Notebook 10 - Reinforcement Learning / Self-Driving Cab

CSI4106 Artificial Intelligence
Fall 2020
Prepared by Julian Templeton and Caroline Barrière

+ Code + Text

INTRODUCTION:
In this notebook we will be exploring the use of Reinforcment Learning to help allow an agent solve a specific task in an environment provided by OpenAI's Gym library. This library provides a number of environments that we can train an AI to master. Within this notebook we will be exploring a scenario in which a taxi located on a grid must be controlled by an agent to pickup a passenger located in one of four positions and drop the passenger off in one of three other positions.

To familiarize yourself with the Self-Driving Cab problem tackled in this notebook, please go to the site https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/ and read section 1 (rewards), section 2 (state space) which will make you understand why there are 500 possible states, section 3 (action space) which describes the possible actions.

Throughout the notebook we will be working with a Baseline approach and a Q-Learning-based approach. This will provide insight into how Q-Learning can be applied to problems and how an agent can use Reinforcment Learning to solve problems in an environment.

When submitting this notebook, ensure that you do NOT reset the outputs from running the code (plus remember to save the notebook with ctrl+s).

In order to keep the installation easy, you will be once again running this notebook in Google Colab, NOT on your local machine.

HOMEWORK:
Go through the notebook by running each cell, one at a time.
Look for **(TO DO)** for the tasks that you need to perform. Do not edit the code outside of the questions which you are asked to answer unless specifically asked. Once you're done, Sign the notebook (at the end of the notebook), and submit it.

The notebook will be marked on 30.
*Each **(TO DO)** has a number of points associated with it.*

1.0 - Setting up the Taxi Game

To begin the notebook, we will need to set up and explore the environment that our agent will be working with. OpenAI's Gym provides many different experiments to use. These range from balancing acts to self driving cars to playing a simple Atari game. Unfortunately, not every option available to us can be easily worked with. Many can take hours of training to start seeing some exciting results. Each of these experiments use agents that can be trained by Reinforcment Learning to master how to perform the specified task. The methods used can range from the simple use of Q-Learning to the more complex use of one or more Deep Learning models that work in conjunction with Reinforcement Learning techniques.

One simple, yet interesting, experiment involves an AI controlled taxi that must pick up and dropoff a passenger. This is the problem that we will be exploring throughout the notebook. The code used throughout the notebook comes from this example and has been modified accordingly.

To start, we will install some of the packages that we will need to run the progam.

```
# Install the necessary libraries
!pip install cmake 'gym[atari]' scipy
```

```
Requirement already satisfied: cmake in /usr/local/lib/python3.6/dist-packages (3.12.0)
Requirement already satisfied: gym[atari] in /usr/local/lib/python3.6/dist-packages (0.17.3)
Requirement already satisfied: scipy in /usr/local/lib/python3.6/dist-packages (1.4.1)
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /usr/local/lib/python3.6/dist-packages (from gym[atari]) (1.5.0
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /usr/local/lib/python3.6/dist-packages (from gym[atari]) (1
Requirement already satisfied: numpy>=1.10.4 in /usr/local/lib/python3.6/dist-packages (from gym[atari]) (1.18.5)
Requirement already satisfied: atari-py~=0.2.0; extra == "atari" in /usr/local/lib/python3.6/dist-packages (from gym[at
Requirement already satisfied: Pillow; extra == "atari" in /usr/local/lib/python3.6/dist-packages (from gym[atari]) (7.
Requirement already satisfied: opencv-python; extra == "atari" in /usr/local/lib/python3.6/dist-packages (from gym[atar
Requirement already satisfied: future in /usr/local/lib/python3.6/dist-packages (from pyglet<=1.5.0,>=1.4.0->gym[atari]
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from atari-py~=0.2.0; extra == "atari"->g
```

```
# Import the necessary libraries
import random
import gym
import numpy as np
from IPython.display import clear_output
```

```
from IPython.display import clear_output
```

With all of the libraries installed, we will now make use of the Taxi program provided by Gym. Below we will import Gym, load the program as the active environment, and render an image representing the current state of the program.

From the image seen below, there are four different key locations in the environment, represented by *R*, *G*, *B*, and *Y*. The letter with that is bolded in blue represents where the current passenger needs to get picked up and the letter bolded in purple represents where the passenger wants to dropped off. The yellow block represents the cell which the taxi cab is currently located at. Therefore, the taxi cab must first pick up the passenger and drop them off at the dropoff location. When a passenger is in the taxi, it turns green until the passenger is dropped off.

```
# Load the environment
env = gym.make("Taxi-v3").env
# Render the current state of the program
env.render()
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

Next we will reset the state of the environment and re-render the current state. We also print the total number of actions available to our agent (defined as the *Action Space*) and the *State Space* which represents the state of the program (where is the cab, the passenger, pickup location and dropoff location).

```
env.reset() # reset environment to a new, random state
env.render()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+

Action Space Discrete(6)
State Space Discrete(500)
```

Intuitively, we want our agent to learn which action to take given a specific state. Specifically, which action should be taken based on where the taxi cab is located in relation to the passenger location and drop off location. The six possible actions that the taxi can take at a given time step are:

Action = 0: Head south

Action = 1: Head north

Action = 2: Head east

Action = 3: Head west

Action = 4: Pickup

Action = 5: Dropoff

Below is an example of setting the state to a specific encoding and rendering that state.

