UNIVERSITY OF MARYLAND

Robot Learning Final Project

Self Driving Cab using Reinforcement Learning

By Akshitha Pothamshetty
UID: 116399326

Table of Content

Abstract

In the final project, I have trained a self driving cab using Reinforcement learning to pick up and drop off passengers safely in a small city. I have used the Q-learning technique to accomplish this task. I have used OpenAI gym's virtual environment to emulate this small city. Being a small city, the number of states for our environment is less and hence, Q-learning can be applied without any hesitation. There are 4 locations (labeled by different letters), and our job is to pick up the passenger at one location and drop him off at another. We receive +20 points for a successful drop-off and lose 1 point for every time-step it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions. In this report, I go in detail about how I was able to successfully complete the learning. As we will see, the taxi agent perfectly learns how to pick up and drop off customers in this small 5x5 grid city.

Introduction

Around the world, companies are trying to push autonomous cars from level 3 to level 4 and 5. In 2025, the market for autonomous driving and assistive safety and comfort features is projected to be sized at 26 billion U.S. dollars in an optimistic scenario. Many companies are pushing their efforts at developing self-driving cabs as well.

Autonomous vehicles can get into many different situations on the road. If drivers are going to entrust their lives to self-driving cars, they need to be sure that these cars will be ready for the craziest of situations. A car has to learn and adapt to the ever-changing behavior of other vehicles around it. Machine learning algorithms make autonomous vehicles capable of making decisions in real time. This increases safety and trust in autonomous cars.

In this project, I implement a Reinforcement Learning strategy to tackle the challenges of a similar but simplified problem. Using Q-learning, I train a self-driving cab agent to navigate through a 5x5 grid city. The solution for this project, with a modification of Deep Neural Networks instead of Q-tables can be used for bigger grids as well.

Background

For the project, I had to explore the concept of Reinforcement Learning. In this section, I go in-depth about my research and understanding of RL and how it suits my case.

What is Reinforcement Learning?

Wikipedia: "Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment in order to

maximize the notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning."

Basically, RL is concerned with producing algorithms(agents) that try to achieve some predefined goals. We reward the agent when it does the right thing and put a penalty when it goes wrong. The essence of RL lies in maximizing cumulative rewards rather than immediate rewards. The achievement of this objective is dependent on choosing a set of actions - receiving rewards for good ones and punishments for the bad ones. The agent acts in an environment that has a state and gives rewards and a set of actions (figure 1).

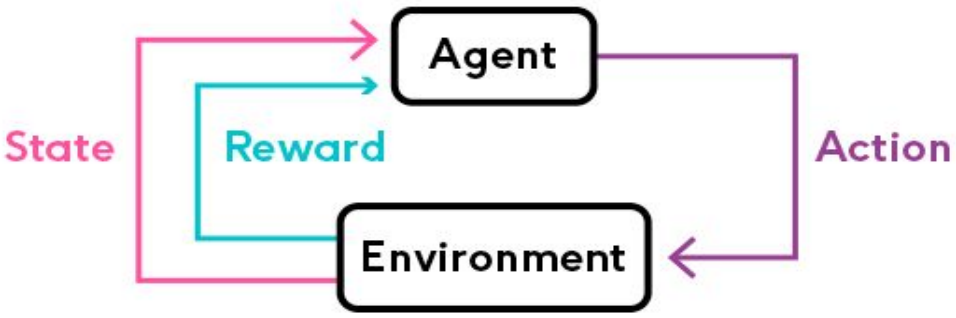


Figure 1. Reinforcement Learning Process

How to represent the RL problem mathematically?

To represent the Reinforcement Learning problem mathematically, we use Markov formulation (Markov Decision Process). Markov property states that the future of a process depends completely on its present state. The current state completely represents the state of the environment.

Wikipedia: "A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker."

A Markov decision process is a 4-tuple *(S, A, P_a , R_a)* where:

- *S* is a finite set of states,
- *A* is a finite set of actions (alternatively, A_s is the finite set of actions available from state *s*),
- $P_a(s, s') = P_r (s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action *a* in state *s* at time *t* will lead to state *s'* at time *t+1*.
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state *s* to *s'*, due to action *a*.

For our case, we introduce one more variable into the MDP process γ , the reward discount factor. The reward discount factor allows us to compare the worth of future rewards vs the immediate rewards. A discount factor of 1 would make future rewards worth just as much as immediate rewards.

How learning actually takes place?

What we actually mean by reinforcement learning, is to train our agent to take right decisions from any given state. To accomplish this, we put the agent in the environment over and over again to learn the right decisions. Reward and penalties are the crucial part of this learning process. For taking a right decision, we present a reward say $100 and for wrong decisions, we impose penalties, say -$50. The goal of the process is to maximize the cumulative reward.

Here is how learning happens in RL context:

```
While t ←0 until done:
   1. Environment gives the agent a state.
   2. Agent chooses an action from sample actions.
   3. Environment gives a reward along with a new state.
   4. Continue until the goal or other terminating condition is met.
```

The objective of the above training is to find an optimal policy, π^* that maximizes the cumulative discounted reward:

$$\sum_{t \geq 0} \gamma^t r_t$$

where r_t is the reward received at step *t* and γ^t is the discount factor at step t_t . A policy π is a function that maps state *s* to action *a*, that our agent believes is the best, given that state.

