Department of Electrical and Computer Engineering

ECE 559B Deep Reinforcement Learning

Project Report - Taxi Game With Reinforcement Learning

University of Victoria

Kaushik Jeyaraman

V00964926

Table of Contents

Introduction

Original Problem/Inspiration

The source for the original problem is obtained from Open AI gym. which simulates a simplified taxi environment, the goal is to teach a taxi agent to be to pick up and drop off passenger to their destination with the least possible number of moves and while avoiding roadblocks(walls).

The simplified environment is broken into a 5x5 Grid and contains roadblocks(walls). The taxi is the only vehicle in this environment. There are 4 possible locations where passengers can be picked up or dropped off, these are labelled as R(ed), G(reen), Y(ellow), B(lue). The agent's task is to pick up a passenger from one of these locations and drop the passenger at another location

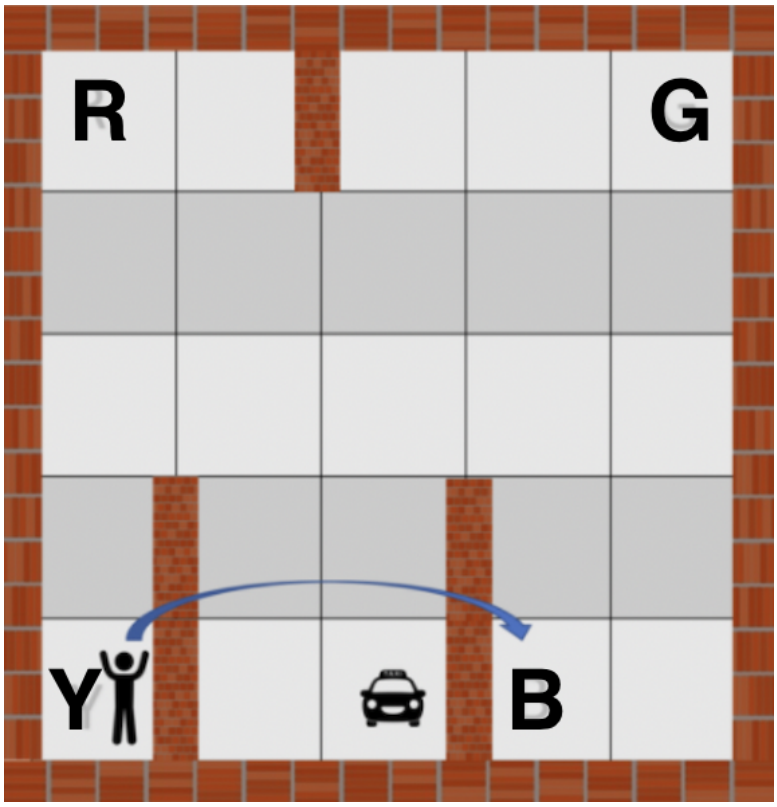


Fig 1. Original Taxi v3 by Open AI Gym

In the original 5x5 environment if the taxi hits the wall, it just gets a default penalty for one timestamp which is -1. It does not move, it remains in the same state.

Modified Problem

We modify the Open AI environment and make it bigger with a 9x9 gird & add plenty more walls. Also, we add one more constraint that the taxi can not hit the walls. If it does hit the wall, if it does hit the wall it's game over and the taxi gets a massive penalty.

The modified environment is shown below

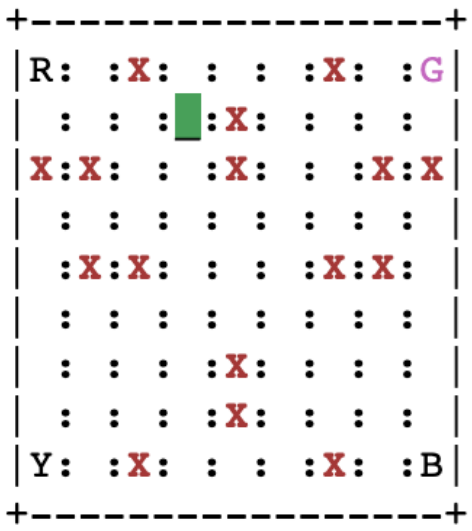


Fig 2: Modified Taxi Problem

Problem Formulation

Agent

The Taxi, which picks up passenger from the source and drops off passengers to their destination.

Environment

The environment consists of

- Road
- Barrier/Walls
- Passenger Location

- Source Location
- Destination Location

Agent's goals:

- Go to the source location & pick up the passenger
- Go to destination & drop off the passenger.
- Save passenger's time by taking a minimum number of steps possible.
- Carefully handle passenger's safety, avoid bumping into barriers/walls

Rewards

In Reinforcement Learning, the agent is motivated by the rewards. The Agent learns by trial and error and will try to maximize the rewards. Therefore, we need to decide the rewards and penalties and accordingly. We take the following points into consideration.