```
# The encoding below represents: (taxi row, taxi column, passenger index, destination index)
state = env.encode(3, 1, 2, 0)
print("State:", state)

env.s = state
env.render()
```

```
State: 328
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

The following example showcases how a state with the passenger within the taxi can be set.

```
# The encoding below represents: (taxi row, taxi column, passenger index, destination index)
state = env.encode(0, 1, 4, 0)
print("State:", state)

env.s = state
env.render()
```

```
State: 36
+---------+
|R:_| : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

(TO DO) Q1

Now that we have seen how to set a state via an encoding, you will need to set the state to match the descriptions below and render them.

a) Set the passenger to be at position G, with the passenger wanting to be dropped off at position R, and the taxi positioned at a random point on the grid (the selected position of the taxi must be selected randomly). After setting the position, render the state.

b) Set the passenger to be in the taxi (at any position without a letter on it) and set the passenger dropoff point to be position B. After setting the position, render the state.

(TO DO) Q1 (a) - 2 marks

a) Set the passenger to be at position G, with the passenger wanting to be dropped off at position R, and the taxi positioned at a random point on the grid (the selected position of the taxi must be selected randomly). After setting the position, render the state.

```
import random
env.reset
row = random.randint(0, 4)
col = random.randint(0,4)
print(row)
print(col)
state1a = env.encode(row, col, 1, 0)
print("State:", state1a)

env.s = state1a
env.render()
```

```
1
4
State: 184
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

(TO DO) Q1 (b) - 2 marks

b) Set the passenger to be in the taxi (at any position without a letter on it) and set the passenger dropoff point to be position B. After setting the position, render the state.

```
state1b = env.encode(2, 3, 4, 3)
print("State:", state1b)

env.s = state1b
env.render()
```

```
State: 279
+---------+
|R: | : :G|
| : | : : |
| : : :_: |
| | : | : |
|Y| : |B: |
+---------+
```

For every action that the taxi can take, we have a list representing the key information with respect to what will happen when an action is performed. After performing an action, the agent will receive a reward or a penalty. This reward or penalty will tell the agent how good or bad their decision to perform the specified action was.

Below we display a dictionary that contains all possible actions along with the following information within the corresponding tuples:

(
The probability of taking that action,
The resulting state after taking that action,
The reward for taking that action,
Whether or not the program will end when performing the action
)

Example tuple: (1.0, 328, -1, False)

```
env.P[328]
```

```
{0: [(1.0, 428, -1, False)],
 1: [(1.0, 228, -1, False)],
 2: [(1.0, 348, -1, False)],
 3: [(1.0, 328, -1, False)],
 4: [(1.0, 328, -10, False)],
 5: [(1.0, 328, -10, False)]}
```

Although not displayed by the code above, if the taxi is holding the passenger and is over the dropoff point, the reward for the dropoff action is 20.

2.0 - Baseline Approach to the Taxi Game

To start, we will perform the simulation of the taxi cab scenario with a baseline approach that does not use Q-Learning. This approach will simply work by selecting a random available action at each time step, regardless of the current state. We will also prepare a method of playing through all frames within an episode to view how the agent controls the taxi in the scenario.

```
def run_single_simulation_baseline(env, state, disable_prints=False):
    '''
    Given the environment and a specific state, randomly select an action for the taxi
    to perform until the goal is completed.
    '''
    if not disable_prints:
        print("Testing for simulation: {}".format(state))
    # Set the state of the environment
    env.s = state
    # Used to hold all information for a single time step (including the image data)
    frames = []
    # Used to determine when the simulation has been completed
    done = False
    # Determines the number of times steps that the application has been run for
    time_steps = 0
    # The total values used to determine how many times the agent mistakenly
    # picks up no one or attempts to dropoff no passenger or attempts to
    # dropoff a passenger in the wrong position.
    penalties, reward = 0, 0
    # Run until the passenger has been picked up and dropped off in the target location
    while not done:
        # Perform a random action from the set of available actions in the environment
        action = env.action_space.sample()
        # From performing the action, retrieve the new state, the reward from taking the action,
        # whether the simulation is complete, and other information from performing the action.
        state, reward, done, info = env.step(action)
        # If an incorrect dropoff or pickup is performed, increment the penalty count
        if reward == -10:
            penalties += 1
        # Put each rendered frame into dict to use for animating the process and
        # tracking the details over the run
        frames.append({
            'frame': env.render(mode='ansi'),
            'state': state,
            'action': action,
            'reward': reward
            }
        )
        # Increment the time step count
        time_steps += 1
    # State the total number of steps taken and the total penalties that have occured.
    if not disable_prints:
        print("Timesteps taken: {}".format(time_steps))
```

```
            print("Penalties incurred: {}".format(penalties))
        # Return the frame data, the total penalties, and the total time steps
        return frames, penalties, time_steps
```

With the baseline approach defined, we will run a test with this approach to see how long it takes an agent using this approach to find a solution for simulation 328 and how many major penalties the agent receives.

```
state = 328
# Run a test and collect all frames from the run
frames, _, _ = run_single_simulation_baseline(env, state)
```

```
Testing for simulation: 328
Timesteps taken: 1795
Penalties incurred: 618
```

After performing a simulation and retrieving the results, we can use the frames obtained from the simulation and pass it to the *print_frames* function below to display an animation containing all frames along with the information that was from each time step that the frame corresponds to.

