INM707

Coursework Report

Deep Reinforcement Learning

By

Vrushang Bawne (Student Number: 210015700)

CITY
UNIVERSITY OF LONDON
EST 1894

MSc Artificial Intelligence 2021-22

[Term 2]

1 Contents

1. INTRODUCTION

Reinforcement learning is one of the three paradigms of machine learning, majorly concerned with how intelligent agents take the best actions in a given environment to accumulate more rewards and maximize them.RL [1] is being applied to many areas such as game theory, simulation-based optimization, swarm intelligence and many more.

Settings for normal RL problems are

1.1 State-space

This defines all possible scenarios or states an agent is in. State space should contain all important information to help the agent make the right decision. It's important to define all possible states

1.2 Action space

This refers to all possible actions an agent could take. For instance, in self-driving cars, an incoming vehicle to cause a head-on collision the action space would be either to turn right or left. Action space is numbered and has less possible outcome compared to state space

1.3 Transition probability

Given the current state (St), Transition probability refers to the probability of the agent going to the next state (St + 1) and the executed instruction rule (Ar). This is an important aspect as we can be able to determine if the agent is following provide commands and acting on them.TP is a necessary component in Q-learning

1.4 Reward

This is a state which acts as an incentive for the agent to reach the desired goal probably quickly and efficiently. A positive reward is given for the correct transition state and a penalty is provided for a wrong transition state.

2. PROBLEM OBJECTIVE

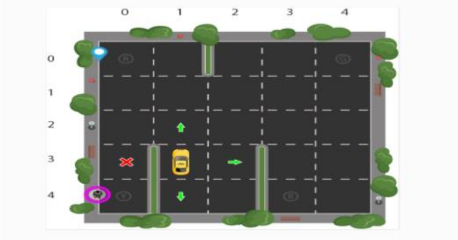
Over the past few years, machine learning has been used to obtain autonomous machines, mostly seen in self-driving cars and taxis. Major companies such as Tesla and goggle have been spearheading the innovation. In this project, we intend to develop a taxi system that can pick and drop off passengers at provided locations.

- ✓ The taxi shall be able to pick and drop off passengers at the right location
- ✓ The taxi shall take a short time possible to reach the destination
- ✓ The taxi shall follow traffic rules

3. PROBLEM SPECIFICATION

- ❖ State-space

  There are four designated locations on the grid

Red	0	
Green	1	
Yellow	2	
Blue	3	

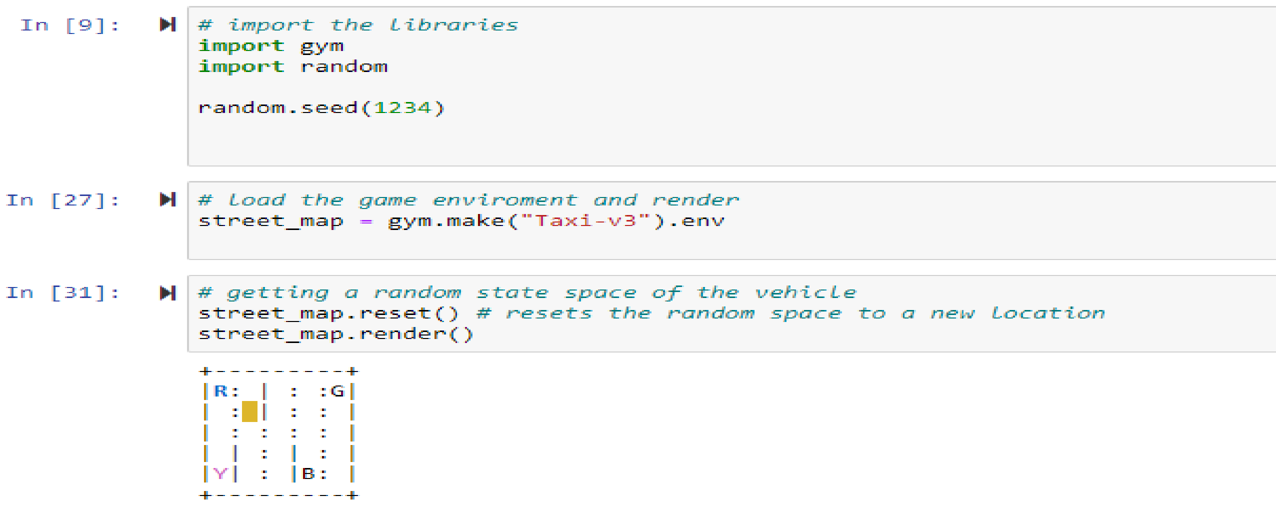
From the above figure, we can deduce there are 4 possible locations our passenger is including the taxi totalling to 5,4 destination locations and (5 x 5) possible taxi positions
This provides a total of 25 x 5 x 4 =500
We have 500 possible state spaces in a 5 by 5 grid provided by the gym environment builder