- For a correct pickup/drop-off, the agent should receive a high positive reward because this behaviour is highly desired.
- For Incorrect pickup/drop-off, the agent should get a large penalty since this is highly undesirable.
- For every delayed timestep to reach the destination, the agent must get a slight penalty. This is acceptable because it is better to reach late rather than reach the wrong destination.
- For hitting the wall there should be a massive penalty and should be the end of the episode.

We consider the rewards to be deterministic & define the following rewards

- Default Reward = -1
- Drop off at right location = +1000
- Pickup/Drop off at wrong location = -1000
- Hit the wall = -1000

State Space

The agent encounters a state & takes action based on its current state. The State Space is the set of all possible scenarios our taxi agent could encounter. The state contains useful information which is required by the agent to take optimal action. In this case, we have:

Information	**Possible Values**	**Number of possibilities**
Taxi row	0,1,2,3,4	9
Taxi column	0,1,2,3,4	9
Destination	R,G,B,Y	4
Passenger location	R,G,B,Y,Onboard	5

Table 1: State space

Size of the state space = 9 * 9* 4* 5 = 1620

Action Space

The agent encounters a state & takes action based on its current state. This action determines the next state the agent goes in. The set of all the actions that our agent can take in a given state is known as Action space. In our case, we have six possible actions.

- Up
- Down
- Left
- Right
- Pickup
- Drop-off

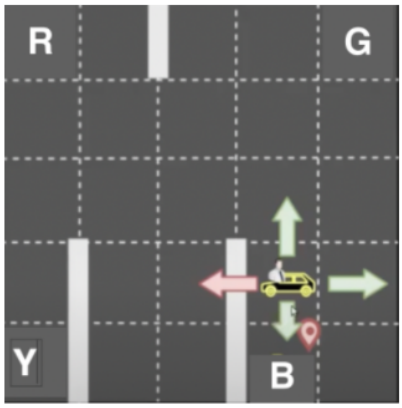


Figure 3: Action space

Based on the current state, the taxi may not be able to perform certain actions due to the presence of Barriers/walls. If the agent hit a wall the agent will be massively penalized & it will mark the end of the episode. In this way, the agent will learn to go around the wall.

