HW4 : Reinforcement Learning

Part I. Implementation

Part1. Taxi

choose_action

```
# Begin your code
"""
To do exporation with deterministic policy, we use eplison-greedy method
we output arbitary action with probabality 1 - eplison,
          action of biggest Q with probabality eplison
"""
if np.random.rand() < 1 - self.epsilon:
    return self.env.action_space.sample()  # Explore: choose a random action
else:
    return np.argmax(self.qtable[state])  # Exploit: choose the action with highest Q-value
# End your code
```

learn

```
# Begin your code
"""
to update Q table , there are 2 cases
First, if the tragectory done, there is no action in the future, thus target = reward
Second, if it haven't done, target = r + gamma * max( Q(s, a') for all a' )
"""
if done:
    target = reward  # If episode is done, only the immediate reward contributes to the target
else:
    target = reward + self.gamma * np.max(self.qtable[next_state])  # Calculate target Q-value
self.qtable[state][action] += self.learning_rate * (target - self.qtable[state][action])  # Update Q-value
# End your code
np.save("./Tables/taxi_table.npy", self.qtable)
```

check_max_Q

```
# Begin your code
"""
get max Q in initial state
"""
return np.max(self.qtable[state])
# End your code
```

Part2. Cartpole

init_bins

```python
# Begin your code
"""
np.linsapce select num_bins numbers from the range lower_bound, upper_bound
choose [-1, 1] to skip the first element 0
"""
return np.linspace(lower_bound, upper_bound, num_bins)[1:-1]
# End your code
```

discretize_value

```python
# Begin your code
"""
np.digitize find the position x of value where bins[x-1] <= value < bins[x],
then it turn float value to integer by returning x
"""
return np.digitize(value, bins)
# End your code
```

discretize_observation

```python
# Begin your code
"""
for each obersvations, cart position, cart velocity, pole angle, tip velocity,
we turns the values in the observations into integer by discretize
"""
state = []
for i in range(len(observation)):
    state.append(self.discretize_value(observation[i], self.bins[i]))
return state
# End your code
```

choose_action

```python
# Begin your code
"""
To do exporation with deterministic policy, we use eplison-greedy method
we output arbitary action with probabality 1 - eplison,
          action of biggest Q with probabality eplison
"""
if np.random.rand() < 1 - self.epsilon:
    return np.random.randint(0, self.env.action_space.n)
else:
    return np.argmax(self.qtable[tuple(state)])
# End your code
```

learn

```python
# Begin your code
"""
to update Q table , there are 2 cases
First, if the tragectory done, there is no action in the future, thus target = reward
Second, if it haven't done, target = r + gamma * max( Q(s, a') for all a' )
"""
if done:
    target = reward
else:
    target = reward + self.gamma * np.max(self.qtable[tuple(next_state)])
self.qtable[tuple(state)][action] += self.learning_rate * (target - self.qtable[tuple(state)][action])
# End your code
```

check_max_Q

```
    # Begin your code
    """
    get choose max value with np.max
    among all actions in qtable of initial states
    """
    return np.max(self.qtable[tuple(self.discretize_observation(self.env.reset()))])
    # End your code
```

Part3. DQN

learn

```
    # Begin your code
    """
    sample (s, a, r, s', done) from buffer
    then turn these tuples to torch tensor

    By Q-learning formula, target = r + gamma * max(Q(s', a')) * (1 - done) for all a'
    we compute loss between target and current_Q by mean square error

    Finally, we update Q by optimizer and backpropagation
    """
    observations, actions, rewards, next_observations, done = self.buffer.sample(self.batch_size)

    # Convert tuple to tensors
    rewards = torch.tensor(rewards, dtype=torch.float32)
    observations = torch.tensor(observations, dtype=torch.float32)
    actions = torch.tensor(actions, dtype=torch.int64)  # Ensure actions are of type int64
    next_observations = torch.tensor(next_observations, dtype=torch.float32)
    done = torch.tensor(done, dtype=torch.float32)

    # Forward pass through the neural networks
    current_Q = self.evaluate_net(observations)
    target_Q = rewards + self.gamma * torch.max(self.target_net(next_observations), dim=1)[0] \
                * (1 - done)

    # Compute the loss
    loss = F.mse_loss(current_Q.gather(1, actions.unsqueeze(1)), target_Q.unsqueeze(1))

    # Zero-out the gradients, backpropagate, and optimize the loss
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    self.count += 1
    # End your code
```

