Part I. Implementation

Part 1: Q learning in Taxi-v3

choose_action(self, state)

```
"""
Choose an action a in the current world state (s)
First we randomize a number
"""
exp_exp_tradeoff = random.uniform(0,1)

"""
If this number > greater than epsilon --> exploitation (taking the biggest Q value for this state)
"""
if exp_exp_tradeoff > self.epsilon:
    action = np.argmax(self.qtable[state,:]) # get max index from (state + 1) row
# Else doing a random choice --> exploration
else:
    action = env.action_space.sample()

return action
# End your code
```

learn(self, state, action, reward, next_state, done)

```
'''
Update Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
qtable[new_state,:] : all the actions we can take from new state
'''
self.qtable[state, action] = self.qtable[state, action] + self.learning_rate * (reward + self.gamma * np.max(self.qtable[next_state,:]) - self.qtable[state, action])
```

check_max_Q(self, state)

```
"""
return max Q-value of given state
"""
max_q = np.max(self.qtable[state,:])

return max_q
```

Part 2: Q learning in CartPole-v0

init_bins(self, lower_bound, upper_bound,num_bins)

```
    # slice the interval with given lower_bound, upper_bound and num_bins
    bins_arr = np.linspace(lower_bound,upper_bound, num_bins, endpoint=False)
    bins_arr = np.delete(bins_arr,0)
    return bins_arr
    # End your code
```

discretize_value(self, value, bins)

```
        # Discretize the value with given bins.
        return np.digitize(value,bins)
```

discretize_observation(self, observation)

```
# Discretize the observation, using the function discretize_value() to get the discretized data of the 4 feature in observation and store in state list
state = []
for i in  range(len(observation)):
    state.append(self.discretize_value(observation[i], self.bins[i]))
return tuple(state)
# End your code
```

choose_action(self, state)

```
    # generate a random number between 0 and 1. If it's bigger than epsilon, return the maximum index in Q_table; otherwise, return ramdom action
    if np.random.uniform(0,1) < self.epsilon:
        return env.action_space.sample()
    else:
        return np.argmax(self.qtable[state])
    # End your code
```

learn(self, state, action, reward, next_state, done)

```
# based on the formula of Q_learning
estimated_optimal_Q = self.qtable[state + (action, )]
next_max = max(self.qtable[next_state])
if done: next_max = 0
self.qtable[state + (action, )] = (1 - self.learning_rate) * estimated_optimal_Q + self.learning_rate * (reward + self.gamma * next_max)
# End your code
```

check_max_Q(self)

```
        # return the max Q_value of given state, the initial state should also be discretized
        return max(self.qtable[self.discretize_observation(self.env.reset())])
        # End your code
```

Part 3: DQN in CartPole-v0

learn(self)

```
    # 2. Sample trajectories of batch size from the replay buffer, use function "sample" in class "replay_buffer" to get sample data.
    # convert the data into tensor form
    sample = self.buffer.sample(self.batch_size)
    states = torch.tensor(np.array(sample[0]), dtype=torch.float)
    actions = torch.tensor(sample[1], dtype=torch.long).unsqueeze(1)
    rewards = torch.tensor(sample[2], dtype=torch.float)
    next_states = torch.tensor(np.array(sample[3]), dtype=torch.float)
    dones = torch.tensor(sample[4], dtype=torch.bool)

    # 3. Forward the data to the evaluate net and the target net.
    # q_eval is predicted value from evaluate network which is based on "action"
    # q_next is actual value from target.
    # q_target is expected value from formula "reward + gamma + max(q_next)".
    q_eval = self.evaluate_net(states).gather(1, actions)
    q_next = self.target_net(next_states).detach() * (~dones).unsqueeze(-1)
    q_target = rewards.unsqueeze(-1) + self.gamma * q_next.max(1)[0].view(self.batch_size, 1)

    # 4. Compute the loss of q_eval and q_target with MSE.

    loss_func = nn.MSELoss()
    loss = loss_func(q_eval.float(), q_target.float())

    # 5. Zero-out the gradients before backpropagation, otherwise the gradient would be combination of
    # previous gradient.
    self.optimizer.zero_grad()

    # 6. Do backpropagation.
    loss.backward()

    # 7. Optimize the loss function.
    self.optimizer.step()
```

choose_action(self, state)

