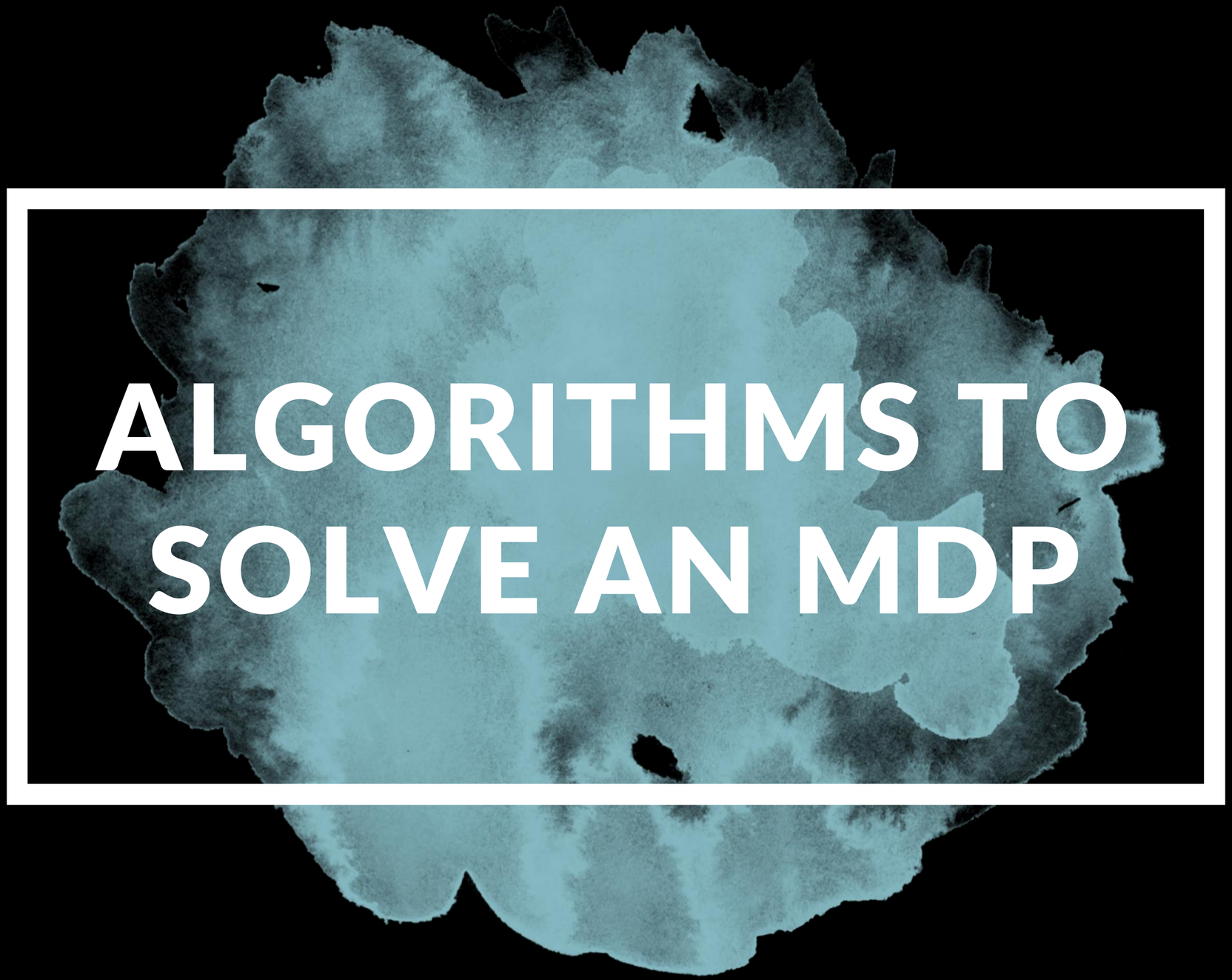

ALGORITHMS TO
SOLVE AN MDP

TO SOLVE AN MDP REFERS TO ESTIMATING THE OPTIMAL STATE VALUES OR THE OPTIMAL POLICY USING WHICH OUR AGENT CAN MAKE START DECISIONS REGARDING THE ACTIONS TO BE TAKEN IN THE ENVIRONMENT.

