Teach a Taxi Cab to drive using Q learning

Donal Loitam

December 31, 2022

Abstract

This is the report of a project which I did during my tenure as an Epoch Club Member. My choice was to use a simple basic example, python friendly, and OpenAI-gym is such a very good framework to start with. The toy example I chose was the taxi-cab environment

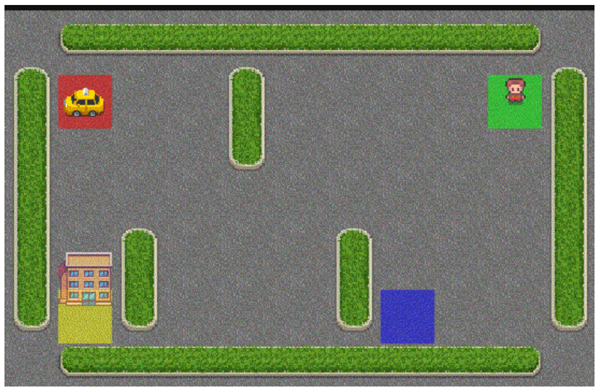


Figure 1: Toy Taxi Environment

1 The Environment

"There are four designated locations in the grid world indicated by R(ed), G(reen), Y(ellow), and B(lue). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the destination (another one of the four specified locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends." The map looks like :-

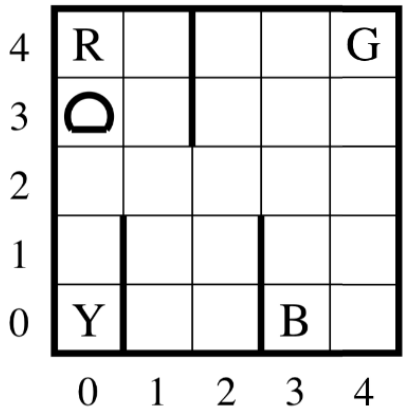


Figure 2:

As stated in the documentation, you drive a taxi-cab. The Smartcab's job is to pick up the passenger at one location and drop them off in another. Here are a few things that we'd love our Smartcab to take care of:

- Drop off the passenger to the right location.
- Save passenger's time by taking minimum time possible to drop off
- Take care of passenger's safety and traffic rules

There are different aspects that need to be considered here while modeling an RL solution to this problem: rewards, states, and actions.

2 Aspects to be designed

2.1 Rewards

Since the agent (the imaginary driver) is reward-motivated and is going to learn how to control the cab by trial experiences in the environment, we need to decide the rewards and/or penalties and their magnitude accordingly. Here a few points to consider:-

- High positive reward for a successful dropoff
- High penalty if it tries to drop off a passenger in wrong locations
- Slight negative reward for not making it to the destination after every time-step.

"Slight" negative because we would prefer our agent to reach late instead of making wrong moves trying to reach to the destination as fast as possible

This is how the reward is designed in the taxi environment

- +20 delivering passenger correctly
- -10 for illegal pickup and dropoff
- -1 per step unless other rewards are triggered

2.2 State Space:

The State Space is the set of all possible situations our taxi could inhabit. The state should contain useful information the agent needs to make the right action.

We can break up the parking lot into a $(5x5)$ grid, which gives us 25 possible taxi locations. These 25 locations are one part of our state space. Notice the current location state of our taxi is coordinate (3, 1).

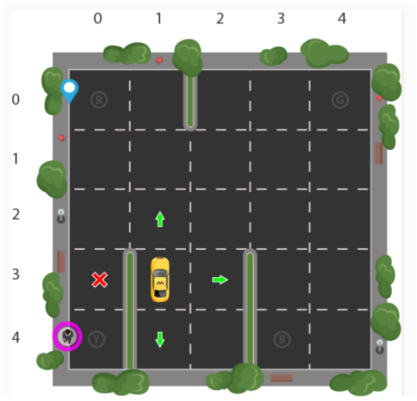


Figure 3: This frog was uploaded via the file-tree menu.

You'll also notice there are four (4) locations that we can pick up and drop off a passenger: R, G, Y, B or [(0,0), (0,4), (4,0), (4,3)] in (row, col) coordinates. Our illustrated passenger is in location Y and they wish to go to location R.

When we also account for one (1) additional passenger state of being inside the taxi, we can take all combinations of passenger locations and destination locations to come to a total number of states for our taxi environment; there's four (4) destinations and five (4 + 1) passenger locations.

So, our taxi environment has 5 x 5 x 5 x 4 = 500 total possible states.