Value Function

To get the maximum expected future reward the agent will get, starting from some state *s* and following some policy, π , we use the following Value function:

$$V_\pi(s) = \mathop{\mathbb{E}}_{\pi}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s]$$

There exists an optimal value function that has the highest value for all states. Defined as:

$$V^*(s) = \max_\pi V^\pi(s) \quad \forall s \in \mathbb{S}$$

Q - Function

Q-function gives the expected return starting from state *s*, taking action *a* and following policy π :

$$Q_\pi(s,a) = \mathop{\mathbb{E}}_{\pi}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

Just like V*, Q* is optimal Q-function defined as:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \quad \forall s \in \mathbb{S}$$

Interestingly, the maximum expected total reward when starting at state *s* is the maximum of optimal Q-function over all possible actions:

$$V^*(s) = \max_a Q^*(s,a) \quad \forall s \in \mathbb{S}$$

The optimal policy π^* can be found by choosing the action *a* that gives the maximum reward Q*(s,a)* for state s:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \quad \forall s \in \mathbb{S}$$

Using these formulations we can build an optimal agent for a given environment.

How Q-Learning simplifies our learning task?

Wikipedia: "Q-learning is a model-free reinforcement learning algorithm to learn a policy telling an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations."

For Q-learning all we need is all possible states of the environment. Because it finds the q-value for each state-action pair, it can become slow in a short time if the environment is large. In our example, the city is a 5x5 grid having 500 possible states, which the Q-learning model can handle with no problem at all!

Bellman Equation

Given a state *s* and action *a*, we can express *Q(s,a)* in terms of itself:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

where *r* is the reward agent received for being in state *s* and γ is the reward factor.

Above equation evaluates the maximum possible future reward as the current reward, *r* plus the maximum future reward for state *s'* for every possible action.

We can define iterative approximation of the Bellman equation as follows:

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha(r_{t+1} + \gamma \max_a Q_t(s_{t+1},a) - Q_t(s_t,a_t))$$

where α is the learning rate that controls how much the difference between previous and new *Q* value is considered.

The above expression is all we need to start our training through Q-learning. In summary, through training over a long period, what we want to achieve is a Q-table that maps the best possible actions from each state through *q-values*. Later, when training is completed, we can use this table instead of calculating anything to complete the task. Practical implementation of Q-learning is covered in the *Approach* section.

Approach

In this section, I explain how I have used the Q-learning algorithm to train the agent.

Environment

To keep the focus on learning rather than developing a car model myself, I have used OpenAI GYM's Taxi environment. There are 4 locations (labeled by different letters) and your job is to pick up the passenger at one location and drop him off in another. You receive +20 points for a successful dropoff, and lose 1 point for every timestep it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions.

What makes the choice of this environment more tempting is it can simply be printed to the console and doesn't require any graphics. Here is a sample environment initialized to a random state:

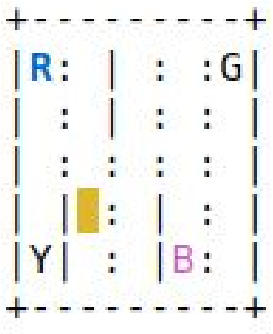


Figure 2. Random State

There are four pick up and drop off destinations: *R, G, Y and B*. Currently, the passenger is at R position waiting to be picked up. Location where the passenger waits for pick up is highlighted in *Blue*. The Passenger in this scenario wants to reach *B*, highlighted in *Pink*. Cab is currently at *(row: 3, column: 1)* in the environment.

State Space

The state space is the set of all possible situations our taxi could inhabit. Our environment consists of 500 possible states:

- Since it is a 5x5 grid, with 4 pick up or drop off location, with one additional state of passenger being inside the taxi:

5x5 x (4+1) x 4 = 500 states

Each state has a unique ID associated with it. For example, the state shown in *figure 2* has ID: 323. In every state, there will be a certain number of actions available.

Action Space

At any state, there are six possible states:

1. *Go South*
2. *Go North*
3. *Go East*
4. *Go West*
5. *Pick Up Passenger*

6. *Drop Off a Passenger*

The taxi cannot perform certain actions in certain states due to walls. We will put a penalty of -1 when it hits a wall. Later after training, the agent will learn to avoid the walls.

For each state, there is a reward table associated which lists the reward info corresponding to each action. This information is listed in the following way:

Action: [Probability, NextState, Reward, GoalReached]

The 0-5 corresponds to the actions (south, north, east, west, pickup, dropoff) the taxi can perform at our current state in the illustration. In this env, *probability* is always 1.0. The *nextstate* is the state we would be in if we take the action at this index of the dict. *Reward* informs the points that would be received on taking that action. *GoalReached* is used to tell us when we have successfully dropped off a passenger in the right location.