- ❖ Action space

  As the agent encounters the 500 state spaces, there are a defined number of action states it can only take, these are North, West, South, East, Pickup and, drop-off

4. IMPLEMENTATION

The implementation of this system environment is particularly provided by the OpenAI Gym library, which we shall be using.



```
In [9]:  # import the libraries
         import gym
         import random

         random.seed(1234)

In [27]: # load the game enviroment and render
         street_map = gym.make("Taxi-v3").env

In [31]: # getting a random state space of the vehicle
         street_map.reset() # resets the random space to a new location
         street_map.render()

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

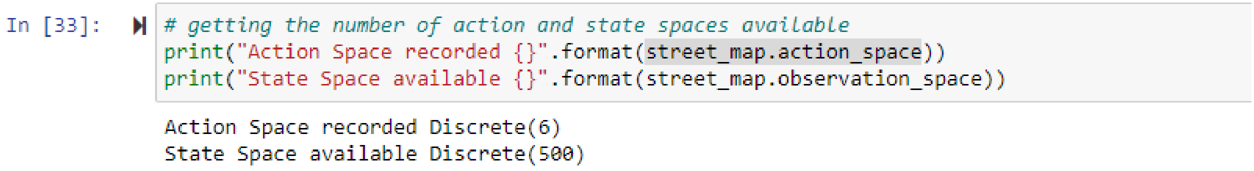
Fig 4.1 importing libraries and rendering game environment

From the above picture with four locations, the main goal is to pick up a passenger and drop him off at another location, once done successfully we get 10 points.

Every time we call the function a new start space state is created

- ✓ The squares represent the taxi, yellow without a passenger and green with a passenger.
- ✓ The line ("|") represents a boundary that the taxi cannot cross.
- ✓ R, G, Y, B are the possible pickup and destination locations.
- ✓ The blue letter represents the current passenger pick-up location, and the purple letter is the current destination.

State are defines with four parameters
(taxi_row,taxi_column,passenger_index,passenger_destination)
From the above figure, we can say state=(1,2,1,0)

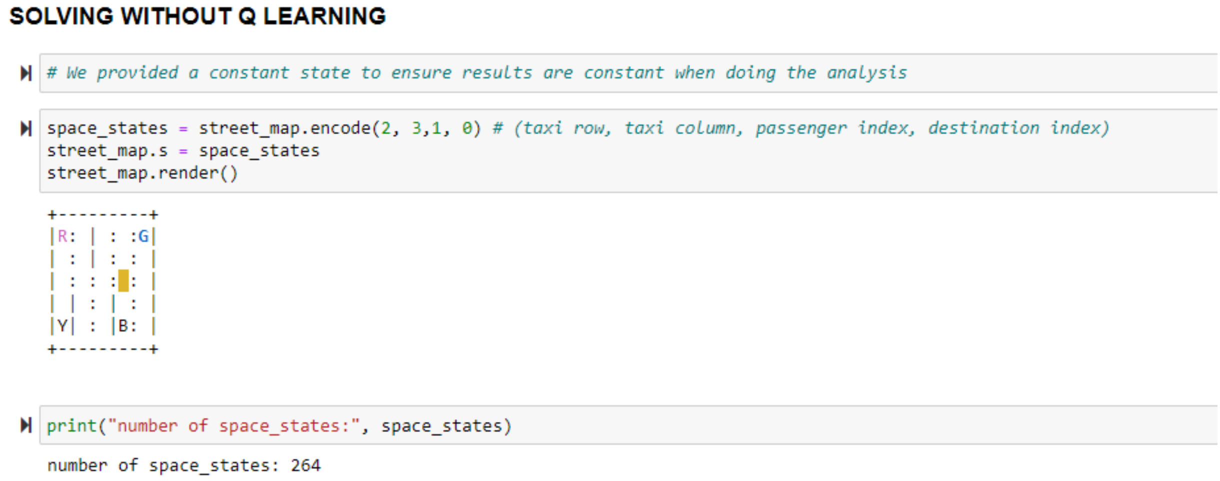


```
In [33]: # getting the number of action and state spaces available
         print("Action Space recorded {}".format(street_map.action_space))
         print("State Space available {}".format(street_map.observation_space))

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

Fig 4.2 shows action and state spaces

4.1 Without Deep Q learning



SOLVING WITHOUT Q LEARNING

```
# We provided a constant state to ensure results are constant when doing the analysis
```

```
space_states = street_map.encode(2, 3,1, 0) # (taxi row, taxi column, passenger index, destination index)
street_map.s = space_states
street_map.render()
```