2.3 Action Space:

The agent encounters one of the 500 states and it takes an action. The action in our case can be to move in a direction or decide to pickup/dropoff a passenger.
All in all, There are 6 discrete deterministic actions:

Index	Action
0	move south
1	move north
2	move east
3	move west
4	pickup
5	drop off

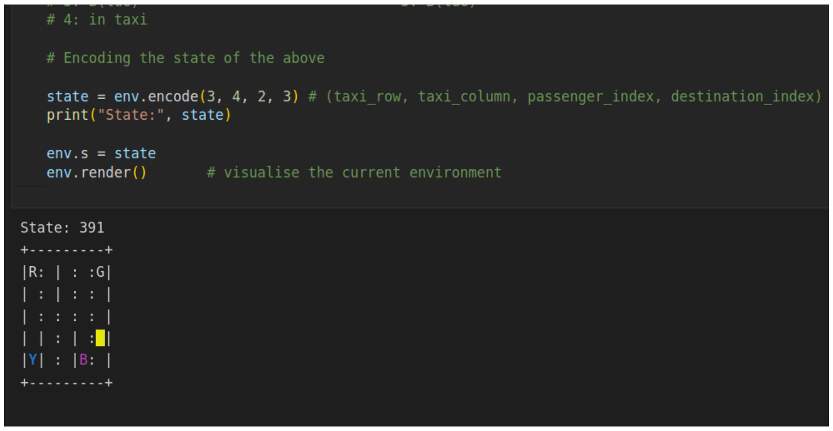
Table 1: Possible Action space

3 Observations :

Note that there are 400 states that can actually be reached during an episode. The missing states correspond to situations in which the passenger is at the same location as their destination, as this typically signals the end of an episode. Four additional states can be observed right after a successful episodes, when both the passenger and the taxi are at the destination. This gives a total of 404 reachable discrete states.

Each state space is represented by the tuple: **(taxi_row, taxi_col, passenger_location, destination)**

An observation is an integer that encodes the corresponding state. The state tuple can then be decoded with the "decode" method. In my code, I used encoding of state = 391



```
    # 4: in taxi

    # Encoding the state of the above

    state = env.encode(3, 4, 2, 3) # (taxi_row, taxi_column, passenger_index, destination_index)
    print("State:", state)

    env.s = state
    env.render()       # visualise the current environment
```

```
State: 391
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | :█|
|Y| : |B: |
+---------+
```

Figure 4: State encoding I used

4 Implementing Q Learning :-

- Q-learning lets the agent use the environment's rewards to learn, over time, the best action to take in a given state.
- In our Taxi environment, we have the reward table, P, that the agent will learn from. It does thing by looking receiving a reward for taking an action in the current state, then updating a Q-value to remember if that action was beneficial.
- The values store in the Q-table are called a Q-values, and they map to a (state, 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.
- Q-values are initialized to an arbitrary value(zero), and as the agent exposes itself to the environment and receives different rewards by executing different actions, the Q-values are updated using the equation:

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\Big(reward + \gamma \max_{a} Q(next\ state, all\ actions)\Big)$$

- α(alpha) is the learning rate ($0 < \alpha \leq 1$) - Just like in supervised learning settings, α is the extent to which our Q-values are being updated in every iteration.
- γ (gamma) is the discount factor ($0 \leq \gamma \leq 1$) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to 1) captures the long-term effective award, whereas, a discount factor of 0 makes our agent consider only immediate reward, hence making it greedy.

4.1 Interpretation of the Q equation:

We are assigning or updating, the Q-value of the agent's current state and action by first taking a weight $(1-\alpha)$ of the old Q-value, then adding the **learned value**. The **learned value** is a combination of the reward for taking the current action in the current state, and the discounted maximum reward from the next state we will be in once we take the current action.

Basically, we are learning the proper action to take in the current state by looking at the reward for the current state/action combo, and the max rewards for the next state. This will eventually cause our taxi to consider the route with the best rewards strung together.