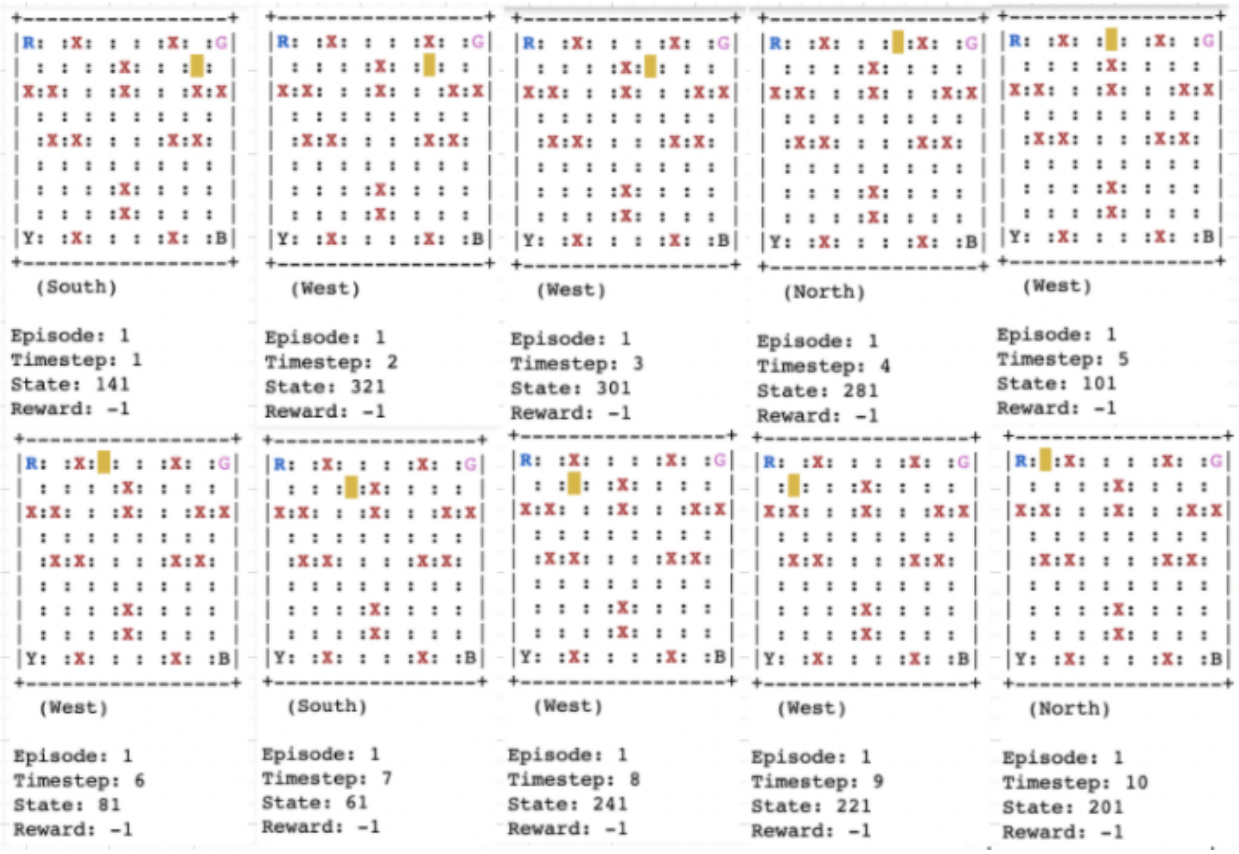
State Initialisation

Each episode starts with a random state. It begins with a random taxi location, a random passenger source location and a random passenger destination. An episode ends when the Taxi has dropped off the passenger at the correct destination.

Transition Dynamics

At each state, the taxi can take any one of the 6 actions. Based on the current state, the taxi may not be able to perform certain actions due to the presence of Barriers/walls.

A sample simulation is shown below



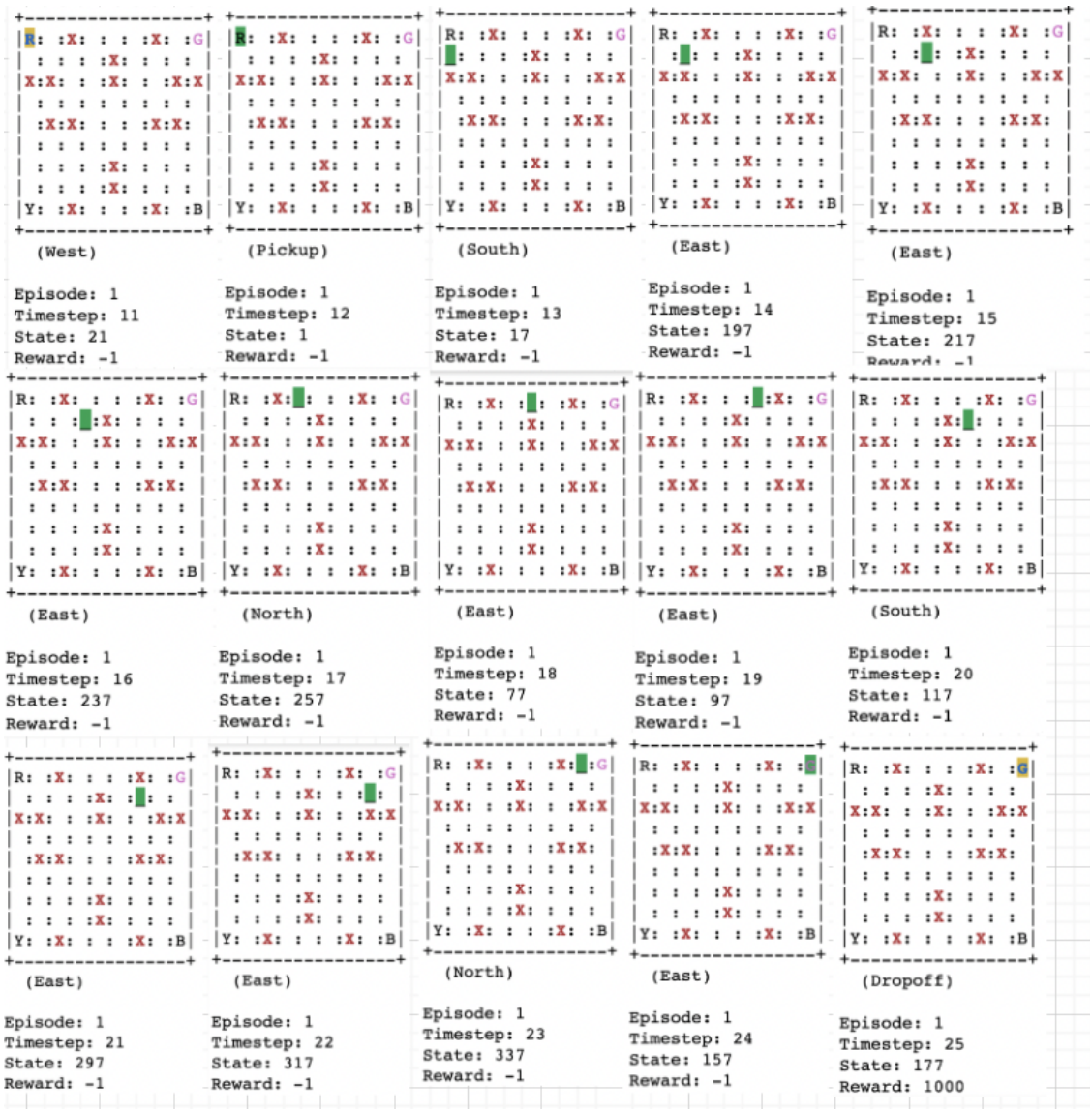


Figure 4: Sample Simulation

Information Available to Agent

The agent receives the following information while training/testing the model. Based on these information the agent takes action.