HOWEVER, AS WE KNOW THAT STATE VALUES ARE THE EXPECTED RETURNS FROM A STATE OVER A LARGE NUMBER OF FUTURE ACTIONS/TILL THE END OF A LARGE EPISODE, WE CANNOT DIRECTLY COMPUTE THEM BY FINDING ALL POSSIBLE PATHWAYS

HENCE WE DEPEND ON ITERATIVE OR RECURSIVE SOLUTIONS (DYNAMIC PROGRAMMING) WHERE THE VALUE OF ONE QUANTITY CAN BE ESTIMATED IN TERMS OF VALUE OF THE QUANTITY IN THE NEXT TIME STEP/ STATE.

MODEL-BASED AND MODEL-FREE METHODS

RECAP

MODEL BASED METHODS ALREADY HAVE ALL THE INFORMATION ABOUT THE ENVIRONMENT DYNAMICS, LIKE THE TRANSITION STATE PROBABLITIES, AND THE REWARDS REVCEIVED FOR EVERY STATE-ACTION PAIR, BUT THE MODEL FREE METHODS EXPLORE THE ENVIRONMENT TO SOLVE MDP

First, we'll discuss two model based algorithms:
Value Iteration
Policy Iteration

Value Iteration

In value iteration, we start with assuming random state values for each state, and using these random values, we iterate through different state action pairs using dynamic programming and correspondingly update the state values till convergence.

Algorithm 1: Value Iteration

Result: Find $V^{\pi^*}(s)$ and $\pi^*(s)$, $\forall s$

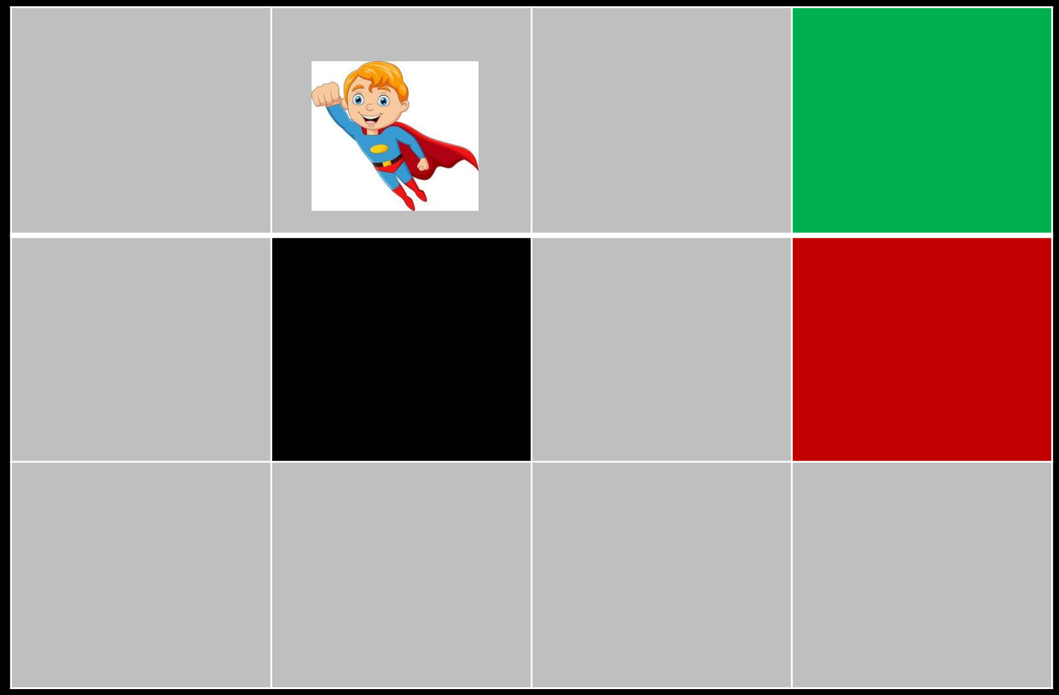
$V(s) \leftarrow 0, \ \forall s \in S;$
$\Delta \leftarrow \infty;$
while $\Delta \geq \Delta_\circ$ **do**
  $\Delta \leftarrow 0;$
  for *each* $s \in S$ **do**
    $temp \leftarrow V(s);$
    $V(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')];$
    $\Delta \leftarrow \max_a(\Delta, |temp - V(s)|);$
  end
end
$\pi^*(s) \leftarrow argmax_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s'), \forall s \in S;$
return $\pi^*(s), \forall s \in S$