4.2 Steps of Q-Learning

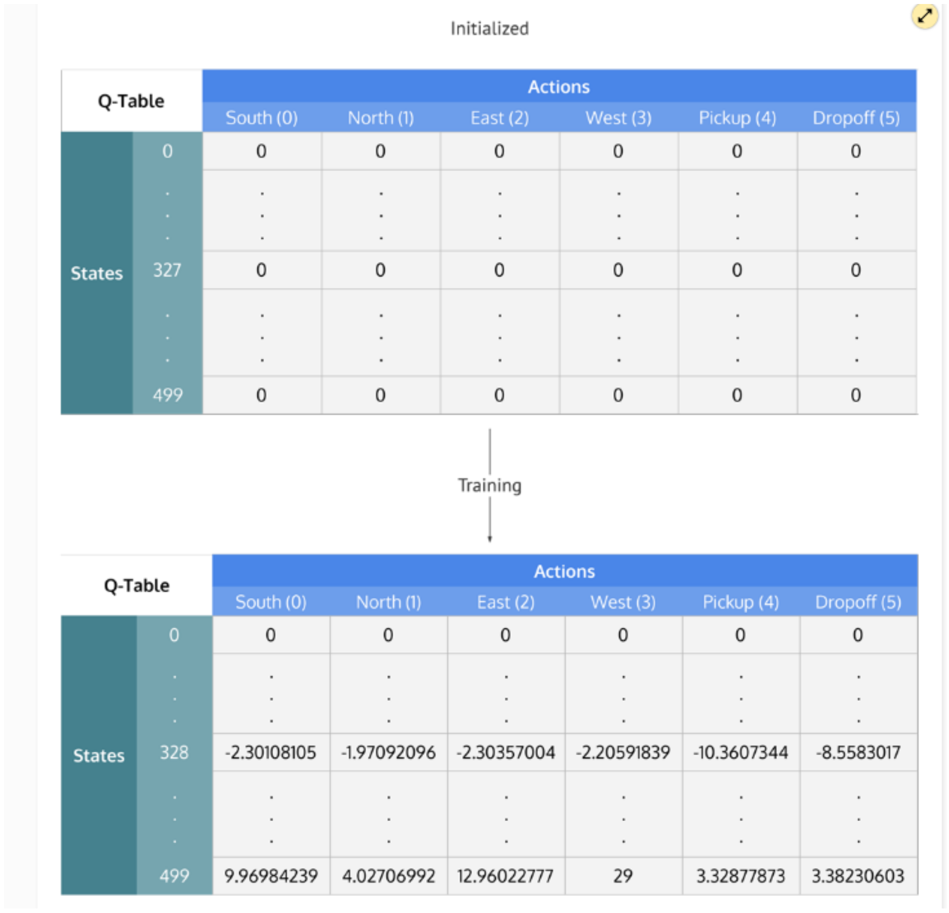
Breaking it down into steps, we get

- Initialize the Q-table by all zeros.
- Start exploring actions: For each state, select any one among all possible actions for the current state (S).
- Travel to the next state (S') as a result of that action (a).
- For all possible actions from the state (S') select the one with the highest Q-value.
- Update Q-table values using the equation.
- Set the next state as the current state.
- If goal state is reached, then end and repeat the process.

The way we store the Q-values for each state and action is through a Q-table

4.3 Q table:

The Q-table is a matrix where we have a row for every state (500) and a column for every action (6). It's first initialized to 0(in the code), and then values are updated after training.



Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	327	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure 5: Q-Table values are initialized to zero and then updated during training to values that optimize the agent's traversal through the environment for maximum rewards

4.4 Exploit or Explore :

"Should I go for the decision that seems to be optimal, assuming that my current knowledge is reliable enough? Or should I go for a decision that seems to be sub-optimal for now, making the assumption that my knowledge could be inaccurate and that gathering new information could help me to improve it?"

There's a tradeoff between exploration (choosing a random action) and exploitation (choosing actions based on already learned Q-values). We want to prevent the action from always taking the same route, and possibly overfitting, so we'll be introducing another parameter called ϵ**("epsilon")** to cater to this during training.
Instead of just selecting the best learned Q-value action, we'll sometimes favor exploring the action space further. Lower epsilon value results in episodes with more penalties (on average) which is obvious because we are exploring and taking randim actions

5 Conclusion and Results:

These are our trained model's performance evaluation after 100 episodes(pickup drop-off) :-

Metric	Q-learning
1	8.04
2	0.00
3	12.96

6 Limitations of Q learning:

Q-learning is one of the easiest Reinforcement Learning algorithms. The problem with Q-learning however is, once the number of states in the environment are very high, it becomes difficult to implement them with Q table as the size would become very, very large. State of the art techniques uses Deep neural networks instead of the Q-table (Deep Reinforcement Learning). The neural network takes in state information and actions to the input layer and learns to output the right action over the time. Deep learning techniques (like Convolutional Neural Networks) are also used to interpret the pixels on the screen and extract information out of the game (like scores), and then letting the agent control the game.

7 About Hyperparameters and Optimisation:

The values of (α) alpha, (γ) gamma, and (ϵ) epsilon were mostly based on intuition and some "hit and trial", but there are better ways to come up with good values. Ideally, all three should decrease over time because as the agent continues to learn, it actually builds up more resilient priors

- (the learning rate) should decrease as you continue to gain a larger and larger knowledge base.
- as you get closer and closer to the deadline, your preference for near-term reward should increase, as you won't be around long enough to get the long-term reward, which means your gamma should decrease.
- : as we develop our strategy, we have less need of exploration and more exploitation to get more utility from our policy, so as trials increase, epsilon should decrease.

.

8 Possible Future Improvements:

- Use DeepQ Learning and compare its performance with Qlearning
- Tune the hyperparameters(as discussed in the subsection 7)