choose action

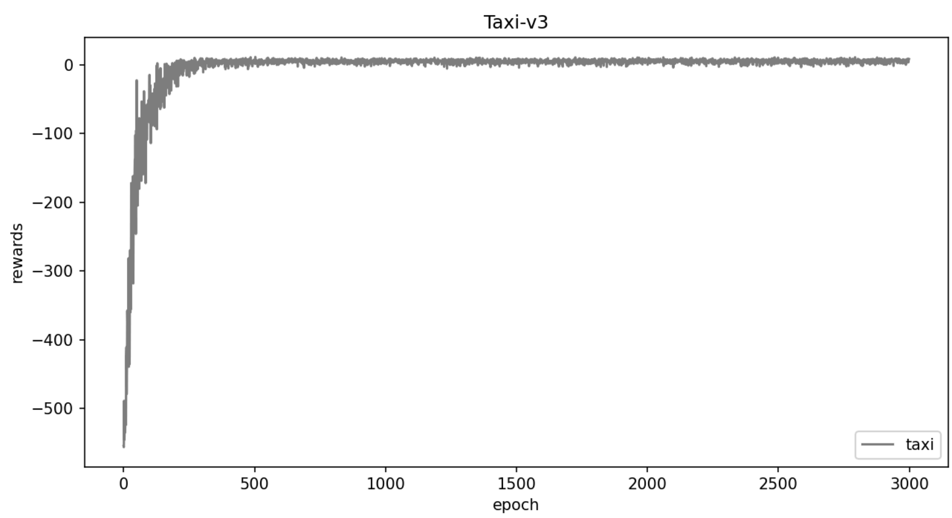
```
    with torch.no_grad():
        # Begin your code
        """
        First, we turn state from tuple to torch tensor
        Then, to do exporation with deterministic policy, we use eplison-greedy method
        we output arbitary action with probabality 1 - eplison,
                action of biggest Q with probabality eplison
        """
        # Convert the state to the same data type as the weights
        state_tensor = torch.tensor(state, dtype=torch.float32)
        # Forward pass through the neural network
        q_values = self.evaluate_net(state_tensor)
        # Choose the action with epsilon-greedy policy
        if np.random.rand() < 1 - self.epsilon:
            action = self.env.action_space.sample()
        else:
            action = int(torch.argmax(q_values).item())
        # End your code
```