For the first episode that you view, it is recommended to run through the entire process at a slower speed (such as 0.3 or 0.5 in the sleep call). However you are free to increase the speed of the process by reducing the number in the sleep function call in the *print_frames* function below.

```
from IPython.display import clear_output
from time import sleep

def print_frames(frames):
    '''
    For each frame, show the frame and display the timestep it occurred at,
    the number of the active state, the action selected, adn the corresponding reward.
    '''
    for i, frame in enumerate(frames):
        clear_output(wait=True)
        print(frame['frame'])
        print(f"Timestep: {i + 1}")
        print(f"State: {frame['state']}")
        print(f"Action: {frame['action']}")
        print(f"Reward: {frame['reward']}")
        # Can adjust speed here
        sleep(.1)
# Print the frames from the episode
print_frames(frames)
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)

Timestep: 1795
State: 0
Action: 5
Reward: 20
```

(TO DO) Q2

a) Using the state defined from Q1 (a), retrieve the corresponding frames obtained from using the baseline approach above. Then display those frames.

b) Using the state defined from Q1 (b), retrieve the corresponding frames obtained from using the baseline approach above. Then display those frames.

(TO DO) Q2 (a) - 2 marks

a) Using the state defined from Q1 (a), retrieve the corresponding frames obtained from using the baseline approach above. Then display those frames.

```
frames1a, _, _ = run_single_simulation_baseline(env, state1a)
print_frames(frames1a)
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
```

```
|Y| : |B: |
+---------+
  (Dropoff)

Timestep: 2487
State: 0
Action: 5
Reward: 20
```

(TO DO) Q2 (b) - 2 marks

b) Using the state defined from Q1 (b), retrieve the corresponding frames obtained from using the baseline approach above. Then display those frames.

```
frames1b, _, _ = run_single_simulation_baseline(env, state1b)
print_frames(frames1b)
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)

Timestep: 778
State: 475
Action: 5
Reward: 20
```

With the ability to simulate single runs of an episode with the Baseline approach, we will now define a function that we will use to evaluate the general performance of the baseline model when averaged over many episodes. The *evaluate_agent_baseline* function below accepts as input the total number of randomly selected episodes to run along with the environment, runs the random episodes, displays the average amount of timesteps taken per episode along with the average penalties incurred, and returns the frame data.

If the evaluate_agent_baseline function ever seems to be running for far too long (several minutes, not just one), stop the run by clicking the button at the top-left of the code cell being executed and run it again.

```
def evaluate_agent_baseline(episodes, env):
    '''
    Given a number of episodes and an environment, run the specified
    number of episodes, where each run begins with a random state, display the
    naverage timesteps per episode and the average penalties per episode, and output
    the frames to be displayed.
    '''
    total_time_steps, total_penalties = 0, 0
    frames = []
    # Run through the total number of episodes
    for _ in range(episodes):
        # Get a random state
        state = env.reset()
        # Run the simulation, obtaining the results
        frame_data, penalties, time_steps = run_single_simulation_baseline(env, state, True)
        # Update the tracked data over all simulations
        total_penalties += penalties
        total_time_steps += time_steps
        frames = frames + frame_data
    print(f"Results after {episodes} episodes:")
    print(f"Average timesteps per episode: {total_time_steps / episodes}")
    print(f"Average penalties per episode: {total_penalties / episodes}")
    return frames
```

(TO DO) Q3

a) Use the *evaluate_agent_baseline* function defined above to run through 100 random episodes for the environment.

b) From the output seen from Q3 (a), how did the Baseline approach do and why do you think that it performed well or poorly? Explain with respect to the average timesteps per episode and the average penalties per episode.

c) Without moving to a Reinforcment Learning approach how can the Baseline approach be modified to perform slightly better?

(TO DO) Q3 (a) - 1 mark

a) Use the *evaluate_agent_baseline* function defined above to run through 100 random episodes for the environment.

```
# episodes3a = []
# for i in range(0,99):
#   row = random.randint(0, 4)
#   col = random.randint(0,4)
```

```
    #   origin = random.randint(0,4)
    #   des = random.randint(0,3)
    #   while(origin==des):
    #     des= random.randint(0,3)
    #   state3a = env.encode(row, col, origin, des)
    #   episodes3a.append(state3a)
    evaluate_agent_baseline(100, env)
```

```
     Results after 100 episodes:
     Average timesteps per episode: 2406.42
     Average penalties per episode: 780.26
     [{'action': 5,
       'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : |\x1b[43m \x1b[0m: |\n|\x1b[34;
       'reward': -10,
       'state': 368},
      {'action': 2,
       'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | :\x1b[43m \x1b[0m|\n|\x1b[34;
       'reward': -1,
       'state': 388},
      {'action': 3,
       'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : |\x1b[43m \x1b[0m: |\n|\x1b[34;
       'reward': -1,
       'state': 368},
      {'action': 2,
       'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | :\x1b[43m \x1b[0m|\n|\x1b[34;
       'reward': -1,
       'state': 388},
      {'action': 4,
       'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | :\x1b[43m \x1b[0m|\n|\x1b[34;
       'reward': -10,
       'state': 388},
     {'action': 0,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[34;1mY\x1b[0m| : |
      'reward': -1,
      'state': 488},
     {'action': 1,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | :\x1b[43m \x1b[0m|\n|\x1b[34;
      'reward': -1,
      'state': 388},
     {'action': 1,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : :\x1b[43m \x1b[0m|\n| | : | : |\n|\x1b[34;
      'reward': -1,
      'state': 288},
     {'action': 1,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : :\x1b[43m \x1b[0m|\n| : : : : |\n| | : | : |\n|\x1b[34;
      'reward': -1,
      'state': 188},
     {'action': 3,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | :\x1b[43m \x1b[0m: |\n| : : : : |\n| | : | : |\n|\x1b[34;
      'reward': -1,
      'state': 168},
     {'action': 4,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | :\x1b[43m \x1b[0m: |\n| : : : : |\n| | : | : |\n|\x1b[34;
      'reward': -10,
      'state': 168},
     {'action': 5,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | :\x1b[43m \x1b[0m: |\n| : : : : |\n| | : | : |\n|\x1b[34;
      'reward': -10,
      'state': 168},
     {'action': 4,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | :\x1b[43m \x1b[0m: |\n| : : : : |\n| | : | : |\n|\x1b[34;
      'reward': -10,
      'state': 168},
     {'action': 0,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : :\x1b[43m \x1b[0m: |\n| | : | : |\n|\x1b[34;
      'reward': -1,
```