We use this updation formula from the Bellman equation

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

Let’s create a simple game

Consider a 3*4 grid as our environment where our agent Superhero is trapped, he needs to reach the green cell from where he can gain higher powers and get out, but if he falls in the red cell, he dies due to deathtrap.



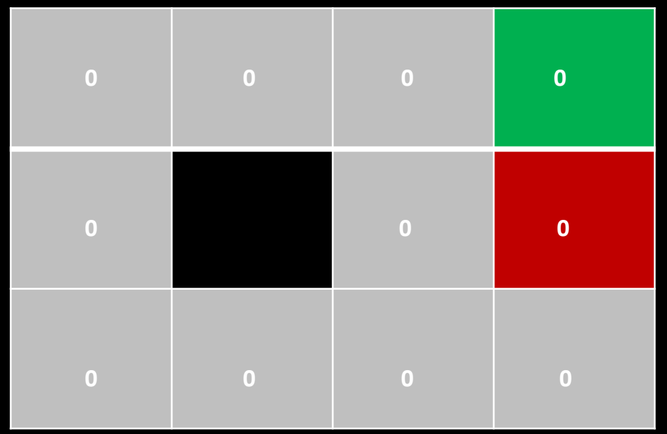
We need to view this as an RL model

11 STATES, 4 ACTIONS, +1,-1 REWARD

MAY THE BEST STATE ~~WIN~~ BE SELECTED

Step 1: Initialisation

In the first step, we have no clue about the state values so we just randomly allot the states, or bring them all to 0.



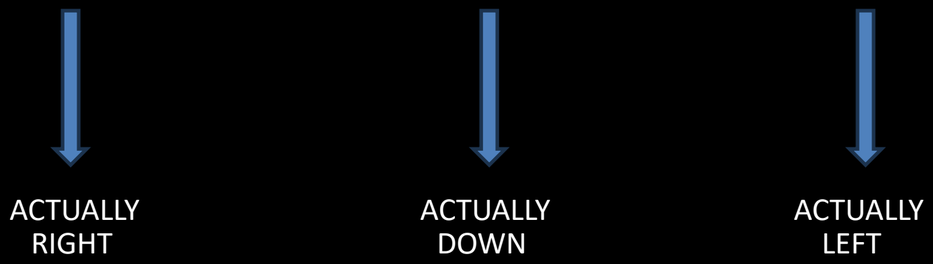
Step 2: Q value estimation

Hence, in next iteration, we find our state value to be as follows
Next, we use these updated values to find the Q values for all state-action pairs in the model.
First let's focus on one particular state: (1,3)
Now, Q values for this state:

(1,3),RIGHT = 0.8*(0 + 0.9*1) + 0.1*(0 + 0.9*0) + 0.1*(0 + 0.9*0) = 0.72

(1,3),DOWN = 0.1*(0 + 0.9*1) + 0.8*(0 + 0.9*0) + 0.1*(0 + 0.9*0) = 0.09

(1,3),LEFT = 0.1*(0 + 0.9*1) + 0.1*(0 + 0.9*0) + 0.8*(0 + 0.9*0) = 0.09



0	0	0	1
0		0	-1
0	0	0	0

Step 3: State Value Updation

Now, we repeat step 2 and 3 with our updated state value table, and in the further iterations we get

0	0	0.72	1
0	0	0	-1
0	0	0	0

0	0.52	0.78	1
0	0	0.43	-1
0	0	0	0

We keep on repeating our iterations until the difference between the current and the previous iteration is negligible/less than the threshold value. Hence, we get to a convergent value of the state value function, which is our final optimal state value function!

Step 4: Optimal Policy

With our optimal state value function, we can easily make a deterministic optimal policy such that from any state, the corresponding action should be such that the agent moves to the state with the maximum state value of all other possibilities.