For the state shown in *figure 2*(323), following is the action-reward pair:

{0: [(1.0, 423, -1, False)],
1: [(1.0, 223, -1, False)],
2: [(1.0, 343, -1, False)],
3: [(1.0, 323, -1, False)],
4: [(1.0, 323, -10, False)],
5: [(1.0, 323, -10, False)]}

Q-Learning Approach

As described in the Background section, we will use the iterative Bellman equation to update our Q-values in the Q-table. Here is the general algorithm I implemented in the jupyter notebook while training the model:

```
1. Initialize a Q-values table
2. Observe initial state s
3. Choose action a and act
4. Observe reward r and a new state s′
5. Update the Q table using r and the maximum possible
   reward from s′  (iterative Bellman equation)
```

```
6. Set the current state to the new state and repeat from
   step 2 until a terminal state
```

Exploration vs exploitation

In Reinforcement Learning, our strategy has to choose how often it will try something new and how often it will use something it already knows. This is known as exploration/exploitation tradeoff. Exploration is finding new information about the environment. Whereas exploitation is using existing information to maximize the reward.

Initially the agent has no information about the environment. During this time, we want the agent to explore a lot. But after obtaining experience of the environment, we want it to choose actions based on its memory. Eventually all choices will be based on what is learned. This is controlled by exploration rate, ε. As we train more and more, we want ε to decrease over each successful episode.

Implementation

We start with an empty Q-table of size *500x6*. We create a training algorithm to update this Q-table over the training period.

To switch between exploration/exploitation we compare ε with a randomly generated float between 0-1. Here is how I switch between exploitation and exploration. Over time *min_epsilon* decreases, hence more and more exploitation happens.

```
# Explore Action Space
if random.uniform(0, 1) < min_epsilon:
    action = env.action_space.sample()
# Exploit learned Values
else:
```

```
    action = np.argmax(q_table[state])
```

Bellman equation is used in the following way to update the q-values:

```
# Learning: update current q-value based on best chosen next step
new_q_value = (1 - alpha) * old_value + alpha * (reward + gamma *
next_max)
q_table[state, action] = new_q_value
```

After training over tens of thousands of periods, we finish training successfully. Now we analyze what the model has learned through a few example states and what the model predicts in those scenarios, refer to *Figure 3*.

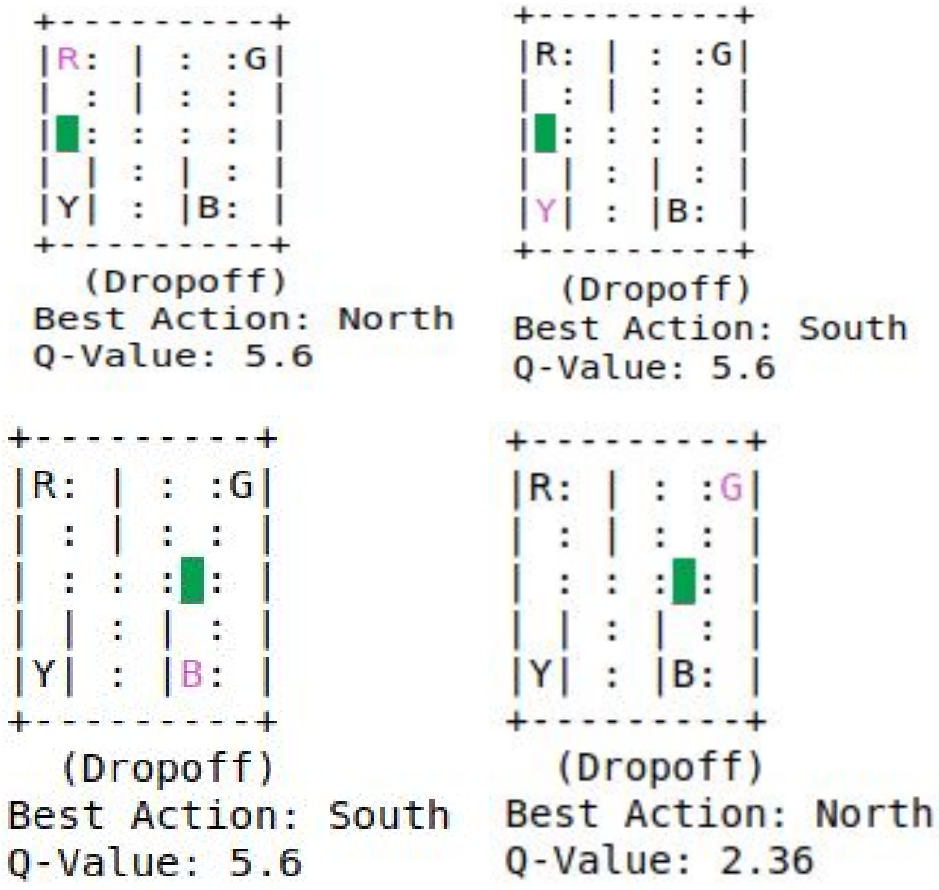


Figure 3. Example States and Predicted actions.

As we can see after the training has been completed, the cab agent is able to predict the best step to reach the destination accurately.

1. In example 1, the cab with the passenger is at (2,0). Dropping destination for the case is R(0). The most sensible step is to move north. The model predicts this accurately as: **North; 5.6**

2. In example 3, the cab with the passenger is at (2,3). Dropping destination for the case is B(3). The most sensible step is to move South. The model predicts this accurately as: **South; 5.6**

3. Similarly, in example 4, can is at (2,3) with the passenger inside. The dropping destination for the case is G(1). Both North and East have the same q-values in this case, 2.36. Thus, the model hasn't overfit and predicts both the correct paths successfully.