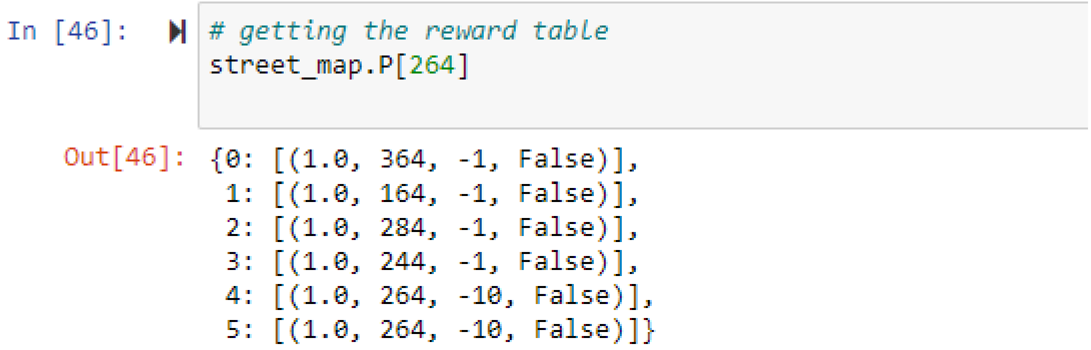
```
+---------+
|R: | : :G|
| : | : : |
| : : :█: |
| | : | : |
|Y| : |B: |
+---------+
```

```
print("number of space_states:", space_states)
```

```
number of space_states: 264
```

Fig 4.3 defines a constant state.

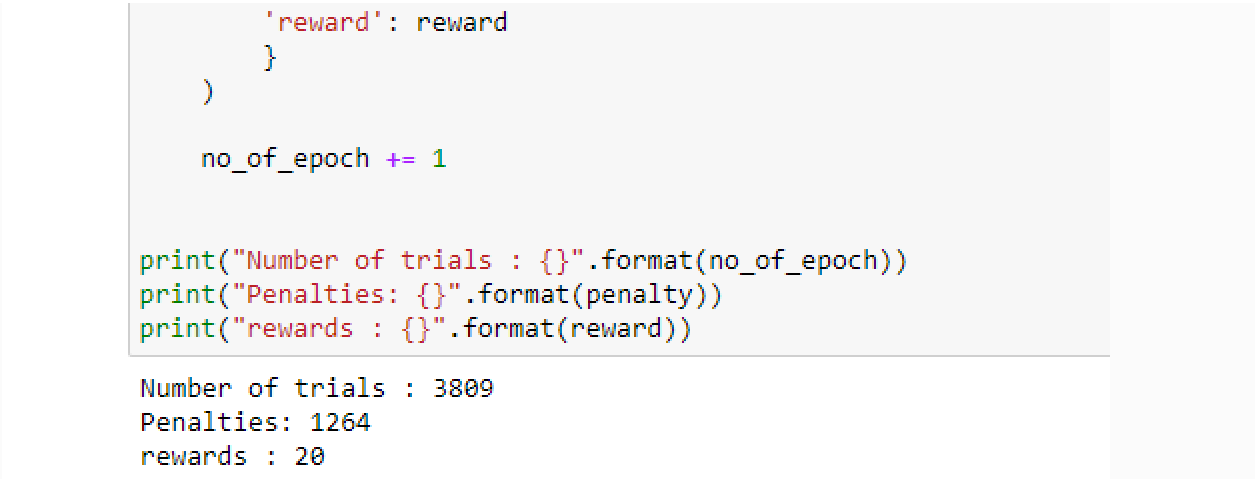
We had to set the environment manually to have a constant result when dealing with analysis and comparison for DQL .using the gym encode system the coordinates provided were (2,3,1,0) which in turn provided the total number of state spaces to 264



```
In [46]:  # getting the reward table
          street_map.P[264]

Out[46]: {0: [(1.0, 364, -1, False)],
          1: [(1.0, 164, -1, False)],
          2: [(1.0, 284, -1, False)],
          3: [(1.0, 244, -1, False)],
          4: [(1.0, 264, -10, False)],
          5: [(1.0, 264, -10, False)]}
```

Fig 4.4 Reward table generated



```
            'reward': reward
            }
        )

    no_of_epoch += 1


print("Number of trials : {}".format(no_of_epoch))
print("Penalties: {}".format(penalty))
print("rewards : {}".format(reward))
```

```
Number of trials : 3809
Penalties: 1264
rewards : 20
```

Fig 4.5 result

From the figure the agent just delivered two successful passengers after 3809 runs, it keeps making mistakes hence huge penalties incurred.