Here, hence since value of state(1,3) will be the largest among all other neighboring cells of state(1,2), the policy asks the Superhero Pranay to move right from the cell (1,2) and similarly from state(1,3) to state(1,4), where he wins!

Policy Iteration

Unlike value iteration, in policy iteration, we start with assuming a random policy for the agent as we do not know which action would be optimal.

Then for this policy, we calculate the state values which are initially all random. After calculating the new state values, we update our policy accordingly and iterate till convergence.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

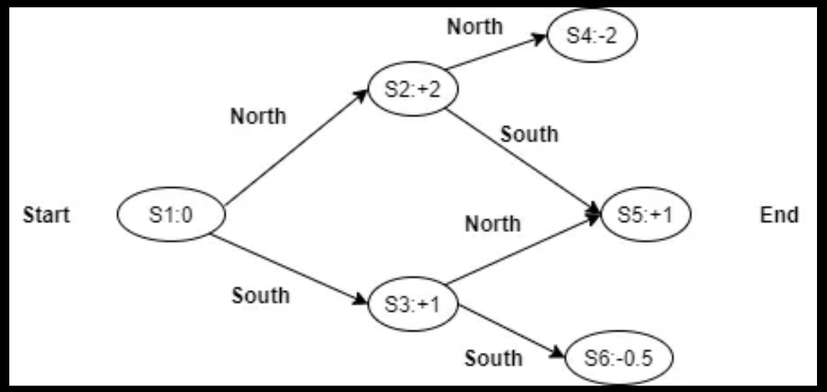
Policy iteration has 2 main steps:
1) Policy Evaluation
2) Policy Improvement

Policy Evaluation

Now, we can randomly initialise our state values to 0 after a random policy and using our infamous Bellman Equation, we can compute the state values for the next iteration.

$$V(s) = r(s) + \gamma * \max\left(\sum_{s',r} p(s', r|s, \pi(s) * V(s')\right)$$

Example:



State Diagram

$T[A(North)]$

	$S1$	$S2$	$S3$	$S4$	$S5$	$S6$
$S1$	0	0.8	0.2	0	0	0
$S2$	0	0	0	0.8	0.2	0
$S3$	0	0	0	0	0.8	0.2
$S4$	0	0	0	0	0	
$S5$	0	0	0	0	0	
$S6$	0	0	0	0	0	

$T[A(South)]$

	$S1$	$S2$	$S3$	$S4$	$S5$	$S6$
$S1$	0	0.2	0.8	0	0	0
$S2$	0	0	0	0.2	0.8	0
$S3$	0	0	0	0	0.2	0.8
$S4$	0	0	0	0	0	
$S5$	0	0	0	0	0	
$S6$	0	0	0	0	0	

Transition Probabilities

Let our random policy be to take North everytime for S1, S2 and S3.