- Taxi row
- Taxi col
- Passenger location
- Passenger destination

Problem Classification

This problem is episodic, every episode starts with a random taxi location, passenger location and destination. An episode ends when a taxi drops off a passenger or hits the wall. The state space is finite. the state changes with every action/timestamp. The number of states spaces is 9*9*5*4=1620. (This is explained in Table.1 above)

Solution

Greedy?

The most common method to solve this kind of problem is the Greedy algorithm. This works on the greedy principle, i.e it chooses the best action based on immediate reward. However, there are things that are easy to do for instant gratification, and there are things that provide long term rewards. The idea is to avoid becoming greedy and seeking immediate rewards, instead, we optimize for maximum rewards throughout the course of the training.

The sequence of Action?

The sequence of action matters. The reward that the agent receives does not only depend on the current state but the entire history of states. Unlike supervised and unsupervised learning, time is important here. Reinforcement Learning lies between Supervised Learning & Unsupervised Learning

Q Tables

A Q-table is basically a look-up table that stores values that indicate the maximum predicted future rewards for a specific action in a specific state (known as Q-values). It will inform our agent that when it meets a specific situation, certain actions are more likely to result in higher rewards than others. It becomes a 'cheatsheet,' advising our agent on the best course of action.

Figure 5 below depicts how our 'Q-table' will appear:

- Each row represents a distinct state in the 'Taxi' world.
- Each column represents a possible action for our agent.
- Each cell corresponds to the Q-value for that state-action pair - a higher Q-value means a higher maximum reward our agent can expect to get if it takes that action in that state.

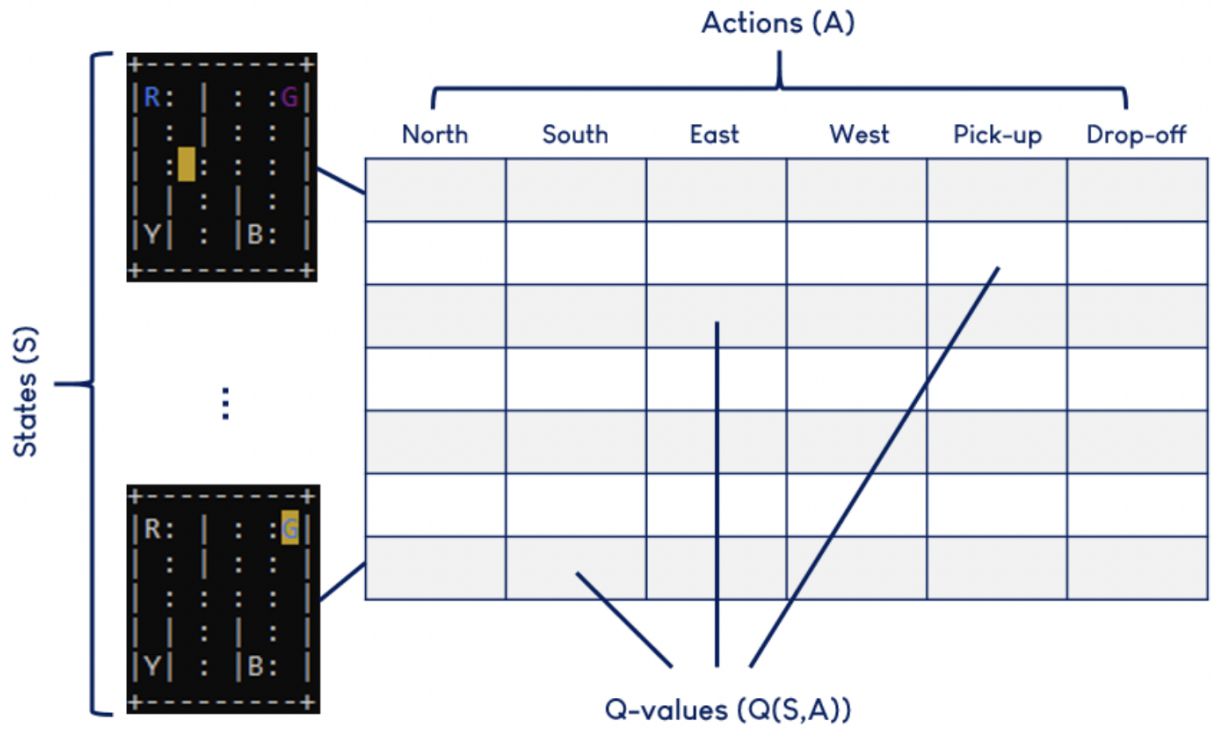


Figure 5: Q-tables

For problems with finite states, Q-table works well. For each state, it tells us which is the best possible action. Therefore, for this problem we tend to use Q-table based solutions, We shall use the two most popular algorithms Q-learning & SARSA.