check_max_Q

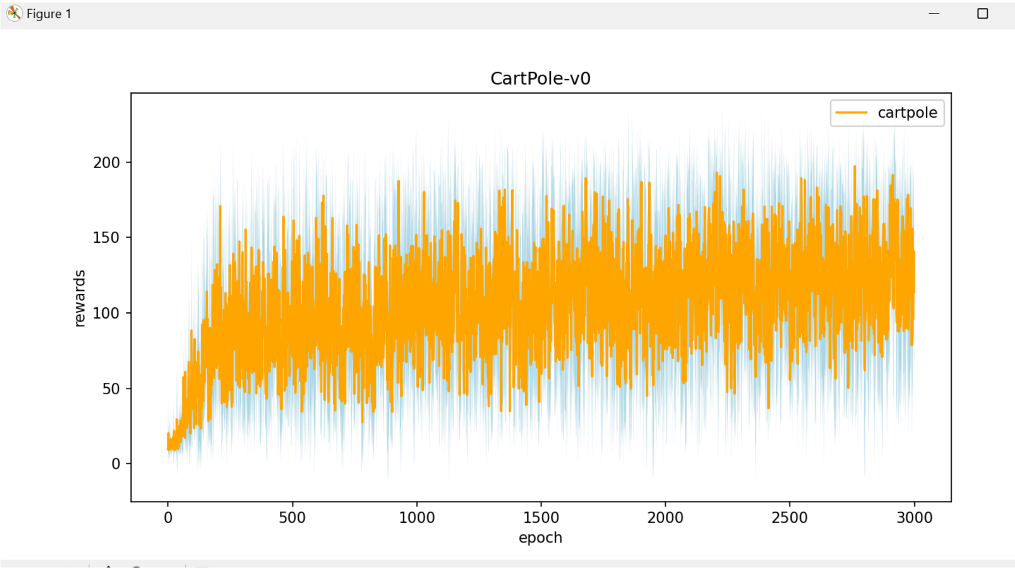
```
"""
First, turn intial state to torch tensor
Then, choose max Q(s0, a) for all actoin a using target network
"""
s0 = self.env.reset()
s0_tensor = torch.FloatTensor(s0).unsqueeze(0)
with torch.no_grad():
    q_values = self.target_net(s0_tensor)
    max_q = float(torch.max(q_values).item())
return max_q
# End your code
```