(TO DO) Q3 (b) - 1 mark

b) From the output seen from Q3 (a), how did the Baseline approach do and why do you think that it performed well or poorly? Explain with respect to the average timesteps per episode and the average penalties per episode.

The baseline performs rather poorly, this can be seen by the rewards of each approach, most are negative. This is because the baseline approach has random actions and therefore will have many more negative rewards since the driver will try to drop off the passenger at random locations

(TO DO) Q3 (c) - 1 mark

c) Without moving to a Reinforcment Learning approach how can the Baseline approach be modified to perform slightly better?

the baseline approach can be modified to perform slightly better by only allowing the drop off action when the taxi is at the correct location so there will be less negative rewards.

3.0 - Training an Agent with Q-Learning to play the Taxi Game

Now that we have had an agent use the baseline model to complete the taxi simulation, we will have the agent use Q-Learning to try applying a Reinforcement Learning approach to the problem. To start the process, we will create a matrix of Q values for each action-state possibility (initializing it as zero). The agent will update this matrix when training and will need the matrix reset whenever the agent wants to reset its training.

```
# Initialize the table of Q values for the state-action pairs
q_table = np.zeros([env.observation_space.n, env.action_space.n])
```

With the matrix of Q values initialized, we will now define the training function that adjusts the Q values within *q_table*. The training process consists of running through a number of random simulations and updating the Q values for each state via Q-Learning.

There are a number of hyperparameters used by the training function:

- *alpha*: Learning parameter (you will need to describe it in a later question).
- *gamma*: The long term reward discount parameter.
- *epsilon*: Exploitation/Exploration parameter (you will need to describe it in a later question).
- *num_simulations*: Represents how many random episodes should be generated to have the agents use to update its Q values.

Thus, by running through this algorithm, an agent can learn which Q-values to use when working with other episodes.

```
def train_agent(alpha, gamma, epsilon, num_simulations):
    '''
    Trains an agent by updating its Q values for a total of num_simulations
    episodes with the alpha, gamma, and epsilon hyperparameters.
    '''
    # For plotting metrics
    all_time_steps = []
    all_penalties = []
    # Generate the specified number of episodes
    for i in range(1, num_simulations + 1):
        # Generate a new state by resetting it
        state = env.reset()
        # Variables tracked (time steps, total penalties, the reward value)
        time_steps, penalties, reward, = 0, 0, 0
        done = False
        # Run the simulation
        while not done:
            # Select a random action is the randomly selected number from a
            # uniform distribution is less than epsilon
            if random.uniform(0, 1) < epsilon:
                action = env.action_space.sample() # Explore action space
            # Otherwise use the currently learned Q values
            else:
                action = np.argmax(q_table[state]) # Exploit learned values
            # Retrieve the relevant information after performing the action
            next_state, reward, done, info = env.step(action)
            # Retrieve the old Q value and the maximum Q value from the next state
            old_value = q_table[state, action]
            next_max = np.max(q_table[next_state])
            # Update the current Q value
            new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
            q_table[state, action] = new_value
            # Track anytime an incorrect dropoff or pickup is made
            if reward == -10:
                penalties += 1
            # Proceed to the next state and time step
            state = next_state
            time_steps += 1
        # Display progress for each 100 episodes
        if i % 100 == 0:
            clear_output(wait=True)
            print(f"Episode: {i}")
    print("Training finished.\n")
```

We now use the training function with a set of hyperparameters to train the agent with Q-Learning to potentially improve performance over time.

```
# Hyperparameters
alpha = 0.1
gamma = 0.5
epsilon = 0.1
num_simulations = 100000
# Train the agent
train_agent(alpha, gamma, epsilon, num_simulations)
```

```
Episode: 100000
Training finished.
```

After the training, we can look at the Q-values that have been obtained in our state-action table for a specific state. Below we see that each Q-value for the six possible actions available for state 328 have been updated accordingly.

(TO DO) Q4 - 2 marks

Below we print the Q-values that are available for the six actions at state 328 and we render that state to view it. Based on the available Q-values (assuming we are in exploitation mode), which action would be the next to be selected (or if there are ties, list all possible actions that would be considered)? Do any of the actions that contain larger Q values seem problematic if they were selected? Why or why not?

```
print(q_table[328])
env.s = 328
env.render()
```

```
[ -1.97651797  -1.95703125  -1.98455195  -1.97698499 -10.44100439
 -10.31824964]
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
```

TODO... The second action, North, is going to be selected next since it has the largest reward (or in this case the least negative). This would be benificial given the location of the taxi. The pickup and drop off actions contain higher (more negative) Q values, this is because dropping off/picking up the passenger at the wrong location has a larger penalty. This can be problematic because it can mean that the passenger may not be dropped off/ picked up even when the taxi is at the correct location.