V[S1] = 0; V[S4] = -2

V[S2] = 2; V[S5] = 1

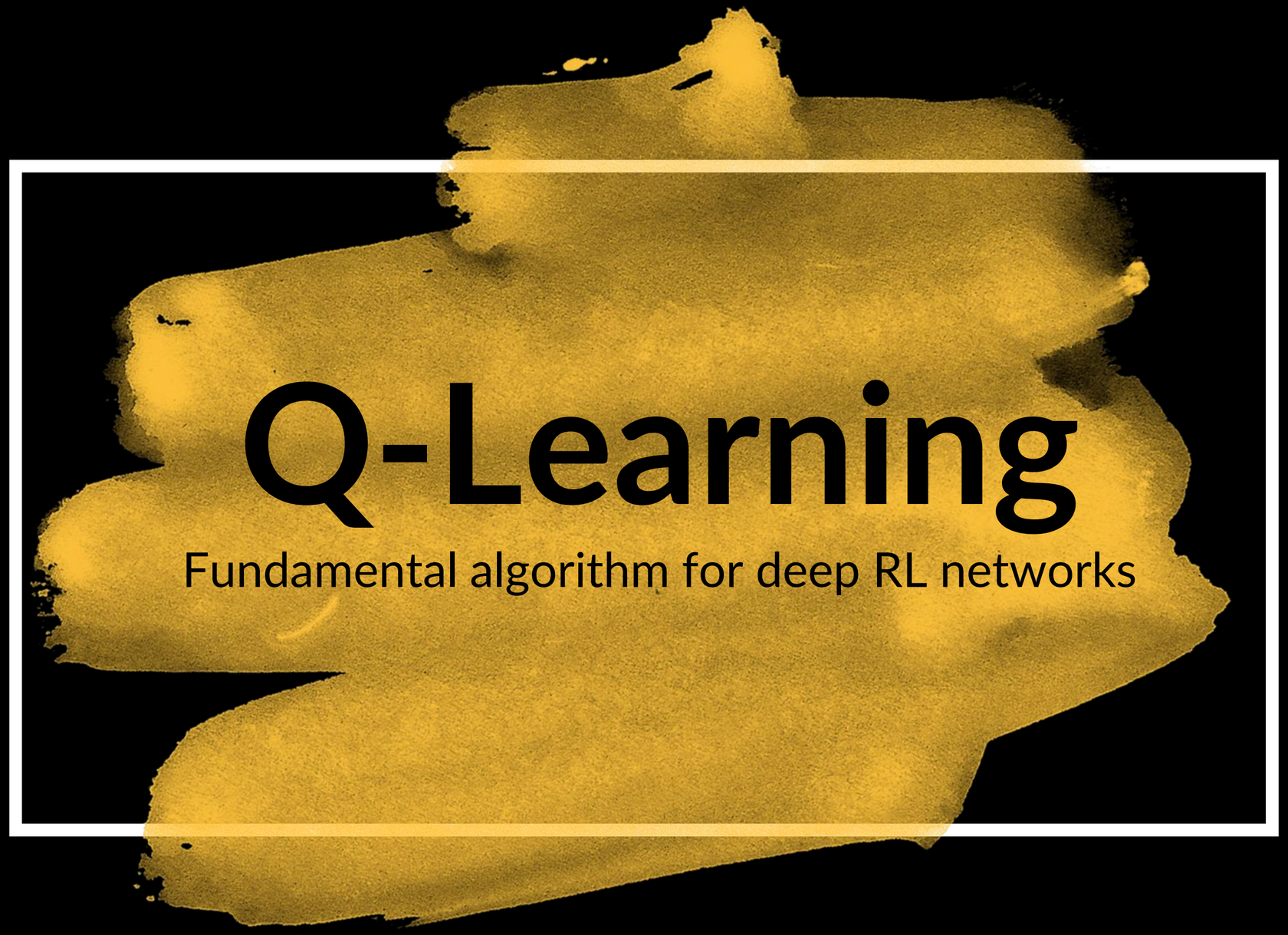
V[S3] = 1; V[S6] = -0.5

Policy Improvement

State	Action	V[S]	Max Action
S1	North	V[S] = 0.8 * 2 + 0.2 * 1 = 1.8	
	South	V[S] = 0.2 * 2 + 0.8 * 1 = 1.2	
			North
S2	North	V[S] = 0.8 * -2 + 0.2 * 1 = -1.4	
	South	V[S] = 0.2 * -2 + 0.8 * 1 = 0.4	South
S3	North	V[S] = 0.8 * 1 + 0.2 * -0.5 = 0.7	North
	South	V[S] = 0.2 * 1 + 0.8 * -0.5 = -0.2	

After computing the state values, before directly jumping to the next iteration, we first check if our policy is optimal, we do that by changing our policy such that our agent follows the action leading to a state with maximum state value for a given initial state.

In this case, we can check that for state 2, the optimal action according to Bellman Equation would be to move South, hence we change our policy to {N,S,N}.


Q-Learning
Fundamental algorithm for deep RL networks

Q-LEARNING IS A POPULAR MODEL-FREE REINFORCEMENT LEARNING ALGORITHM USED TO FIND AN OPTIMAL POLICY FOR AN AGENT IN AN ENVIRONMENT WITHOUT REQUIRING A MODEL OF THE ENVIRONMENT'S DYNAMICS.