Part II. Experiment Results

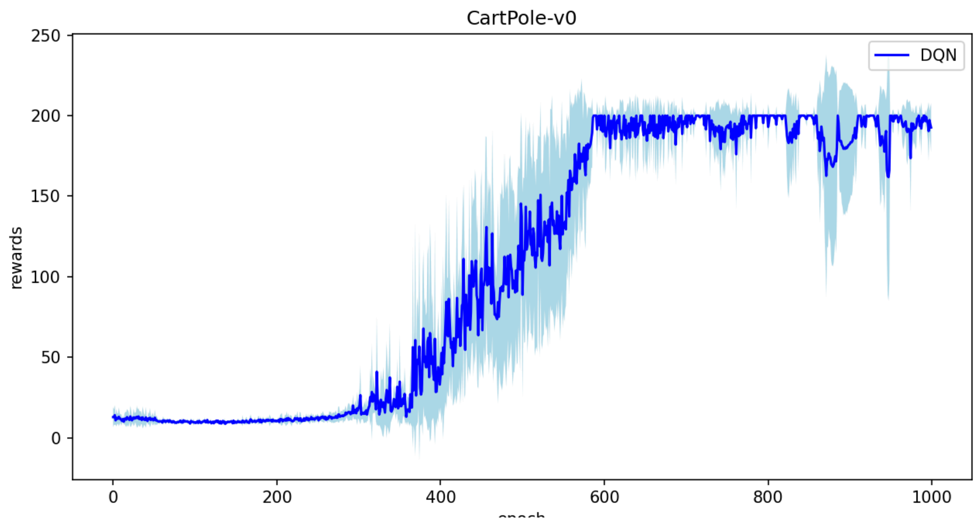
taxi.png



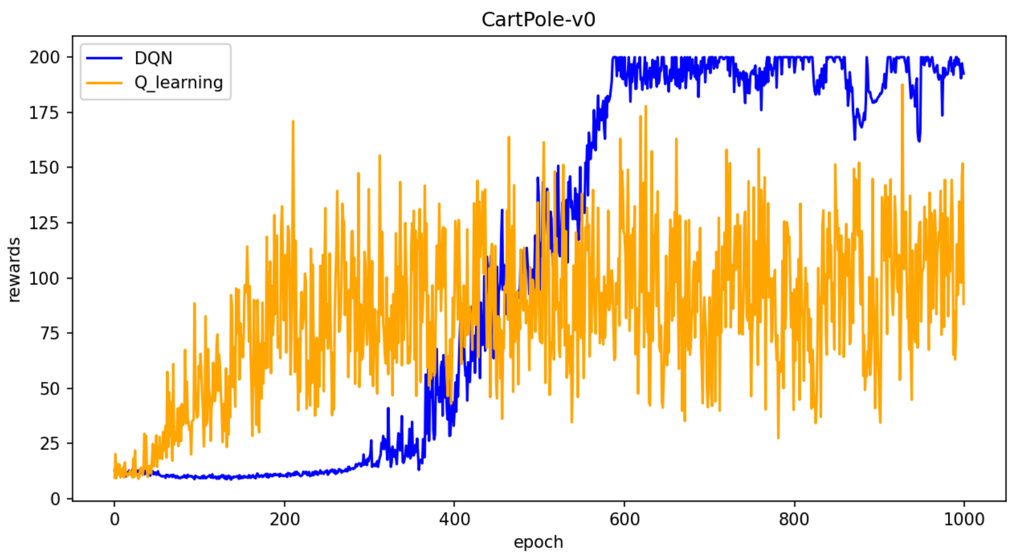
cartpole.png



DQN.png



compare.png



Part III. Question Answering

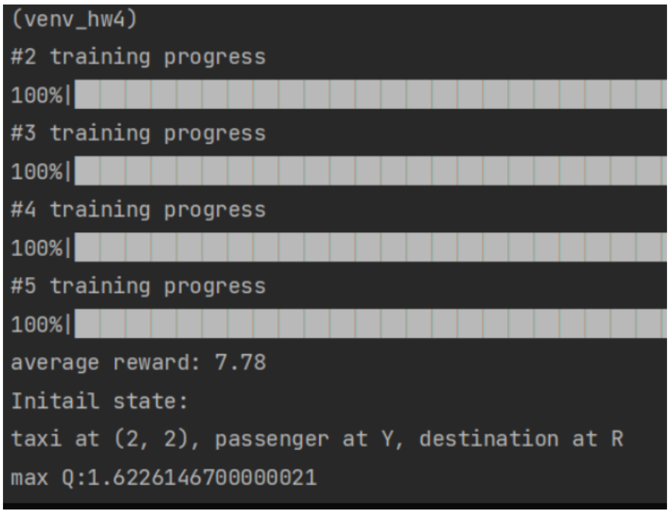
1. Calculate the optimal Q-value of a given state in Taxi-v3, and compare with the Q-value you learned

Given that, Y is at (0, 0), R is at (0, 4), the initial state is (2, 2).
In the optimal policy, it takes 5 states to reach passenger, 4 states to from pickup to destination
Besides, when we haven't deliver passenger successfully, reward remains -1 from t = 0 to 8

$$Therfore,$$
$$Q^*(s,a) = \ optimal\ return = \sum_{t=0}^{8} \gamma^t * r_t + \gamma^9 * r_9$$
$$= (-1) * \frac{1-\gamma^9}{1-\gamma} + 20 * \gamma^9 \quad where\ \gamma is 0.9 \quad = \ 1.62261467$$

which is similar to what I learned (1.62261467)

check max Q result

<table>
<tr><th>Map</th><th>check max Q</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