With the training complete, we can now evaluate the Q-Learning approach in a similar method that we used to evaluate the Baseline approach. By passing the number of episodes to test for and the environment, we generate that number of random episodes and average the results obtained from running the Q-Learning approach to complete the episodes. Unlike the training, it is important to note that the hyperparameters that are used there are not used here. The agent simply uses the maximum Q-value at each step to determine which action to take at a given time step.

If the evaluate_agent_QL function ever seems to be running for far too long (a minute or more), stop the run by clicking the button at the top-left of the code cell being executed and run it again. This occurs because the training was insufficient at setting valid Q values and resulted in a dead-end for a specific state.

```
def evaluate_agent_QL(episodes, env):
    '''
    Given a number to specify how many random states to run and the environment to use,
    display the averaged metrics obtained from the tests and return the frames obtained from the tests.
    '''
    total_time_steps, total_penalties = 0, 0
    frames = []
    for _ in range(episodes):
        # Generate a random state to use
        state = env.reset()
        # The information collected throughout the run
        time_steps, penalties, reward = 0, 0, 0
        # Determines when the episode is complete
        done = False
        # Run through the episode until complete
        while not done:
            # Select the action containing the maximum Q value
            action = np.argmax(q_table[state])
            # Run that action and retrieve the reward and other details
            state, reward, done, info = env.step(action)

            # Put each rendered frame into dict for animation
            frames.append({
                'frame': env.render(mode='ansi'),
                'state': state,
                'action': action,
                'reward': reward
                }
            )
```

```
            )
            # Specify whether the agent incorrectly chose to pick up or dropoff a passenger
            if reward == -10:
                penalties += 1
            # Increment the current time step
            time_steps += 1
        # Track the totals
        total_penalties += penalties
        total_time_steps += time_steps
    # Display the performance over the tests
    print(f"Results after {episodes} episodes:")
    print(f"Average timesteps per episode: {total_time_steps / episodes}")
    print(f"Average penalties per episode: {total_penalties / episodes}")
    # Return the frames to allow a user to view the runs
    return frames
```

(TO DO) Q5

a) Run the *evaluate_agent_QL* for 100 episodes to retrieve the average number of time steps and the average penalty after training.

b) Given your results from Q5 (a), how do the observed results from the tests compare to the tests from the Baseline model in Q3 (a)? Specifically, which agent performs better with respect to the average number of penalties throughout the tests and which agent is able to solve the problems quicker (on average).

(TO DO) Q5 (a) - 1 mark

a) Run the *evaluate_agent_QL* for 100 episodes to retrieve the average number of time steps and the average penalty after training.

```
evaluate_agent_QL(100, env)
```

```
Results after 100 episodes:
Average timesteps per episode: 12.93
Average penalties per episode: 0.0
[{'action': 0,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | :\x1b[43m \x1b[0m: |\n| : : : : |\n| | : | : |\n|\x1b[34;
  'reward': -1,
  'state': 168},
 {'action': 0,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : :\x1b[43m \x1b[0m: |\n| | : | : |\n|\x1b[34;
  'reward': -1,
  'state': 268},
 {'action': 3,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : :\x1b[43m \x1b[0m: : |\n| | : | : |\n|\x1b[34;
  'reward': -1,
  'state': 248},
 {'action': 3,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| :\x1b[43m \x1b[0m: : : |\n| | : | : |\n|\x1b[34;
  'reward': -1,
  'state': 228},
 {'action': 3,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n|\x1b[43m \x1b[0m: : : : |\n| | : | : |\n|\x1b[34;
  'reward': -1,
  'state': 208},
 {'action': 0,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n|\x1b[43m \x1b[0m| : | : |\n|\x1b[34;
  'reward': -1,
  'state': 308},
 {'action': 0,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[34;1m\x1b[43mY\x1b
  'reward': -1,
  'state': 408},
 {'action': 4,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[42mY\x1b[0m| : |B:
  'reward': -1,
  'state': 416},
 {'action': 1,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n|\x1b[42m_\x1b[0m| : | : |\n|Y| : |B:
  'reward': -1,
  'state': 316},
 {'action': 1,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n|\x1b[42m_\x1b[0m: : : : |\n| | : | : |\n|Y| : |B:
  'reward': -1,
  'state': 216},
 {'action': 1,
  'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n|\x1b[42m_\x1b[0m: | : : |\n| : : : : |\n| | : | : |\n|Y| : |B:
  'reward': -1,
  'state': 116},
 {'action': 1,
  'frame': '+---------+\n|\x1b[35m\x1b[42mR\x1b[0m\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|Y| : |B:
  'reward': -1,
  'state': 16},
 {'action': 5,
  'frame': '+---------+\n|\x1b[35m\x1b[34;1m\x1b[43mR\x1b[0m\x1b[0m\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | :
  'reward': 20,
  'state': 0},
 {'action': 0,
  'frame': '+---------+\n|R: | : :G|\n| :\x1b[43m \x1b[0m| : : |\n| : : : : |\n| | : | : |\n|\x1b[35mY\x1b[0m| : |\x
```

```
'reward': -1,
```

(TO DO) Q5 (b) - 2 marks

b) Given your results from Q5 (a), how do the observed results from the tests compare to the tests from the Baseline model in Q3 (a)? Specifically, which agent performs better with respect to the average number of penalties throughout the tests and which agent is able to solve the problems quicker (on average).

TODO ...
Therefore the Q-Learning agent performs better than the Baseline model agent. This can be seen in both the Timesteps per episode and penalties per episode. The timesteps per episode for the baseline was 2406.42 while the Q-learning was 12.93. The penalties per episode for baseline was 780.26 while the Q-learning was 0. As it's shown, the Q-learning performs better on every measure and is able to solve the problems quicker on average.