Results and Analysis

To evaluate our model, we test the model on new random episodes and analyze its performance. We get the following result on testing over 100 episodes:

Average time steps per episode: 13

Average penalties per episode: 0

It is worth noting that our agent incurred no penalties at all while completing each episode. It also minimized the time steps taken to complete each episode. Hence, our model has been trained perfectly! We perform analysis during the training to tune the hyperparameters and see if the model trains correctly. During training, we want our model to learn to make best decisions to increase the cumulative reward over each episode. As we can see in *figure 3*, our rewards start to increase after 10,000 episodes. It also makes sense to stop training after 25,000 episodes instead of currently 100,000 episodes.

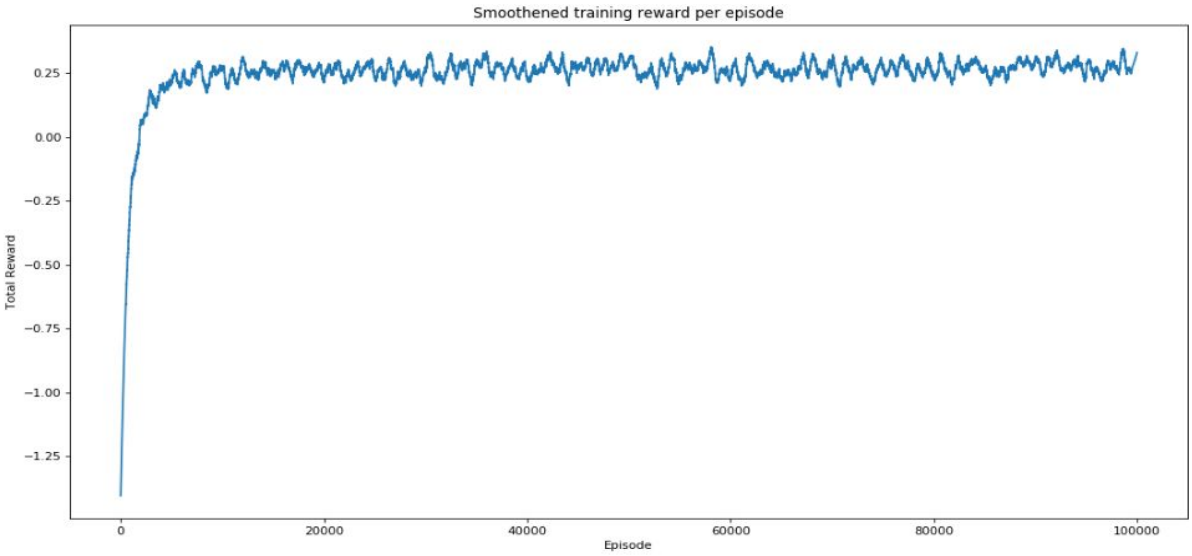


Figure43. Smoothened training reward per episode.

Secondly, we want to see how our model transitions from exploration mode to exploitation mode. It's desired that as the model learns the environment completely, it's *epsilon* value decreases over time i.e. exploration factor reduces over time. As we can see in figure *4*, the exploration factor reduces over the training period. Also, as the model learns the environment completely around 20,000 episodes, the exploration factor reduces exponentially.