Q Learning

Q Learning is a method of learning a policy that instructs the agent on the best course of action in a given situation. This method does not necessitate the use of a model of the environment. It may be used to solve problems involving stochastic transitions and rewards without requiring any modifications.

Q-learning determines an optimal policy for any finite Markov decision process (FMDP) starting from the current state, in the sense that it maximises the expected value of the total reward across all stages. Q-learning can determine an optimal action-selection policy for any FMDP given infinite exploration time and a partly-random policy. A function in the algorithm calculates the quality of a state-action combination:

$Q: S\ x\ A\ \ -> \ \ R$

Q is initialised to an arbitrary fixed value before the learning begins. The agent then selects an action at, observes a reward rt, enters a new state st+1 (which may be dependent on both the previous state and the current action), and Q is updated at each time t. The algorithm's core is built on a simple value iteration update that employs a weighted average of the old and new values:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}}$$

SARSA

The SARSA algorithm is quite similar to the Q learning algorithm. It is An on-policy learning algorithm, which means the agent interacts with the environment and adjusts the policy based on the actions made. The Q value for a state-action is updated whenever an error is made, it is adjusted by a factor of the learning rate a. These Q values represent the potential reward that can be obtained in the next time step after taking action at in state st, plus the discounted future reward received from the next state-action observation.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Difference Between Q-Learning & SARSA

In Q learning, we take action using an epsilon-greedy policy and, while updating the Q value, we simply pick up the maximum action. In SARSA, we take the action using the epsilon-greedy policy and also, while updating the Q value, we pick up the action using the epsilon-greedy policy. Figure 6 below shows the difference.

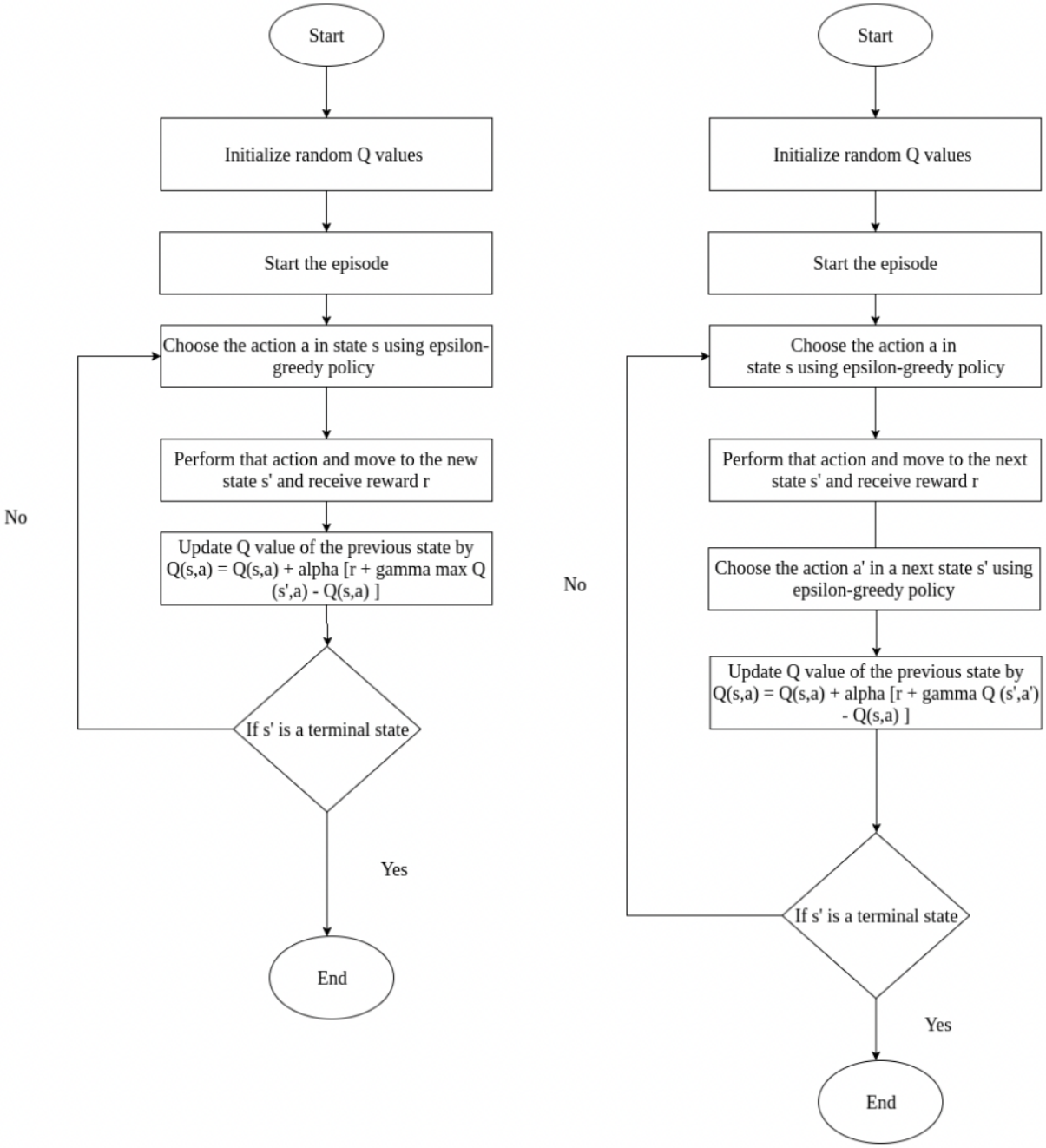


Figure 6: Comparison of Algorithms Q-learning(on left) vs SARSA (on Right)

