the available data. In contrast, during exploration, we opt for actions that haven’t been extensively explored or may not be the best, in order to uncover additional possibilities. This ensures that we avoid overly relying on the currently known optimal strategy during the learning process.

b. What will happen, if we don’t use the epsilon greedy algorithm in the

CartPole-v0 environment?

I run the cartpole.py without epsilon greedy, and the result as follow:



We can find that the rewards will stay about 9.5, and no increasing. This is because without epsilon greedy, this agent is lack of exploration, so it won't find another way which may get more rewards, and always uses the currently known optimal strategy.

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

Yes. If we design another reward function or a different exploration strategy, it may be possible. Also, a good initial policy is helpful. Because with these strategies, we are still possible to reach the more valuable action like epsilon greedy algorithm doing.

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

In the training phase, we calculate the best Q-value for states. During the testing phase, the system simply follows the best path we've trained to maximize rewards. It should rely entirely on what we've trained without any exploration. Therefore, there's no need for the epsilon-greedy algorithm.

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

Because we don't update the parameters in the section since we just depend on the currently known q value to choose, we don't need to trace the

gradient. With this one, when we run the code inside it will not care about any data of gradient, which can save the resource and increase the calculation efficiency.