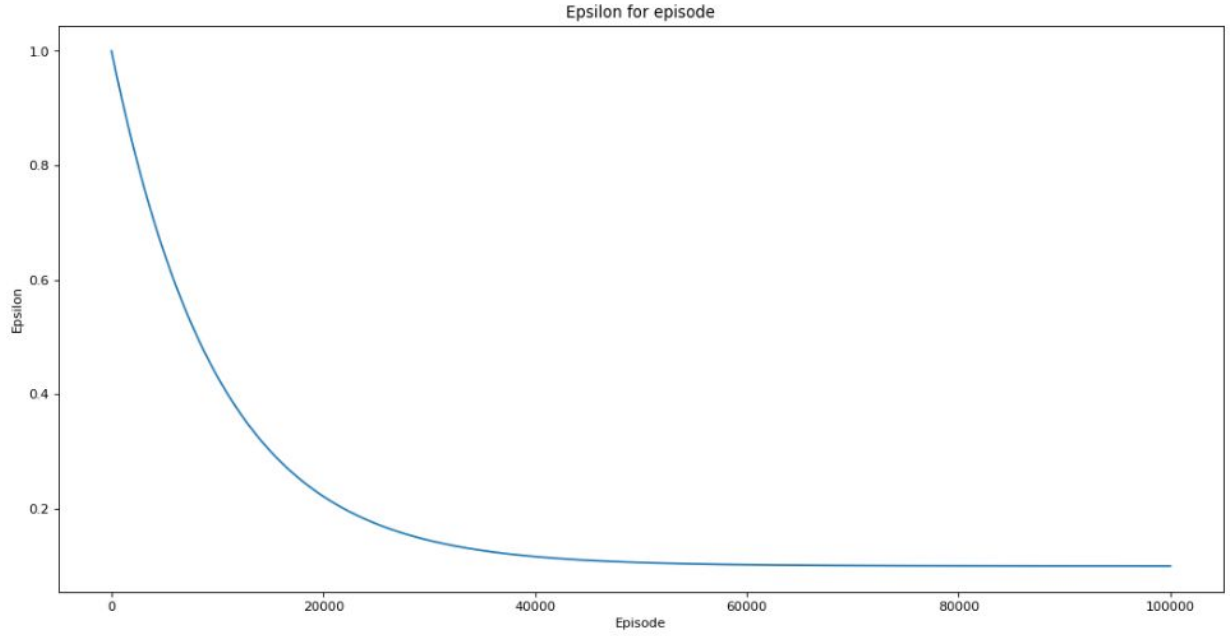


Figure 5. Epsilon factor ε over episodes.

These analyses show that the training was successful. To prove the point, we tested the model in new scenarios and it incurred no penalties and also minimized its route's length.

We evaluate our agent according to the following metrics,

- **Average number of penalties per episode:** The smaller the number, the better the performance of our agent. Ideally, we would like this metric to be zero or very close to zero.
- **Average number of timesteps per trip:** We want a small number of timesteps per episode as well since we want our agent to take minimum steps(i.e. the shortest path) to reach the destination.
- **Average rewards per move:** The larger the reward means the agent is doing the right thing. That's why deciding rewards is a crucial part of Reinforcement Learning. In our case, as both timesteps and penalties are negatively rewarded, a higher average reward would mean that the agent reaches the destination as fast as possible with the least penalties

For comparison, I ran a Brute-force model which would make random moves to complete episodes. Thus, it contains no memory at all. Referring to Table 1, where we compare performances over 100 episodes, we can clearly see our Q-learning model is way superior than a brute force model.

Measure	Random agent's performance	Q-Learning agent's performance
Average rewards per move	-3.9	0.25
Average number of penalties per episode	920.45	0.0
Average number of timesteps per trip	2848.14	12.38

Table 1. Comparison between performance of Random agent vs Q-learning agent

Conclusions and Future Work

In this project, I learned the concept of Q-learning and applied it to make a taxi autonomously pick up and drop off passengers. I used OpenAI's Gym in python to develop my agent and evaluate it. Initially, I gave the agent no intelligence at all and it chose actions randomly. This resulted in poor results. Next, I infused the Q-learning algorithm from scratch into my model. This resulted in perfect results! Our agent successfully completed new episodes after the training.

The Q-learning algorithm worked perfectly fine for our small city. But once the number of states in the environment are very high, it becomes difficult to implement them with a Q table as the size would be very large. Then, we can use Deep Neural networks instead of the Q-table. The neural network takes in state information and actions to the input layer and learns to output the right action over the time. In Future, I would like to implement DNN for the same task and expand the current project to large grids, cities.

Bibliography

1. Wikipedia: Reinforcement Learning
2. Wikipedia: Q-learning
3. Wikipedia: Markov Decision Process
4. OpenAI Gym Documentation
5. OpenAI Taxi-v3 source code

Training an Autonomous Cab to pick up and drop passengers using Q-Learning

Author: **Akshitha Pothamshetty** UID: **116399326** About: **M.Eng Robotics 2020, University of Maryland, College Park**

About Project

In this project, I have used Reinforcement Learning to pick up and drop off passengers at the right locations. I have used Q-Learning to train the model. To keep the focus on applying Q-Learning, I have used existing OpenAI GYM environment, Taxi-v3. This task was introduced in [Dietterich2000] to illustrate some issues in hierarchical reinforcement learning. There are 4 locations (labeled by different letters) and our job is to pick up the passenger at one location and drop him off in another. We receive +20 points for a successful dropoff, and lose 1 point for every timestep it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions. In this notebook, I will go through the project step by step.

1: Install and import dependencies

Following packages needs to be installed successfully for executing rest of the notebook: 1. CMAKE 2. Scipy 3. Numpy 4. Atari Gym

In [3]:
```
!pip install cmake matplotlib scipy numpy 'gym[atari]' # Run only once.
```

In [6]:
```
import gym # Import OpenAI GYM package
from IPython.display import clear_output, Markdown, display # For vis
ualization
from time import sleep # For visualization
import random # Randomly generating states
import numpy as np # For Q-Table
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter

random.seed(42) # Setting random state for consistent results.


env = gym.make("Taxi-v3").env # Using existing Taxi-V3 environment

# Generate a random environment
env.reset()
env.render()

# Properties of our environment
print("Property: Action Space {}".format(env.action_space))
print("Property: State Space {}".format(env.observation_space))
```

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

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

2: Explore our environment

Our environment consists of 6 actions: 1. Go South 2. Go North 3. Go East 4. Go West 5. Pick Up Passenger 6. Drop Off a Passenger Our environment consists of 500 possible states: * It is a 5x5 grid, with 4 pick up or drop off location, with one additional state of passenger being inside the taxi: * 5x5 x (4+1) x 4 = 500 states