4.2 With Q learning

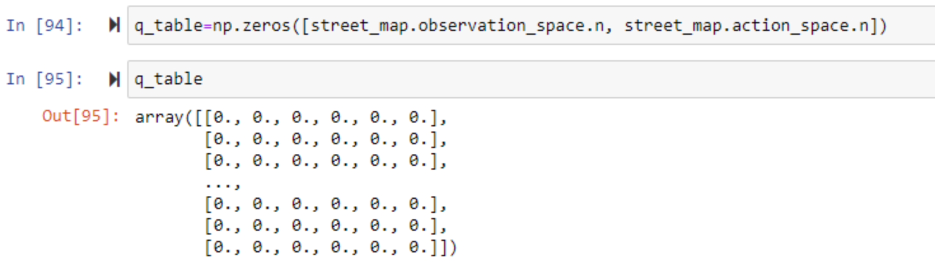
Q Learning is a type of Value-based learning algorithm. The agent's objective is to optimize a "Value function" suited to the problem it faces. We have previously defined a reward function R(s, a), in Q learning we have a value function that is similar to the reward function, but it assesses a particular action in a particular state for a given policy. It takes into account all future rewards resulting from taking that particular action, not just a current reward. In a Q learning process, we have a Q-table that stores the Q value for each state and each possible action, the agent explores the environment and makes an update to the Q values iteratively

Q learning lets the agents use rewards to learn, over time it takes the best action to deliver the desired outcome. From our problem, we have a reward table that the agent will learn from. It looks for actions that have a receiving reward from the current state and updates the Q-values to remember that the action was beneficial. The values stored in the Q table are mapped to state, action combination. The Q table is first initialized to zero then updated after training

The pseudo-code for the Q algorithm can be defined as

- ✓ Initialize the Q table with zero values
- ✓ Start exploration by selecting all possible actions for the current state
- ✓ Travel to the next state (S1) after an action (a)
- ✓ For all actions in the current state select the one with the highest Q value
- ✓ Update the Q table
- ✓ Set the next state as the current state
- ✓ If the goal is reached end the process

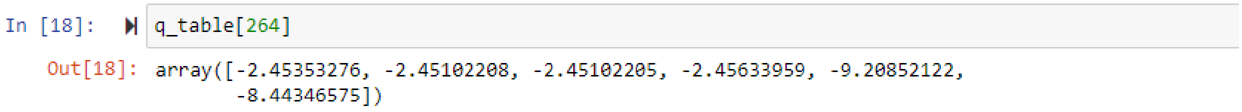
With 500 state spaces and 6 state actions, the Q table will be of dimension 500 x 6 matrix of zeros



```
In [94]:  q_table=np.zeros([street_map.observation_space.n, street_map.action_space.n])

In [95]:  q_table

Out[95]: array([[0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0.],
                ...,
                [0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0., 0.]])
```



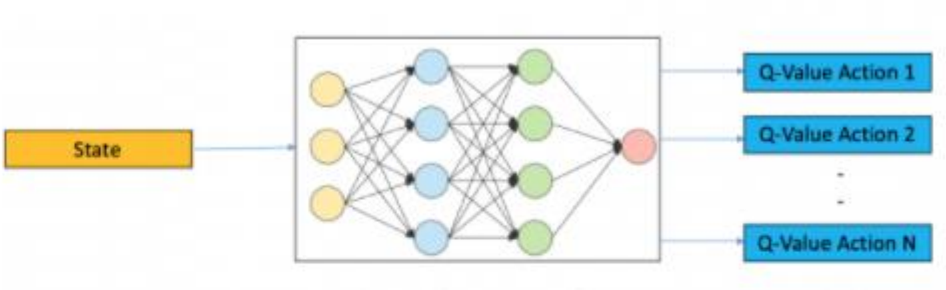
```
In [18]:  q_table[264]

Out[18]:  array([-2.45353276, -2.45102208, -2.45102205, -2.45633959, -9.20852122,
                 -8.44346575])
```

From the image, the best option is -2.45102205

4.3 Deep Q networks

In the Deep Q network we use a neural network to approximate the Q –value function, a state is given as the input and all possible q-values are generated