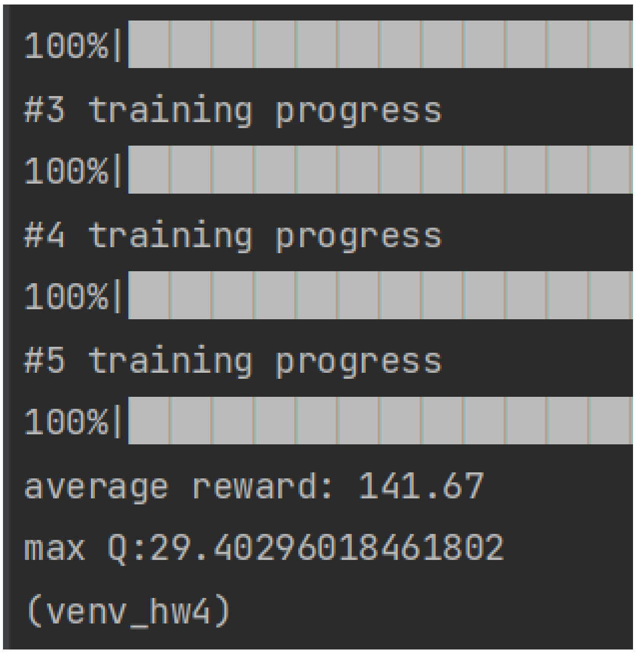
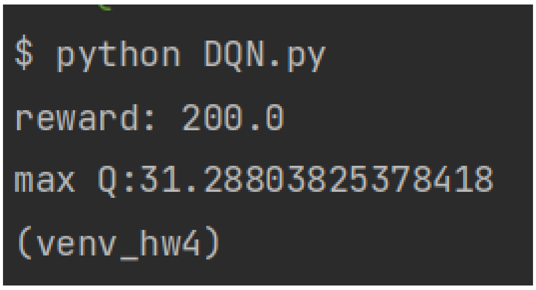
2. Calculate the optimal Q-value of the initial state in CartPole-v0, and compare with the Q-value you learned (both cartpole.py (http://cartpole.py) and DQN.py (http://DQN.py))

In the optimal policy, we could maintain cartpole to be upright in the whole trajectory.

$$Therfore,$$
$$Q^*(s,a) = \ optimal\ return = \sum_{t=0}^{199} \gamma^t * r_t$$
$$= 1 * \frac{1-\gamma^{201}}{1-\gamma} \quad where\ \gamma\ \ is\ 0.97 \quad = \ 33.26021987$$

which is larger than what I've learned in cartpole (29.40) and DQN (31.28)

check max Q result

[cartpole.py (http://cartpole.py)](http://cartpole.py)	[DQN.py (http://DQN.py)](http://DQN.py)
	

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

If we allow the observation to be continuous (not discrete), there will be infinite states, that is, it is time-comsuming to update table for all states.
Moreover, if we couldn't update all states, Q-learning is hard to converge due to stochastic approximation theorem (the convergence condition of Q-learning)

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

if we increase num_bins, the value of states will be more accurate. Therefore, the performance will be better in the long run

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

By SA (stochastic approximation) theorem, Q-learning converge if all state action pair are explored infinitely many times. However, if we increase num_bins, we increase the number of states simultaneously.
Therefore, it will take more time for model to converge

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

Compare.png exhibit that DQN performs better in Cartpole-v0.

It is because in in DQN, model learn the observation by embedding. The embedding is updated from the environment.

While in discretized Q learning, we learn the observation by predifined bins. The predifined bins is set by human.

Therefore in DQN, we has a better performance because DQN has more accurate state.

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

Q-learning is deterministic policy, that is, it usually choose the action with max Q.
If we didn't do exploration when selecting actoins, it will stuck at some state action pair, while it is not the optimal policy.
If we use epsilon greedy algorithm, we could explore more state, action pair. Then we may converge to optimal Q due to SA convergence theorem.

5.b. What will happen, if we don't use the epsilon greedy algorithm in the CartPole-v0 environment?

If we didn't do exploration when selecting actoins, it will stuck at some state action pair, while it is not the optimal policy.
Therefore, by SA convergence theorem, it will not converge to optimal policy

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

Yes, it is possible. We could use other exploration ways like bootstrapping DQN to achieve the same performance.
In eplison greedy, we may choose one-step action with random. However, choosing one-step action couldn't do "deep exploration".
In contrast, bootstrapping DQN choose random Q function to perform actions. It may do deeper exploration than eplison greedy.

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

When doing testing, we want our action choose the best action but not choosing it randomly. We don't need to do exploration in this time.

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

When choosing action, we don't want to add computation into backpropagation, so we use torch.no_grad() to prevent it from adding computation.