In [7]:
```
# Drop off and pick up locations are indexed from 0-3 for the four
# different  locations.

# Let's initiate a forced state:
# Taxi at: (3,1), Pick Up at R(0)(Blue) and Drop Off at B(3)(Pink)

initialState = env.encode(3, 1, 0, 3)
print "Out of 500 possible states, our unique state ID is:", initialS
tate

env.s = initialState # set the environment
env.render()

# Let's see the possible action space in this state
print "Action: [Probability, NextState, Reward, GoalReached]"
env.P[initialState]
```

```
Out of 500 possible states, our unique state ID is: 323
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+

Action: [Probability, NextState, Reward, GoalReached]
```

Out[7]:
```
{0: [(1.0, 423, -1, False)],
 1: [(1.0, 223, -1, False)],
 2: [(1.0, 343, -1, False)],
 3: [(1.0, 323, -1, False)],
 4: [(1.0, 323, -10, False)],
 5: [(1.0, 323, -10, False)]}
```

3: Establish a baseline, using Brute Force Approach

We will use a Brute force approach to establish a baseline. We will ask the taxi to reach the goal state with no intelligence at all. It will choose a random state to reach the goal. We will evaluate how the model performs from here.

In [8]:
```
# Create a visualizer function
def print_frames(frames, episode=1, verbose=False):
    for i, frame in enumerate(frames):
        clear_output(wait=True)
        print(frame['frame'])
        if (verbose):
            print("\nEpisode Number {}".format(episode))
        print("Timestep: {}".format(i + 1))
        print("State: {}".format(frame['state']))
        print("Action: {}".format(frame['action']))
        print("Reward: {}".format(frame['reward']))
        sleep(.1)

def printmd(string):
    display(Markdown(string))
```

In [12]:
```
env.s = initialState  # Set environment to initial state.

epochs = 0
penalties, reward = 0, 0

frames = [] # for animation

done = False

while not done:
    action = env.action_space.sample()
    state, reward, done, info = env.step(action)

    if reward == -10:
        penalties += 1

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

    epochs += 1

# Start visualization only if steps taken are less than 1000. Otherwi
se it takes too long.
if epochs < 1000:
    print_frames(frames)

print("\nSteps taken: {}".format(epochs))
print("Penalties incurred: {}".format(penalties))
```

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

Timestep: 508
State: 475
Action: 5
Reward: 20

Steps taken: 508
Penalties incurred: 163
```

4: Train the model using Reinforcement Learning

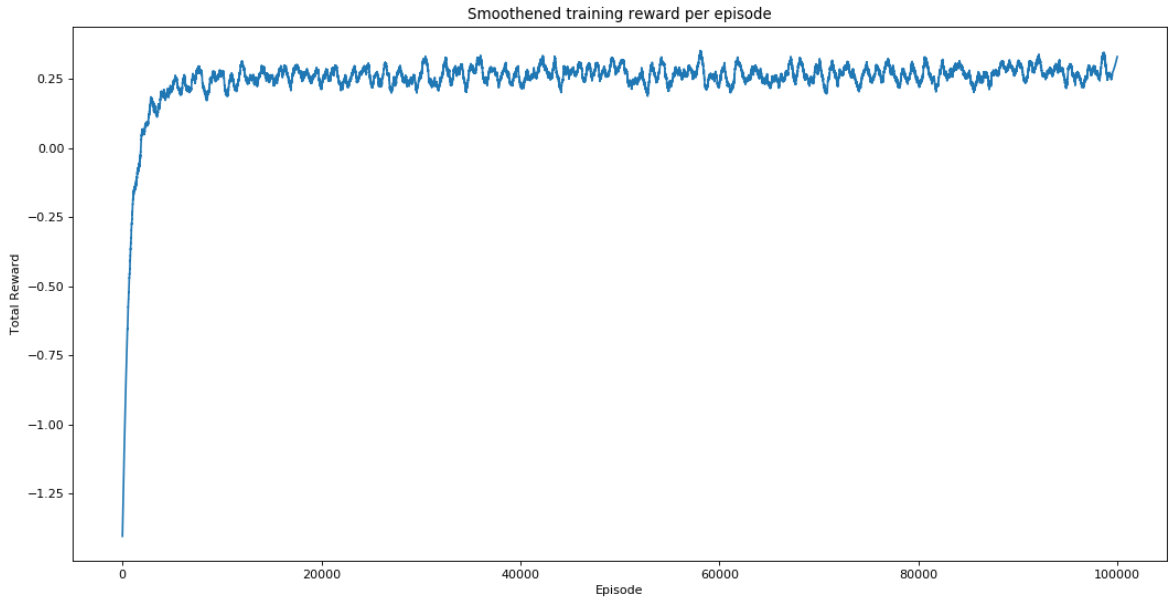
Using a Q-Learning approach to train the model. Q-learning lets the agent use the environment's rewards to learn, over time, the best action to take in a given state. For this purpose, we will create a Q-table to store Q-values, that map to a state and corresponding action combination. A Q-value for a particular state-action combination is representative of the "quality" of an action taken from that state. Better Q-values imply better chances of getting greater rewards. For evaluation of the model, We will keep a track of how many timesteps and number of penalties incurred while training the model.

In [14]:
```
%%time