4.0 - Testing Different Hyperparameters

Now we will try retraining the agent using different set ups for the hyperparameters. This will allow you to explore their impact on the Q-Learning as well as understand their purpose during the training.

(TO DO) Q6

Below we explore variations for all four hyperparameters used by the Q-Learning approach to better understand their impact on the training. When answering the questions, ***be careful to correctly set the hyperparameters***.

a) Retrain the agent by resetting the Q learning values and training for only **35000 episodes** (with the same alpha, gamma, and epsilon values used in section 3.0 of this notebook). Then perform another test for 100 episodes with the environment.
b) Retrain the agent by resetting the Q learning values and training for **100000 episodes**, but with an **epsilon value of 0.8** (with the same alpha and gamma values used in section 3.0 of this notebook). Then perform another test for 100 episodes with the environment.
c) Retrain the agent by resetting the Q learning values and training for **100000 episodes**, but with an **alpha value of 0.7** (with the same gamma and epsilon values used in section 3.0 of this notebook). Then perform another test for 100 episodes with the environment.
d) Retrain the agent by resetting the Q learning values and training for **100000 episodes**, but with an **gamma value of 0.15** (with the same alpha and epsilon values used in section 3.0 of this notebook). Then perform another test for 100 episodes with the environment.
e) Based on your knowledge describe what the alpha and epsilon values are within the training function (i.e. what do they affect/do).
f) Using the results obtained from your tests in Q6 (a), (b), (c), and (d), along with the initial results found from Q5, explain the impacts of modifying the number of episodes trained on (less vs more), the alpha value (lower vs higher), the gamma value (lower vs higher), and the epsilon value (lower vs higher). Even if the difference in the comparisons are minor, state them.

As a note, below are the initial hyperparameter values used from section 3.0 of this notebook to use as reference:

alpha = 0.1
gamma = 0.5
epsilon = 0.1
num_simulations = 100000

(TO DO) Q6 (a) - 2 marks

a) Retrain the agent by resetting the Q learning values and training for only **35000 episodes** (with the same alpha, gamma, and epsilon values used in section 3.0 of this notebook). Then perform another test for 100 episodes with the environment.

```
# TODO: Reset q_table
q_table = np.zeros([env.observation_space.n, env.action_space.n])
# TODO: Retrain with the specified hyperparameters
# Hyperparameters
alpha = 0.1
gamma = 0.5
epsilon = 0.1
num_simulations = 35000
train_agent(alpha, gamma, epsilon, num_simulations)
# TODO: Test for 100 episodes
evaluate_agent_QL(100, env)
```

```
    Episode: 35000
    Training finished.

    Results after 100 episodes:
    Average timesteps per episode: 13.06
    Average penalties per episode: 0.0
    [{'action': 3,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : :\x1b[43m \x1b[0m: |\n| | : | : |\n|\x1b[34;
      'reward': -1,
```

```
 'state': 269},
{'action': 3,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : :\x1b[43m \x1b[0m: : |\n| | : | : |\n|\x1b[34;
 'reward': -1,
 'state': 249},
{'action': 3,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| :\x1b[43m \x1b[0m: : : |\n| | : | : |\n|\x1b[34;
 'reward': -1,
 'state': 229},
{'action': 3,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n|\x1b[43m \x1b[0m: : : : |\n| | : | : |\n|\x1b[34;
 'reward': -1,
 'state': 209},
{'action': 0,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n|\x1b[43m \x1b[0m| : | : |\n|\x1b[34;
 'reward': -1,
 'state': 309},
{'action': 0,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[34;1m\x1b[43mY\x1b
 'reward': -1,
 'state': 409},
{'action': 4,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[42mY\x1b[0m| : |B:
 'reward': -1,
 'state': 417},
{'action': 1,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n|\x1b[42m_\x1b[0m| : | : |\n|Y| : |B:
 'reward': -1,
 'state': 317},
{'action': 1,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n|\x1b[42m_\x1b[0m: : : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 217},
{'action': 2,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| :\x1b[42m_\x1b[0m: : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 237},
{'action': 2,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : :\x1b[42m_\x1b[0m: : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 257},
{'action': 2,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : :\x1b[42m_\x1b[0m: |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 277},
{'action': 2,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : :\x1b[42m_\x1b[0m|\n| | : | : |\n|Y| : |B:
 'reward': -1,
```