Hyperparameters

- Learning Rate($0 < a \leq 1$): The learning rate influences how much freshly learned information trumps previously acquired information. A factor of 0 causes the agent to learn nothing (relying solely on prior knowledge), whereas a factor of 1 causes the agent to only evaluate the most recent information, disregarding existing knowledge in order to explore the unknown.
- Discount factor($0 < \gamma \leq 1$): The relevance of future benefits is determined by the discount factor. A factor of 0 causes the agent to be "myopic," focusing primarily on present benefits, whereas a factor of 1 causes it to seek a long-term high return.
- Exploration Vs Exploitation(ε): Exploration corresponds to choosing a random action & exploitation corresponds to choosing the best action based on current knowledge learned so far. These are mutually exclusive. There is a tradeoff between them. If we explore, we tend to gain knowledge and improve the rewards of the long term future. Whereas, if we exploit, we tend to improve the short term rewards by taking the best actions based on current knowledge, but this comes at the cost of not being able to improve the knowledge. To create a balance between them we another parameter called ε it prevents the agent from always taking the same route, and possibly prevents overfitting.

Implementation

1. We first create a new environment by inheriting an instance of Taxi-v3 by opening an AI gym and modifying it to create a 9x9 grid, add walls, set custom rewards and make hitting wall as the end of an episode.
2. We create 3 agents
    a. a Random Agent
    b. a Q-Learning Agent
    c. a Sarsa Agent
3. We tune the hyperparameters to find the best parameters
4. we build the agents with the best hyperparameters
5. We then compare the performance

The code can be found in the .ipynb notebook submitted with this report

Results

Hyperparameters

A search function was designed to modify the hyperparameters and come up with the best set of values. The function determines which parameters produce the optimum reward/timesteps ratio. We chose the reward/timesteps ratio because we want to establish parameters that allow us to earn the maximum payout as quickly as feasible. To save computational power, just five values in a given space are examined for each value. The tested space for each hyperparameter is as follows:

• alpha ∈ [0.6 , 1.0]

• gamma ∈ [0.6 , 1.0]

• decay rate = 0.001, the values decline by this factor over time as the agent continues to learn and develops a more confident model of behaviour.

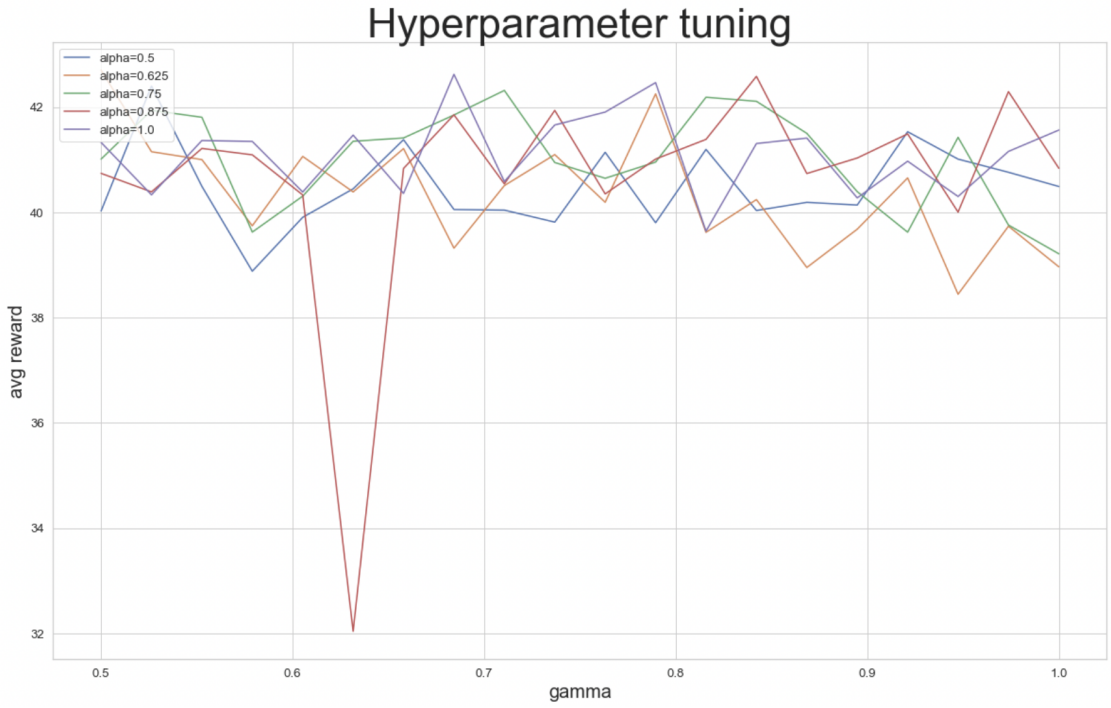


Figure 7: Hyperparameter Tunning for Q-Learning with decay 0.001

By trying out a larger range of values, it is could be possible to get a better approximation of the hyperparameters.