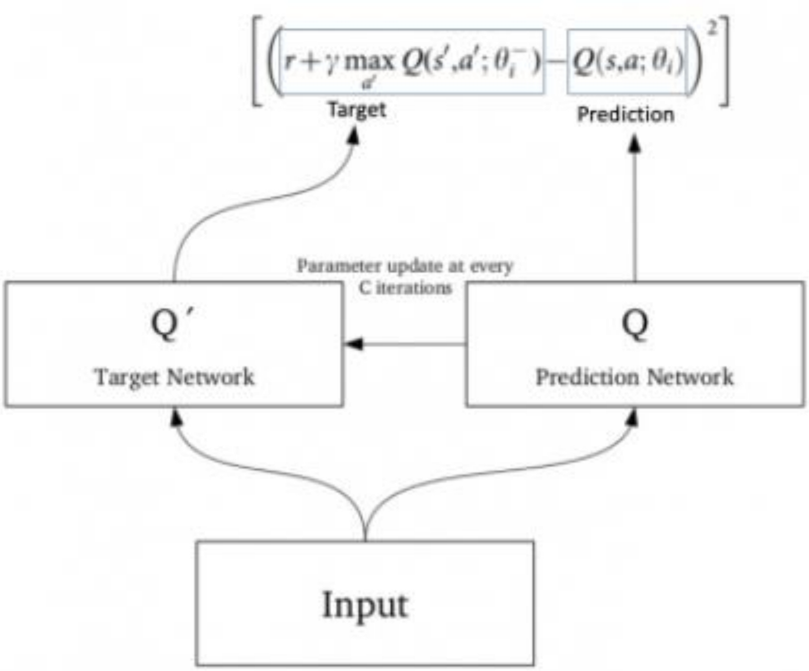


The steps involved in DQL are

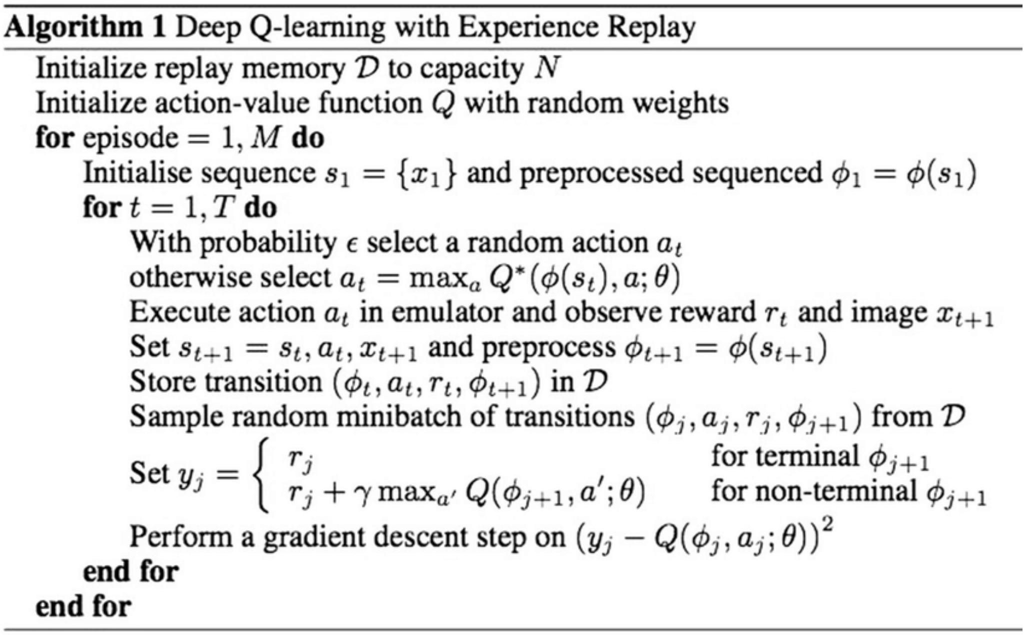
- ✓ Past experiences are stored in a memory state
- ✓ The maximum output of the Q-network determines the next state
- ✓ The loss function here is the mean squared error of the predicted Q-value and the target Q-value – Q*. This is a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation. we have:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\boxed{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)} - Q(S_t, A_t) \right]$$

The squared green part represents the target destination, from the equation, we can see it's predicting its value, the network then updates its gradient using backpropagation to converge into a final true destination


$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every
C iterations
Q′
Target Network
Q
Prediction Network
Input

Since we can’t have one network calculating the target value and the predicted values, we use two separate networks each with a specific task as shown in the figure above, this architecture provides a more stable training as the target value is kept constant and there are no more convergences



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

The steps involved in DQN are

- ✓ Feed the DQN with states and it will return all the Q values of all possible states

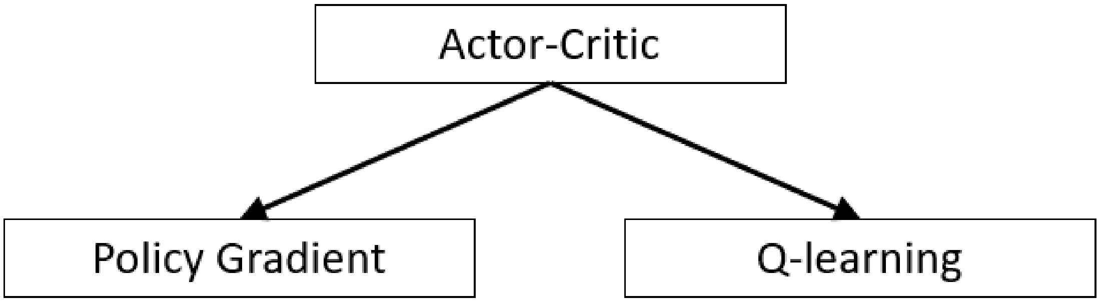
- ✓ Select an action using the epsilon, the action ‘a’ is selected and with a probability of one, the action with the maximum Q-value is selected such that a=argmax(Q(state,action,weight)
- ✓ The action is performed and we move to a new state to receive a reward
- ✓ Sample transitions and calculate the loss

$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

- ✓ Repeat the process with M number of episodes

4.4 Rlib algorithm Advantage Actor-Critic (A2C)

Also known as the hybrid method as it combines value optimization and policy optimization methods. That is Q learning and policy gradient approaches, with two networks that are actor-network and critic network.