(TO DO) Q6 (b) - 2 marks

b) Retrain the agent by resetting the Q learning values and training for **100000 episodes**, but with an **epsilon value of 0.8** (with the same alpha and gamma values used in section 3.0 of this notebook). Then perform another test for 100 episodes with the environment.

```
# TODO: Reset q_table
q_table = np.zeros([env.observation_space.n, env.action_space.n])
# TODO: Retrain with the specified hyperparameters
# Hyperparameters
alpha = 0.1
gamma = 0.5
epsilon = 0.8
num_simulations = 100000
train_agent(alpha, gamma, epsilon, num_simulations)
# TODO: Test for 100 episodes
evaluate_agent_QL(100, env)
```

```
Episode: 100000
Training finished.

Results after 100 episodes:
Average timesteps per episode: 12.88
Average penalties per episode: 0.0
[{'action': 1,
  'frame': '+---------+\n|\x1b[34;1mR\x1b[0m:\x1b[43m \x1b[0m| : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[3
  'reward': -1,
  'state': 22},
 {'action': 3,
  'frame': '+---------+\n|\x1b[34;1m\x1b[43mR\x1b[0m\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[3
  'reward': -1,
  'state': 2},
 {'action': 4,
  'frame': '+---------+\n|\x1b[42mR\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[35mY\x1b[0m| : |B:
  'reward': -1,
  'state': 18},
 {'action': 0,
```

```
 'frame': '+---------+\n|R: | : :G|\n|\x1b[42m_\x1b[0m: | : : |\n| : : : : |\n| | : | : |\n|\x1b[35mY\x1b[0m| : |B:
 'reward': -1,
 'state': 118},
{'action': 0,
 'frame': '+---------+\n|R: | : :G|\n| : | : : |\n|\x1b[42m_\x1b[0m: : : : |\n| | : | : |\n|\x1b[35mY\x1b[0m| : |B:
 'reward': -1,
 'state': 218},
{'action': 0,
 'frame': '+---------+\n|R: | : :G|\n| : | : : |\n| : : : : |\n|\x1b[42m_\x1b[0m| : | : |\n|\x1b[35mY\x1b[0m| : |B:
 'reward': -1,
 'state': 318},
{'action': 0,
 'frame': '+---------+\n|R: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[35m\x1b[42mY\x1b[0m\x1b[0m| : |B:
 'reward': -1,
 'state': 418},
{'action': 5,
 'frame': '+---------+\n|R: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[35m\x1b[34;1m\x1b[43mY\x1b[0m\x1b
 'reward': 20,
 'state': 410},
{'action': 3,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | : |\n|Y| : |\x1b[34;1m\x1b[43
 'reward': -1,
 'state': 473},
{'action': 4,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | : |\n|Y| : |\x1b[42mB\x1b[0m:
 'reward': -1,
 'state': 477},
{'action': 1,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : |\x1b[42m_\x1b[0m: |\n|Y| : |B:
 'reward': -1,
 'state': 377},
{'action': 1,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : :\x1b[42m_\x1b[0m: |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 277},
{'action': 1,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | :\x1b[42m_\x1b[0m: |\n| : : : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 177}
```

(TO DO) Q6 (c) - 2 marks

c) Retrain the agent by resetting the Q learning values and training for **100000 episodes**, but with an **alpha value of 0.7** (with the same gamma and epsilon values used in section 3.0 of this notebook). Then perform another test for 100 episodes with the environment.

```
# TODO: Reset q_table
q_table = np.zeros([env.observation_space.n, env.action_space.n])
# TODO: Retrain with the specified hyperparameters
# Hyperparameters
alpha = 0.7
gamma = 0.5
epsilon = 0.1
num_simulations = 100000
train_agent(alpha, gamma, epsilon, num_simulations)
# TODO: Test for 100 episodes
evaluate_agent_QL(100, env)
```

```
    Episode: 100000
    Training finished.

    Results after 100 episodes:
    Average timesteps per episode: 13.12
    Average penalties per episode: 0.0
    [{'action': 1,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :\x1b[34;1mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | :\x1b[43m \x
      'reward': -1,
      'state': 384},
     {'action': 1,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :\x1b[34;1mG\x1b[0m|\n| : | : : |\n| : : : :\x1b[43m \x1b[0m|\n| | :
      'reward': -1,
      'state': 284},
     {'action': 1,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :\x1b[34;1mG\x1b[0m|\n| : | : :\x1b[43m \x1b[0m|\n| : : : : |\n| | :
      'reward': -1,
      'state': 184},
     {'action': 1,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :\x1b[34;1m\x1b[43mG\x1b[0m\x1b[0m|\n| : | : : |\n| : : : : |\n| | :
      'reward': -1,
      'state': 84},
     {'action': 4,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :\x1b[42mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | : |\n|Y| : |B:
      'reward': -1,
      'state': 96},
     {'action': 0,
      'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : :\x1b[42m_\x1b[0m|\n| : : : : |\n| | : | : |\n|Y| : |B:
      'reward': -1,
      'state': 196},
```

```
{'action': 0,
 'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : : :\x1b[42m_\x1b[0m|\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 296},
{'action': 3,
 'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : : :\x1b[42m_\x1b[0m: |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 276},
{'action': 3,
 'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| : :\x1b[42m_\x1b[0m: : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 256},
{'action': 3,
 'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| : | : : |\n| :\x1b[42m_\x1b[0m: : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 236},
{'action': 1,
 'frame': '+---------+\n|\x1b[35mR\x1b[0m: | : :G|\n| :\x1b[42m_\x1b[0m| : : |\n| : : : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 136},
{'action': 1,
 'frame': '+---------+\n|\x1b[35mR\x1b[0m:\x1b[42m_\x1b[0m| : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 36},
{'action': 3,
 'frame': '+---------+\n|\x1b[35m\x1b[42mR\x1b[0m\x1b[0m: | : :G|\n| : | : : |\n| : : : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
```