Decay of Epsilon

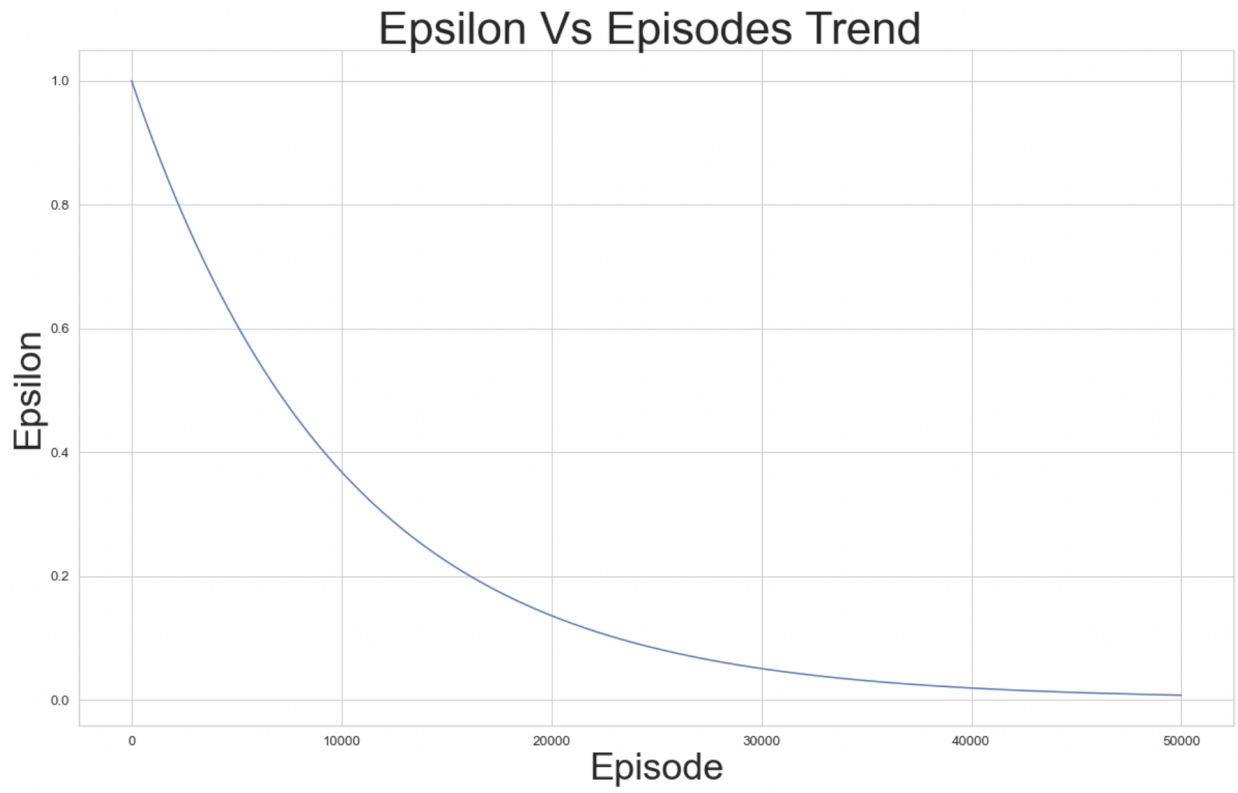


Figure 8: Change of Epsilon due to decay factor of 0.0001

Figure 8, shows how the value of epsilon changes with the number of episodes (this is with a decay factor of 0.001). This shows that exploration decreases as we gain knowledge.

Rewards

Figure 9 shows the rewards obtained after training the Q-learning and Sarsa agent on our environment for 50k episodes. Note that the learning curve is smoothened using savgol_filtersavgol_filter(rewards, window_length=1001, polyorder=2). We notice that both Q-learning and Sarsa perform well and converge on the optimal reward. However, Q-learning seems to be slightly better as it is seen to give slightly better rewards.