# initialize a q-table to store q-values for each state-action pair.
q_table = np.zeros([env.observation_space.n, env.action_space.n]) # I
nitialize a q-table with zeros

print"Training the agent .. .. .." # Typically takes around 3m 17seco
nds around

# Hyperparameters
alpha = 0.1
gamma = 0.6
epsilon = 1.0
min_epsilon = 0.1
max_epsilon = 1.00

# For plotting metrics
all_epochs = []
all_penalties = []
all_epsilons = []

frames = [] # for animation
rewards = [] # for analyzing training.

for episode in range(1, 100001):
    state = env.reset()
    episode_rewards = []

    epochs, penalties, reward = 0, 0, 0
    done = False

    while not done:
        # Explore Action Space
        if random.uniform(0, 1) < min_epsilon:
            action = env.action_space.sample()
        # Exploit learned Values
        else:
            action = np.argmax(q_table[state])

        # Get next state, reward, goal_status
        next_state, reward, done, info = env.step(action)

        # current q-value
        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])

        # Learning: update current q-value based on best chosen next
 step
        new_value = (1 - alpha) * old_value + alpha * (reward + gamma
* next_max)
        q_table[state, action] = new_value

        if done:
            epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.ex
p(-0.0001*episode)
            all_epsilons.append(epsilon)
```

```
        # check if penalties incurred
        if reward == -10:
            penalties += 1

        episode_rewards.append(reward)

        # update state
        state = next_state
        epochs += 1

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

    rewards.append(np.mean(episode_rewards))
    # render training progress
    if episode % 100 == 0:
        clear_output(wait=True)
        print("Training ongoing...\nSuccessfully completed {} Pick-Dr
op episodes.".format(episode))

print("Training finished.\n")
```

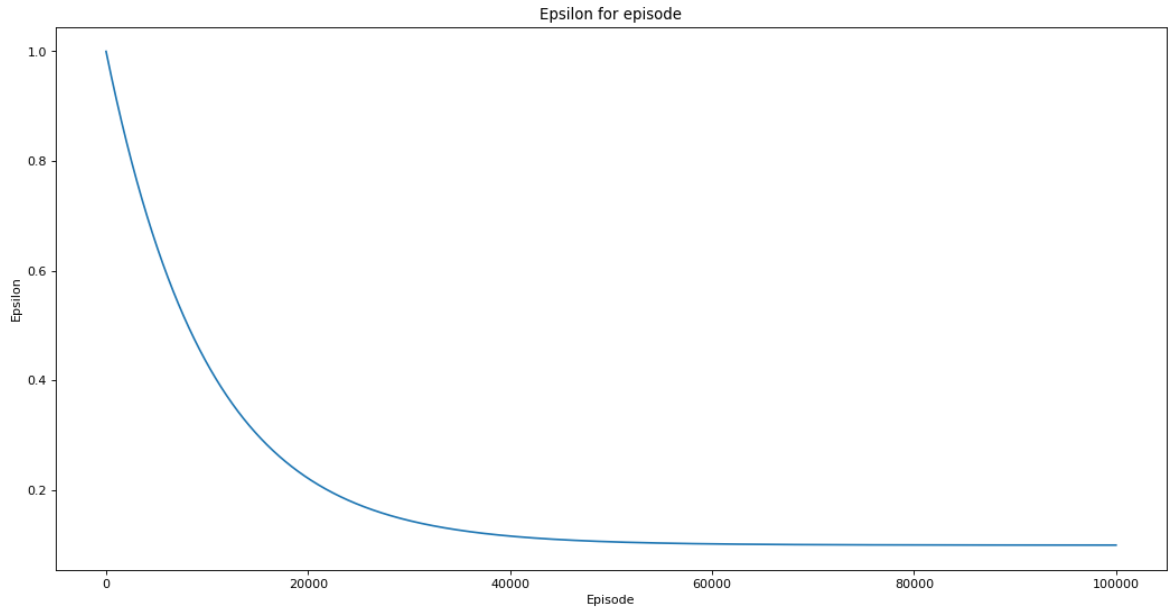
```
Training ongoing...
Successfully completed 100000 Pick-Drop episodes.
Training finished.

CPU times: user 3min 23s, sys: 3.25 s, total: 3min 26s
Wall time: 3min 25s
```

In [49]:
```
plt.figure(figsize=(16, 8), dpi= 80, facecolor='w', edgecolor='k')
plt.plot(savgol_filter(rewards, 1001, 2))
plt.title("Smoothened training reward per episode")
plt.xlabel('Episode');
plt.ylabel('Total Reward');
```



As we can see, rewards are getting more and more positive as the training continues and starts to converge once the model is trained. Depending on the graph, 25,000 epochs seem enough to train the agent successfully.

In [68]:
```
plt.figure(figsize=(16, 8), dpi= 80, facecolor='w', edgecolor='k')
plt.plot(all_epsilons)
plt.title("Epsilon for episode")
plt.xlabel('Episode');
plt.ylabel('Epsilon');
```



Epsilon is the exploration factor. It shuffles the model from exploration mode to exploitation mode. Exploration is finding new information about the environment. Whereas, Exploitation using existing information to maximize the reward. Initially, our driving agent knows pretty much nothing about the best set of driving directions for picking up and dropping off passengers. After good amount of exploration, we want our agent to switch to exploitation mode. Eventually, all choices will be based on what is learned. In this graph, we can see how it decreases over training. Lower values of epsilon switches the agent to exploitation model. This is a desired trend.

5: Analyze what the model has learned

We want to see if the model has learned to take optimum steps from any state to reach a goal position.

In [5]:
```
import pprint
# Drop off and pick up locations are indexed from 0-3 for the four
# different  locations.