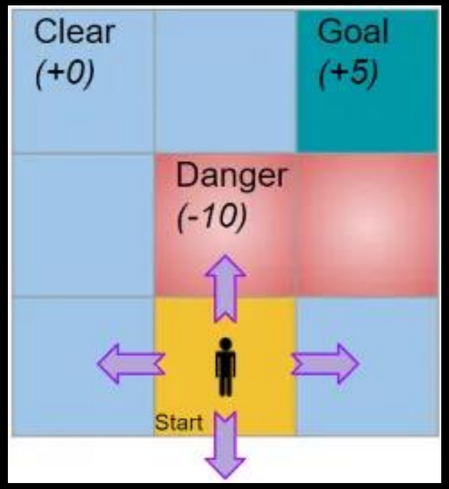
QUITE INTERESTING IS ALL I'LL SAY

Step 1: Initialisation

In the first step, we have no clue about the state-action Q values so we just randomly allot the states, or bring them all to 0.

Consider another simple grid game, a 3x3 grid, where the player starts in the Start square and wants to reach the Goal square as their final destination, with danger cells.

This problem has 9 states since the player can be positioned in any of the 9 squares of the grid. It has 4 actions. So we construct a Q-table with 9 rows and 4 columns.



	Left	Right	Up	Down
(1,1)	0	0	0	0
(1,2)	0	0	0	0
(1,3)	0	0	0	0
(2,1)	0	0	0	0
(2,2)	0	0	0	0
(2,3)	0	0	0	0
(3,1)	0	0	0	0
(3,2)	0	0	0	0
(3,3)	0	0	0	0

Step 2: Action!

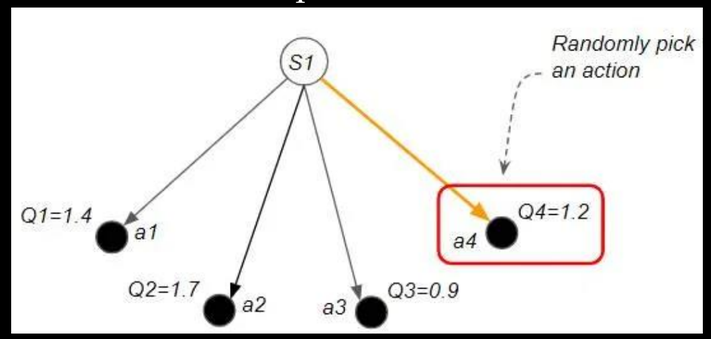
Unlike model-based methods, here in the Q –Learning algorithm, we need to actually interact with the environment to learn something about it. Hence, from whichever state we are in, we need to pick an action correspondingly.
How do we choose the action?

Here, we use the concept of exploitation exploration trade-off.

Step 2: Action!

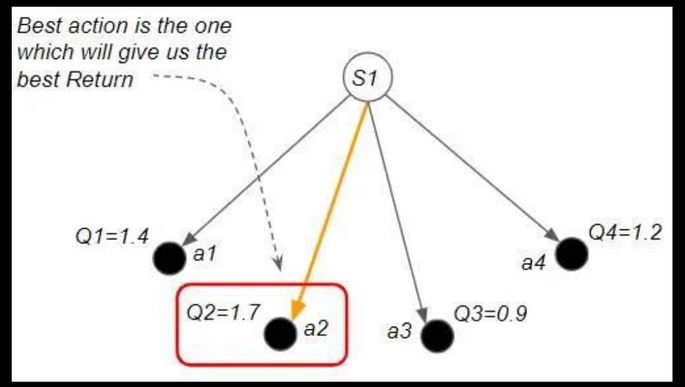
EXPLORATION

When we first start learning, we have no idea which actions are ‘good’ and which are ‘bad. So we go through a process of discovery where we randomly try different actions and observe the rewards.
Hence, exploration refers to the policy of randomly choosing an action from any particular state so that w “explore” the different options.



EXPLOITATION

On the other end of the spectrum, when the model is fully trained, we have already explored all possible actions, so we can pick the best actions which will yield the maximum return. Hence, exploitation refers to the policy of “exploiting” our knowledge of the best action and following that rather than taking risks.