(TO DO) Q6 (d) - 2 marks

d) Retrain the agent by resetting the Q learning values and training for **100000 episodes**, but with an **gamma value of 0.15** (with the same alpha and epsilon values used in section 3.0 of this notebook). Then perform another test for 100 episodes with the environment.

```
# TODO: Reset q_table
q_table = np.zeros([env.observation_space.n, env.action_space.n])
# TODO: Retrain with the specified hyperparameters
# Hyperparameters
alpha = 0.1
gamma = 0.15
epsilon = 0.1
num_simulations = 100000
train_agent(alpha, gamma, epsilon, num_simulations)
# TODO: Test for 100 episodes
evaluate_agent_QL(100, env)
```

```
    Episode: 100000
    Training finished.

    Results after 100 episodes:
    Average timesteps per episode: 12.58
    Average penalties per episode: 0.0
    [{'action': 3,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n|\x1b[43m \x1b[0m: : : |\n| | : | : |\n|\x1b[34;
      'reward': -1,
      'state': 209},
     {'action': 0,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n|\x1b[43m \x1b[0m: | : |\n|\x1b[34;
      'reward': -1,
      'state': 309},
     {'action': 0,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[34;1m\x1b[43mY\x1b
      'reward': -1,
      'state': 409},
     {'action': 4,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | : |\n|\x1b[42mY\x1b[0m| : |B:
      'reward': -1,
      'state': 417},
     {'action': 1,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : : : : |\n|\x1b[42m_\x1b[0m| : | : |\n|Y| : |B:
      'reward': -1,
      'state': 317},
     {'action': 1,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n|\x1b[42m_\x1b[0m: : : |\n| | : | : |\n|Y| : |B:
      'reward': -1,
      'state': 217},
     {'action': 2,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| :\x1b[42m_\x1b[0m: : |\n| | : | : |\n|Y| : |B:
      'reward': -1,
      'state': 237},
     {'action': 2,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : : |\n| : :\x1b[42m_\x1b[0m: : |\n| | : | : |\n|Y| : |B:
      'reward': -1,
      'state': 257},
     {'action': 1,
      'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : |\x1b[42m_\x1b[0m: : |\n| : : : : |\n| | : | : |\n|Y| : |B:
```

```
 'reward': -1,
 'state': 157},
{'action': 2,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | :\x1b[42m_\x1b[0m: |\n| : : : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 177},
{'action': 2,
 'frame': '+---------+\n|R: | : :\x1b[35mG\x1b[0m|\n| : | : :\x1b[42m_\x1b[0m|\n| : : : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 197},
{'action': 1,
 'frame': '+---------+\n|R: | : :\x1b[35m\x1b[42mG\x1b[0m\x1b[0m|\n| : | : : |\n| : : : : |\n| | : | : |\n|Y| : |B:
 'reward': -1,
 'state': 97},
{'action': 5,
 'frame': '+---------+\n|R: | : :\x1b[35m\x1b[34;1m\x1b[43mG\x1b[0m\x1b[0m\x1b[0m|\n| : | : : |\n| : : : : |\n| | :
 'reward': 20,
 'state': 85}
```

(TO DO) Q6 (e) - 2 marks

e) Based on your knowledge describe what the alpha and epsilon values are within the training function (i.e. what do they affect/do).

TODO ...

The alpha is the learning rate and epsilon is the rate at which the agent will perform a random action. The alpha determines how much of the original value of q we take into account when updating the q table. If alpha is too high then we are likely to miss and overshoot the ideal policy and if alpha is too low then the training takes longer (require more simulations). Epsilon determines how often the agent takes a random action, if it is too high then we can not determine a ideal policy properly since it is taking random actions more often than not and will take a long time to train with more simulations. If epsilon is too low then there will not be enough random actions and the policy will not improve.

(TO DO) Q6 (f) - 4 marks

f) Using the results obtained from your tests in Q6 (a), (b), (c), and (d), along with the initial results found from Q5 to serve as the Q-Learning baseline to compare with, explain the impacts of modifying the number of episodes trained on (less vs more), the alpha value (lower vs higher), the gamma value (lower vs higher), and the epsilon value (lower vs higher). Even if the difference in the comparisons are minor, state them.

```
TODO ...
Overall chaning the parameters did not have too much of a dras
it only changed the averagae time step per episode by a maximu
certain is that the lower training simulations led to worse pe
. This makes sense though since generally more training leads
result, especially given the modest values of alpha, gamma and
that run.

Changing alpha:
Increasing the alpha on 6 c) from 0.1 to 0.7 caused the agent
as it increased the average timesteps from 12.93 in Q5 to 13.2
because increasing the alpha led to overshooting the ideal pol

Changing gamma:
Decreasing gamma in 6 d) from 0.5 to 0.15 caused the agent to
better than in Q5, it decreased the average timesteps from 12.
seems focusing less on the longterm goal caused the agent to i
strange since dropping off the passenger is the highest reward

Changing epsilon:
Increasing epsilon from 0.1 to 0.8 in 6 b) led the agent to pe
better. The average timesteps went rfom 12.93 in Q5 to 12.88.
randomness seemed to have helped train the agent slightly bett
```

TODO ... Overall chaning the parameters did not have too much of a drastic difference as it only changed the averagae time step per episode by a maximum of 1. What is certain is that the lower training simulations led to worse performance in 6 a). This makes sense though since generally more training leads to an improved result, especially given the modest values of alpha, gamma and epsilon used in that run.

Changing alpha: Increasing the alpha on 6 c) from 0.1 to 0.7 caused the agent to perform worse as it increased the average timesteps from 12.93 in Q5 to 13.2. This may be because increasing the alpha led to overshooting the ideal policy.

Changing gamma: Decreasing gamma in 6 d) from 0.5 to 0.15 caused the agent to perform slightly better than in Q5, it decreased the average timesteps from 12.93 to 12.58. It seems focusing less on the longterm goal caused the agent to improve, which is strange since dropping off the passenger is the highest reward.

Changing epsilon: Increasing epsilon from 0.1 to 0.8 in 6 b) led the agent to perform slightly better. The average timesteps went rfom 12.93 in Q5 to 12.88. The increased randomness seemed to have helped train the agent slightly better.

SIGNATURE: My name is Shail Patel. My student number is 8234706. I certify being the author of this assignment.