# Let's initiate first forced state:
# Taxi at: (2,0), Passenger inside the taxi and Drop Off at R(0)(Pin
k)

statesList = [[2,0,4,0], [2,0,4,2], [2,3,4,3], [2,3,4,1]]
Actions = ["South", "North", "East", "West", "PickUp", "DropOff"]
exampleCount = 0

for state in statesList:
    exampleCount += 1
    exampleState = env.encode(*state)
    print"Example {}: Passenger inside taxi - To be dropped at locati
on: {}".format(exampleCount, state[3])
    env.s = exampleState # set the environment
    env.render()

    # Let's see the possible action space in this state
    env.P[exampleState]

    stateAction = dict(zip(Actions, q_table[exampleState]))
    max_key = max(stateAction, key=stateAction.get)
    print "Best Action:", max_key, "\nQ-Value:",round(stateAction[max
_key], 2)

    print "\n----------------------------------------------------------
\n"
```

```
Example 1: Passenger inside taxi - To be dropped at location: 0
+---------+
|R: | : :G|
| : | : : |
|_: : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
Best Action: North
Q-Value: 5.6

------------------------------------------------------------

Example 2: Passenger inside taxi - To be dropped at location: 2
+---------+
|R: | : :G|
| : | : : |
|_: : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
Best Action: South
Q-Value: 5.6

------------------------------------------------------------

Example 3: Passenger inside taxi - To be dropped at location: 3
+---------+
|R: | : :G|
| : | : : |
| : : :_: |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
Best Action: South
Q-Value: 5.6

------------------------------------------------------------

Example 4: Passenger inside taxi - To be dropped at location: 1
+---------+
|R: | : :G|
| : | : : |
| : : :_: |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
Best Action: North
Q-Value: 2.36

------------------------------------------------------------
```

As we can see after the training has been completed, the cab agent is able to predict the best step to reach destination accurately. 1. In example 1, cab with the passenger is at (2,0). Dropping destination for the case is R(0). The most sensible step is to move north. The model predicts this accurately as: | Best Action | || | Q-Value | --- | ---| --- | | North | || | 5.6 | 2. In example 3, cab with the passenger is at (2,3). Dropping destination for the case is B(3). The most sensible step is to move South. The model predicts this accurately as: | Best Action | || | Q-Value | --- | ---| --- | | South | || | 5.6 | 3. Similarly, in example 4, can is at (2,3) with the passenger inside. The dropping destination for the case is G(1). Both North and East have same q-values in this case, 2.36. Thus, the model hasn't overfit and predicts both the correct paths successfully.

6: Evaluating agent's performance after training.

Now we want to test our model on finding best routes between checkpoints and analyze penalties it is adding, if any!

In [15]:
```
total_epochs, total_penalties = 0, 0
episodes = 100 # Give 100 tests to the agent.
# frames = [] # for animation
testRewards = [] # Analyzing performance

sleep(5)

for episode in range(episodes):
    state = env.reset() # Reset the environment to a random new state
    epochs, penalties, reward = 0, 0, 0

    done = False # Episode completed?
    frames = []
    episodeRewards = []
    while not done: # While not dropped at correct destination
        action = np.argmax(q_table[state]) # Choose best action for a
given state
        state, reward, done, info = env.step(action) # Get reward, ne
w state and drop status

        if reward == -10: # Increment penalties if occurred
            penalties += 1

        epochs += 1 # Keep track of timesteps required to successfull
y reach the destination.

        # Put each rendered frame into dict for animation
        frames.append({
            'frame': env.render(mode='ansi'),
            'state': state,
            'action': action,
            'reward': reward
            }
        )
        episodeRewards.append(reward)
    total_penalties += penalties
    total_epochs += epochs
    testRewards.append(np.mean(episodeRewards))

    # Visualize the current episode.
    print_frames(frames, episode+1, True)
    sleep(.50)

print"\n----------------- Test Results -------------------- \n"
print("Results after {} episodes:".format(episodes))
print("Average timesteps per episode: {}".format(total_epochs / episo
des))
print("Average penalties per episode: {}".format(total_penalties / ep
isodes))
```

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


Episode Number 100
Timestep: 11
State: 85
Action: 5
Reward: 20

----------------- Test Results ---------------------

Results after 100 episodes:
Average timesteps per episode: 13
Average penalties per episode: 0
```

Test Results

* We can see from the evaluation, the agent's performance improved significantly and it incurred no penalties, which means it performed the correct pickup/dropoff actions with 100 different passengers. * Timesteps taken to reach destination has also reduced significantly, around 12 for each trip, which is what we would expect from a trained driver. * Thus, we see that cab agent learns to navigate in the grid city smoothly without incurring any penalties.