The policy gradient function can be represented as

$$\Delta J(Q) = E_\tau \left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q (a_t, s_t) G_t \right]$$

This policy gradient is updated by A2C via Monte Carlo updates.

- The "Critic" estimates the value function. This could be the action-value (the Q value) or state-value (the V value).
- Critic: Q-learning algorithm that critiques the action that the Actor selected, providing feedback on how to adjust. It can take advantage of efficiency tricks in Q-learning, such as memory replay. [2]

Architecture

With a policy gradient architecture represented by vanilla policy represented

$$\Delta J(Q) = E_{\tau}\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t,\, s_t) G_t\right]$$

$$\Delta J(Q) = E_{\tau}\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t,\, s_t)\right] E_{r_{t+1},\, s_{t+1},\, \ldots,\, r_T s_T}[G_t]$$

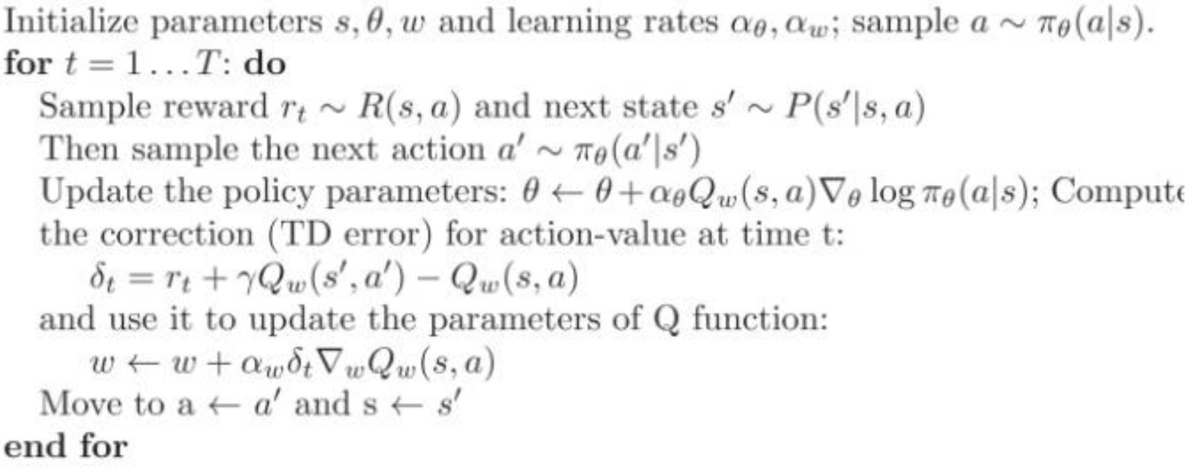
To get the Q value, the second expectation

$$E_{r_{t+1},\, s_{t+1},\, \ldots,\, r_T s_T}[G_t] = Q(s_t,\, a_t)$$

The final update function is

$$\Delta J(Q) = E_{s_0,\, a_0,\, \ldots,\, s_t,\, a_t}\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t,\, s_t)\right] Q(s_t,\, a_t) = E_{\tau}\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t,\, s_t)\right] Q(s_t,\, a_t)$$

Pseudocode



```
Initialize parameters $s, \theta, w$ and learning rates $\alpha_\theta, \alpha_w$; sample $a \sim \pi_\theta(a|s)$.
for $t = 1 \ldots T$: do
  Sample reward $r_t \sim R(s,a)$ and next state $s' \sim P(s'|s,a)$
  Then sample the next action $a' \sim \pi_\theta(a'|s')$
  Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s,a) \nabla_\theta \log \pi_\theta(a|s)$; Compute
  the correction (TD error) for action-value at time t:
      $\delta_t = r_t + \gamma Q_w(s',a') - Q_w(s,a)$
  and use it to update the parameters of Q function:
      $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s,a)$
  Move to a $\leftarrow a'$ and s $\leftarrow s'$
end for
```

4.5 Proximal policy optimization algorithm (PPO)

The policy gradient policy methods have also seen prevalence, one of the main algorithms in the policy gradient is the proximal policy optimization (PPO). It is implemented and maintained by the OpenAI group. Similar to TRPO is defines the probability ratio between the new policies and old policies commonly referred to as r(θ)

Where r(θ) can be represented as

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)}$$

There are two major variants in PPO that PPO-penalty and PPO variant

PPO-penalty: penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient throughout training so that it's scaled appropriately. [2]

PPO-clip: doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy. [3]