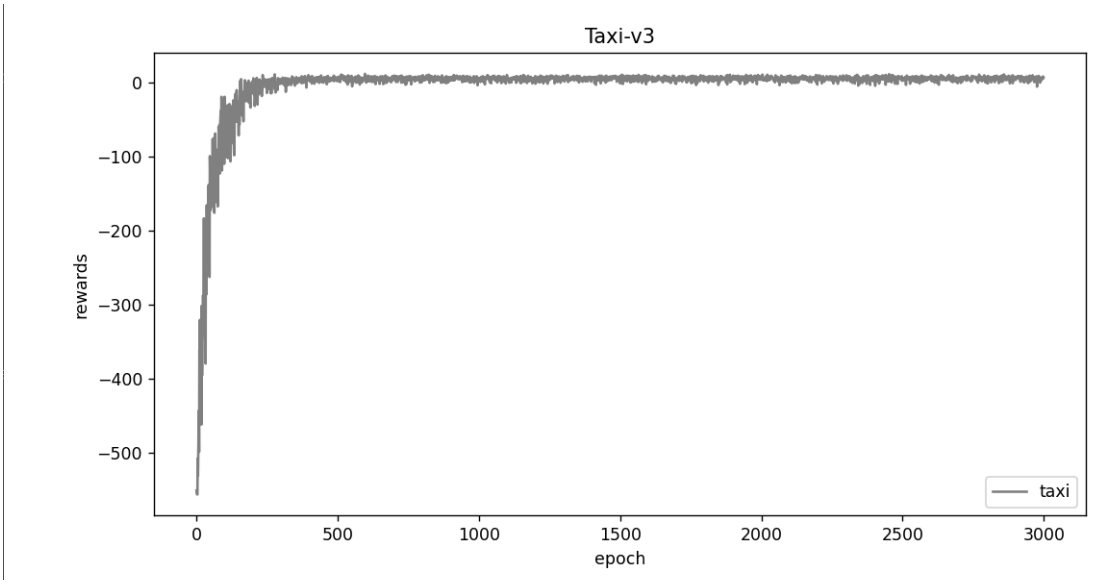
```
    # generate a random number between 0 and 1. If it's bigger than epsilon, return the maximum index of the given state forwarded by evaluate neural network;
    # otherwise, return ramdom action
    if np.random.uniform(0,1) < self.epsilon:
        action = env.action_space.sample()
    else:
        action = torch.argmax(self.evaluate_net(torch.unsqueeze(torch.tensor(state, dtype=torch.float), 0))).item()
    # End your code
turn action
```

check_max_Q(self)

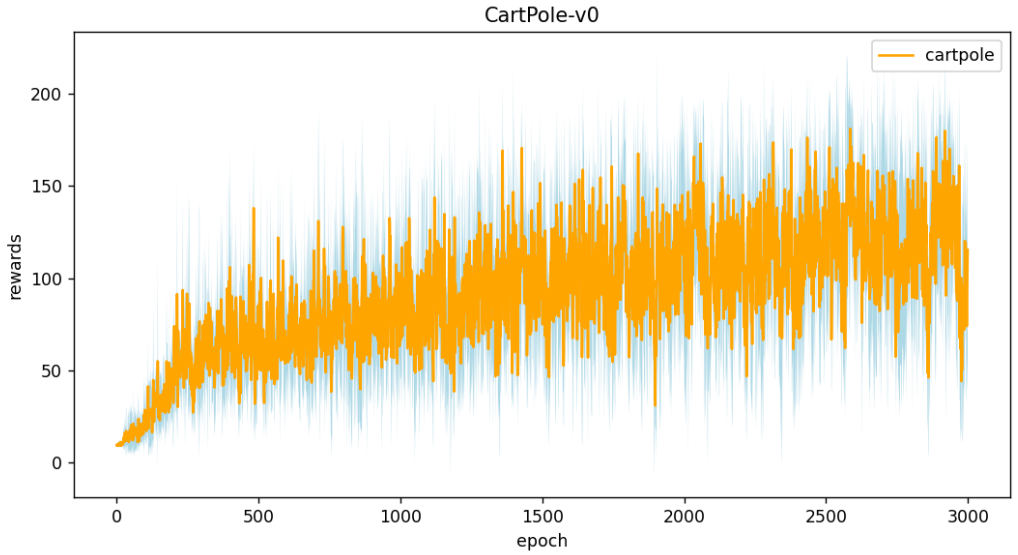
```
    # convert initial state into tensor -> forward the tensor in neural network
    # find max Q-value and return
    x = torch.unsqueeze(torch.tensor(self.env.reset(), dtype=torch.float), 0)
    return torch.max(self.target_net(x)).item()
    # End your code
```

Part II. Experiment Results:

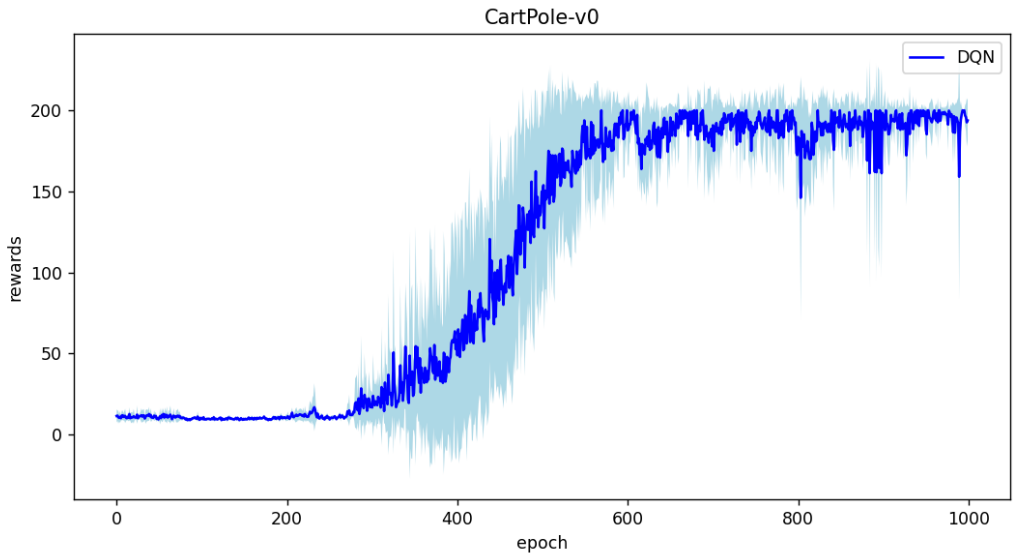
1. taxi.png:



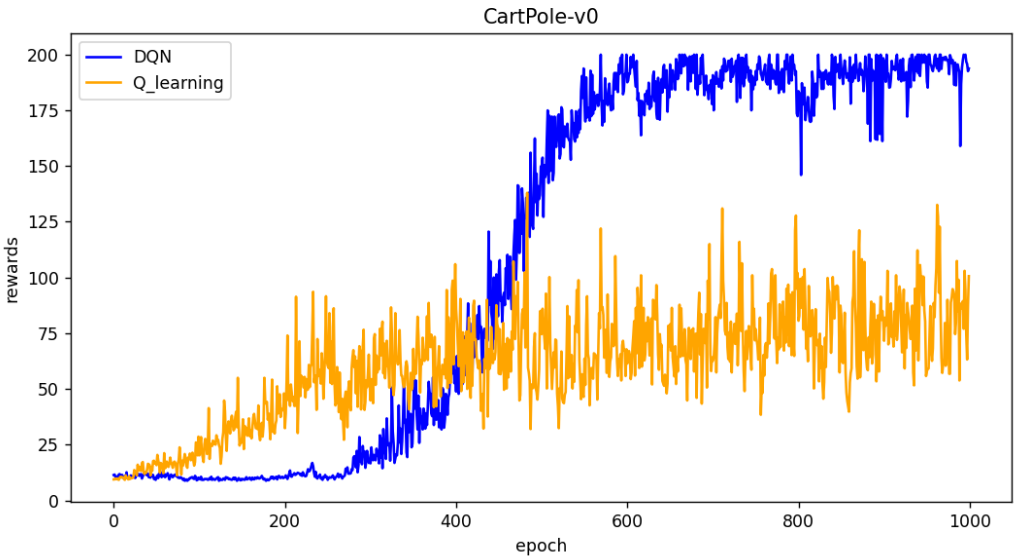
2. cartpole.png



3. DQN.png



4. 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 (Please screenshot the result of the "check_max_Q" function to show the Q-value you learned).

Since the initial state is at (2,2), and we have to send passengers from Y to R. The optimal solution needs 9 steps of -1 and 1 step of 20 rewards.

Optimal Q-value: (-1) * (1-γ^nums_step) / (1-γ) + 20 * γ^nums_step

=> $\frac{-1 \cdot \left(1-0.9^{9}\right)}{1-0.9}+20 \cdot (0.9)^{9}$

=> 1.62261647

```
#1 training progress
100%| 3000/3000 [00:00<00:00, 6070.85it/s
#2 training progress
100%| 3000/3000 [00:00<00:00, 5954.66it/s
#3 training progress
100%| 3000/3000 [00:03<00:00, 803.34it/s
#4 training progress
100%| 3000/3000 [00:00<00:00, 5943.63it/s
#5 training progress
100%| 3000/3000 [00:00<00:00, 5916.39it/s
average reward: 7.88
Initail state:
taxi at (2, 2), passenger at Y, destination at R
max Q:1.6226146700000021
```

which is close to the max Q.

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)

Optimal max Q-value: (1-γ^average_reward) / (1 - γ)

=> $\frac{\left(1-0.97^{132.64}\right)}{1-0.97}$

=> 32.74681...

Q-learning

DQN

DQN is closer than Q-learning to optimal max Q-value.

3.

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

To simplify the state representation and also allow us to build up the look-up table.

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

Better, since the discrete data will become more similar to observation.

c.Is there any concern if we increase "num_bins"?

Training may take more time and also cost more space for Q-table.

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

DQN tends to perform better.
DQN uses a deep neural network to approximate the Q-function, which can learn a more accurate and robust representation of the state-action values compared to a tabular Q-table used in discretized Q-learning. This allows DQN to handle large and complex state spaces more effectively than discretized Q-learning

5.

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

To balance exploration and exploitation, which leads to better reward.

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

Agents can't explore the environment sufficiently to learn about the unknown state-action values while also can't exploit the current knowledge to maximize its rewards. This don't allow the agent to learn an effective policy that balances exploration and exploitation and can't solve the task of balancing the pole on the cart

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

Yes, since there are other alternative algorithms to achieve it.
Ex: Boltzmann exploration strategy, deterministic exploration strategy.

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

Agent has been well-trained, thus no need to gain information from the environment.

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

It is used to disable gradient computation during action selection, which can save memory and computation time and prevent any unwanted gradient updates to the neural network.

.