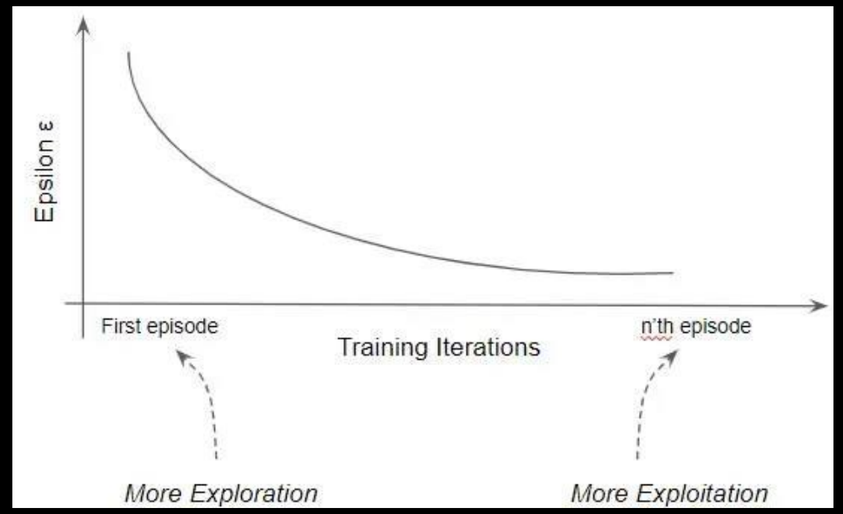


Seeing that both exploration and exploitation have their benefits, we need to find a perfect balance between the two.

Step 2: Action!

Here we use a unique policy called as ***The ε-Greedy.***

Now, whenever it picks an action in every state, it selects a random action) with probability ε. And similarly, with probability ‘1 — ε’, it selects the best.



In the first few iterations, we have learnt close to nothing about the model, hence it is better to explore the different options rather than relying on our little knowledge about the model. Hence ε is close to 1 in the beginning encouraging exploration. Towards the end of an episode or a high number of iterations, we have learnt almost everything about the model and hence we can focus on our knowledge of the model to extract the best actions and gain greater rewards. Hence, at very large number of iterations, the ε boils down to 0, encouraging exploitation.

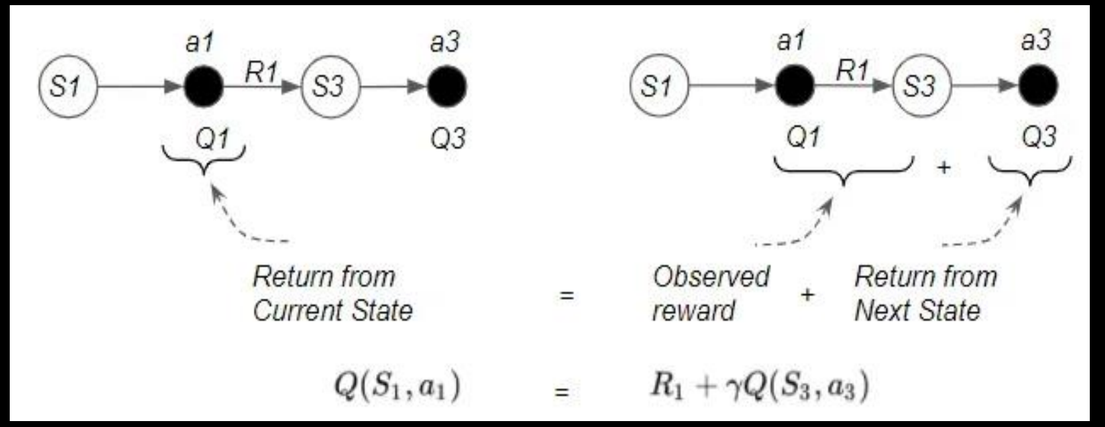
Step 3: Q value estimation

Now that we have picked our action, we need to build our intuition behind the formula.
Now we know that after picking the action, the reward we get (say r) plays a role in the Q value, for an optimum Q value, we can say that