Being a policy algorithm, it updates its policy via

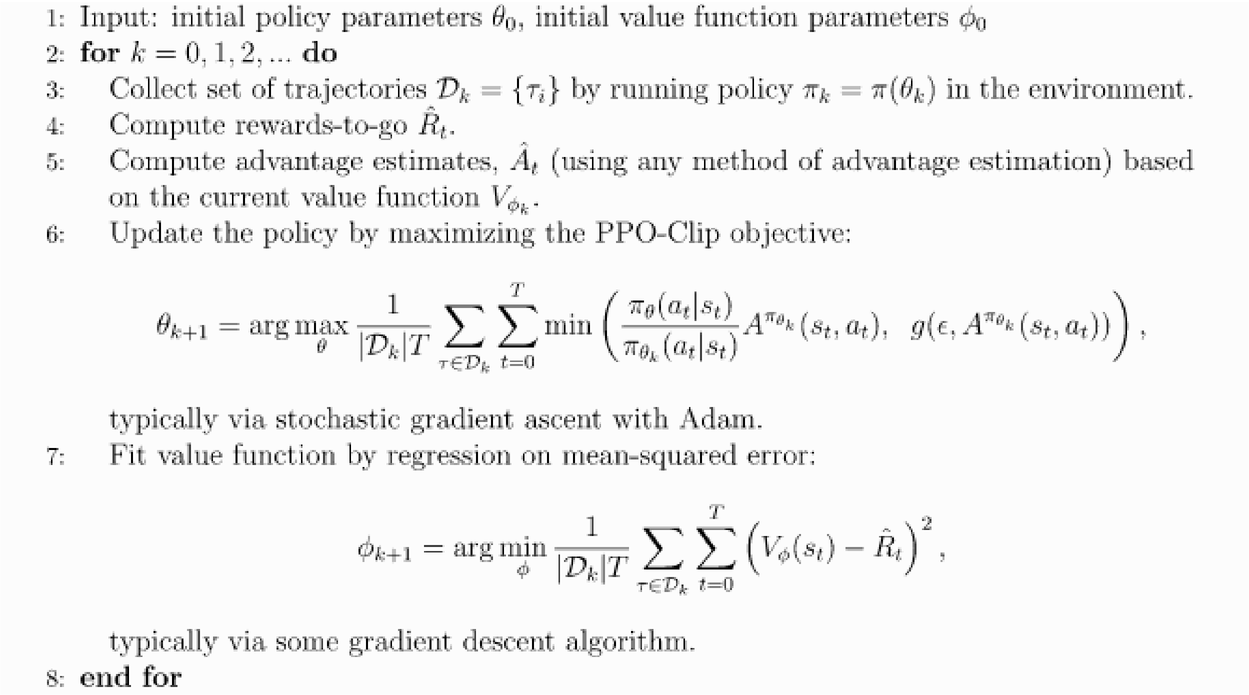
$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L(s,a,\theta_k,\theta) \right],$$

L is represented as

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \;\; \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right),$$

PPO uses stochastic policy when training, this means it explores space using the newest version of the stochastic policy. as it is exploring, and the policy update encourages it to exploit potential rewards that have been found

Pseudocode



```
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, ...$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
    on the current value function $V_{\phi_k}$.
    Update the policy by maximizing the PPO-Clip objective:

    $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$

    typically via stochastic gradient ascent with Adam.
    Fit value function by regression on mean-squared error:

    $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$

    typically via some gradient descent algorithm.
end for
```

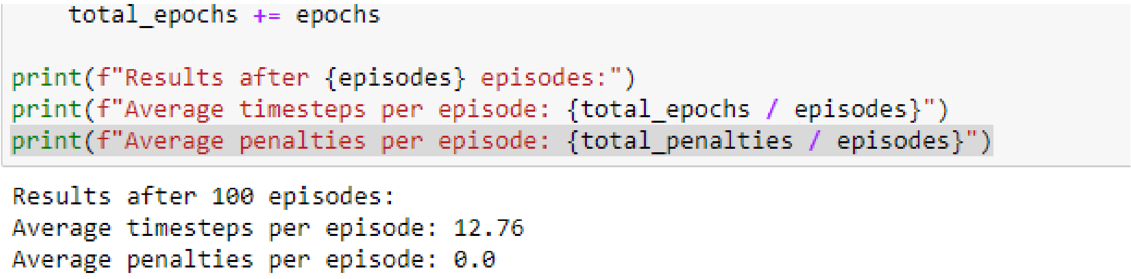
5. ANALYSIS

The main analysis evaluation shall be based on

- ✓ Number of penalties per trip
- ✓ Time take to complete the trip
- ✓ Number of steps taken to complete the trips
- ✓ Rewards