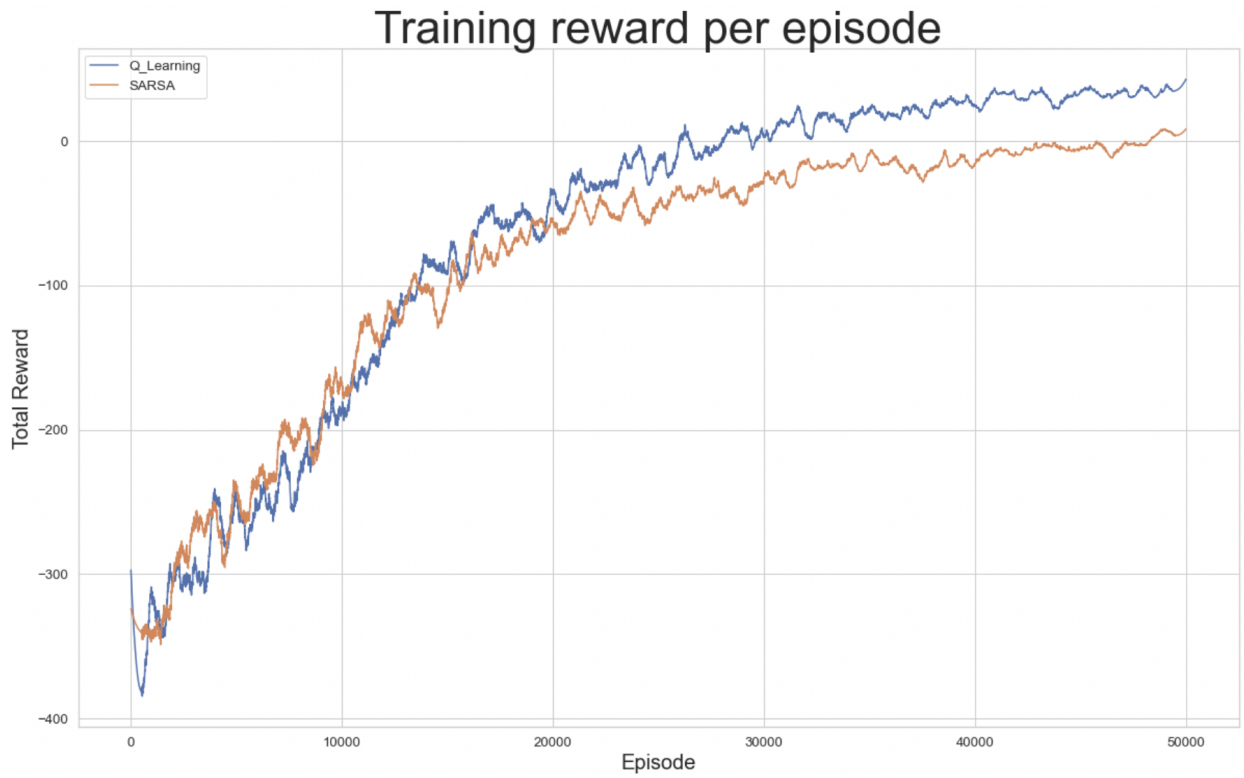


Figure 9: Rewards for Q-Learning & SARSA

Conclusion

After 50.000 episodes of training, we can say that each algorithm performs admirably. Each algorithm aims to reduce the overall number of timesteps while also increasing the episode's total payout. Both Q learning and SARSA have comparable results. This problem has 1620 potential states which make the use of Q and SARSA conceivable.

Learnings

- This was a great hands-on project to learn about reinforcement learning. Especially during the project search phase, I had to go through a variety of applications of Reinforcement Learning. I learnt where and how it is applied. It was cool to see its applications in self-driving cars, Industry robot automation, Trading & NLP.
- While implementing the project, since I had to modify the Open AI gyms' environment, I had to learn how it's been implemented, take that source code and tweak it to achieve my desired goal. Now I know how environments are built and how to make one myself.
- I also explored different Reinforcement Learning mechanisms to determine which one would efficiently solve this problem. The challenge was that there are many potential

solutions & I had to choose the best one. It turns out that environments with a small finite number of states space and action space Q-table work well.

- Of course, implementing 2 algorithms Q-Learning and SARSA gave me great insights to how these algorithms work. Another interesting learning was hyperparameter tuning to find the best possible epsilon, gamma and learning rate.

Area of Improvement

- The hyperparameter tuning(shown in figure 6) didn't seem to work out well. The graph was confusing, Something is potentially wrong with the approach. Given more time I would fix that.
- We could do deep Q-learning as that seems to work well for this problem as well. I did start to implement it, but given the time constraint I had to drop it
- We could make the game more interesting and challenging by adding moving traffic(robot cars), and adding the clause that the taxi should not hit any of the moving cars.

References

1. Towards Datascience Blog: Reinforcement Learning and Q learning —An example of the 'taxi problem' in Python
2. Analytics Vidhya Blog: Reinforcement Learning: Using Q-Learning to Drive a Taxi!
3. Learn Data Science Blog: Reinforcement Q-Learning from Scratch in Python with OpenAI Gym
4. Open AI Gym tutorial
5. Hierarchical Reinforcement Learning- Multi Agent Taxi

Code

The report & source code for this problem could be found in my GitHub portfolio. Please star my repository :D