5.1 Without Q learning Vs. Q learning

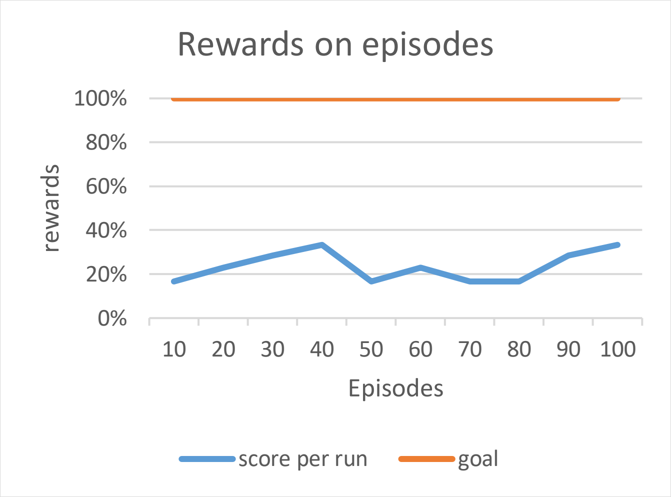
Metric	Without Q learning	With Q learning
Number of episodes	878	100
penalties	275	0
Time taken for the trips to end	20.6	12.5
Rewards	20	20



```
    total_epochs += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")
```

```
Results after 100 episodes:
Average timesteps per episode: 12.76
Average penalties per episode: 0.0
```

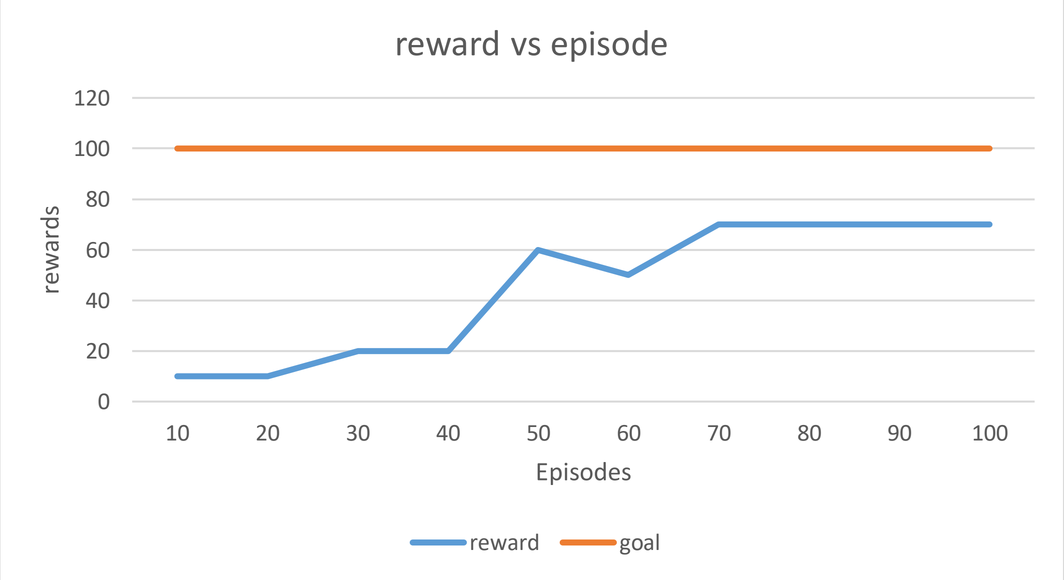


Q learning is the simplest Reinforcement learning algorithm, One of the most common problems experienced with this algorithm is the number of states which tend to be very high thus providing a huge q_table The problem is addressed by the use of DQN

5.2 Deep Q network

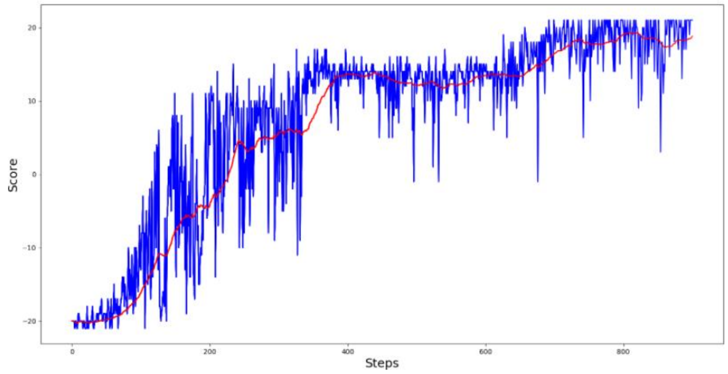
With a deep Q network, a total of 146 state spaces were generated, compared to Q learning with 246, fewer states indicates efficient learning, less time is taken and a higher possibility of rewards.

Metric	Results
Number of episodes	100
penalties	0
Time taken for the trips to end	6
Rewards	70



5.3 A2c

Metric	Reuslts
Number of episodes	100
penalties	0
Time taken for the trips to end	10
Rewards	30


Score
Steps