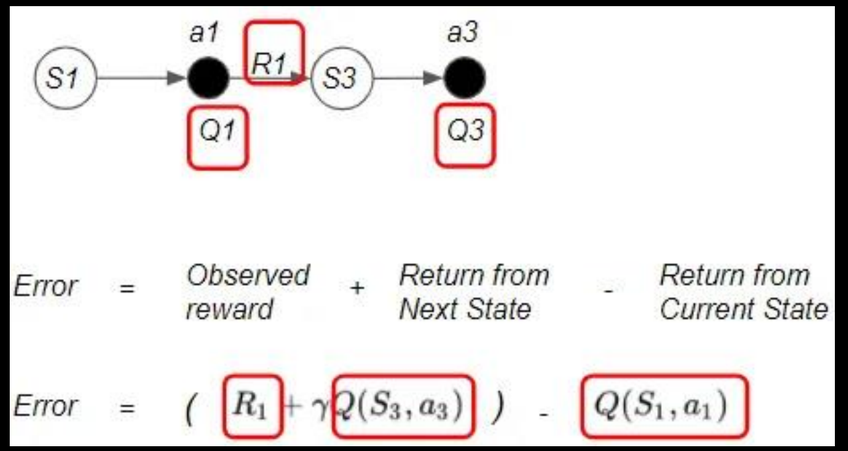
The Q value of a particular state-action is the reward after taking that action + the optimum return expected from the next state-actions.

Hence, we say that the Q value = Reward + gamma*maximum Q value for all actions in the next state.

Therefore we should be training our model such that the two values are close, in other words the ERROR is reduced.



Here, a3 is the action which has the highest Q value for all the actions from state S3



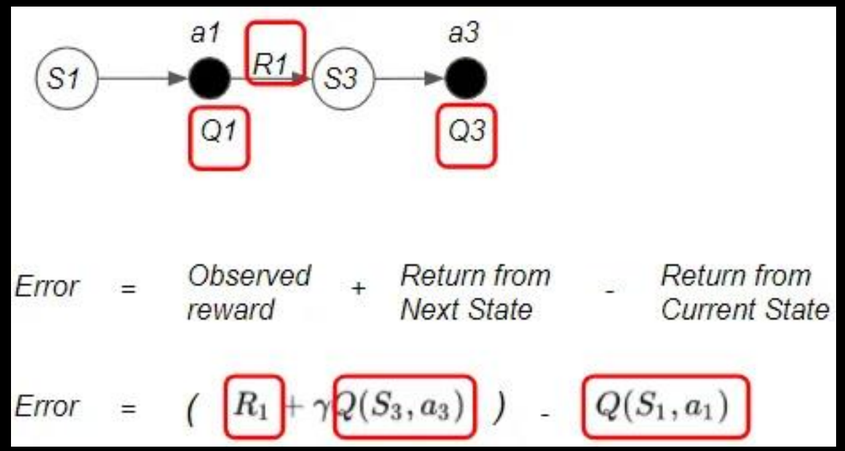
Step 3: Q value estimation

Therefore, we need to add the error term to our estimate such that the two ways of calculating the State-Action Value:

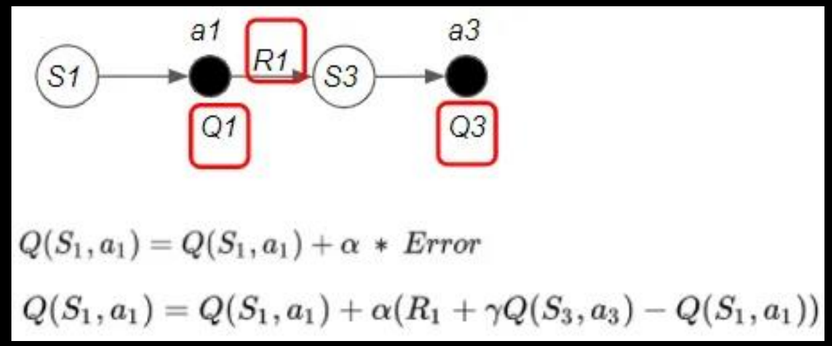
1.One way is the State-Action Value from the current state
2.The other way is the immediate reward from taking one step plus the State-Action Value from the next state

should provide very similar results.
Hence, we add the error term is added scaled with a learning rate.
The algorithm incrementally updates its Q-value estimates in a way that reduces the error.

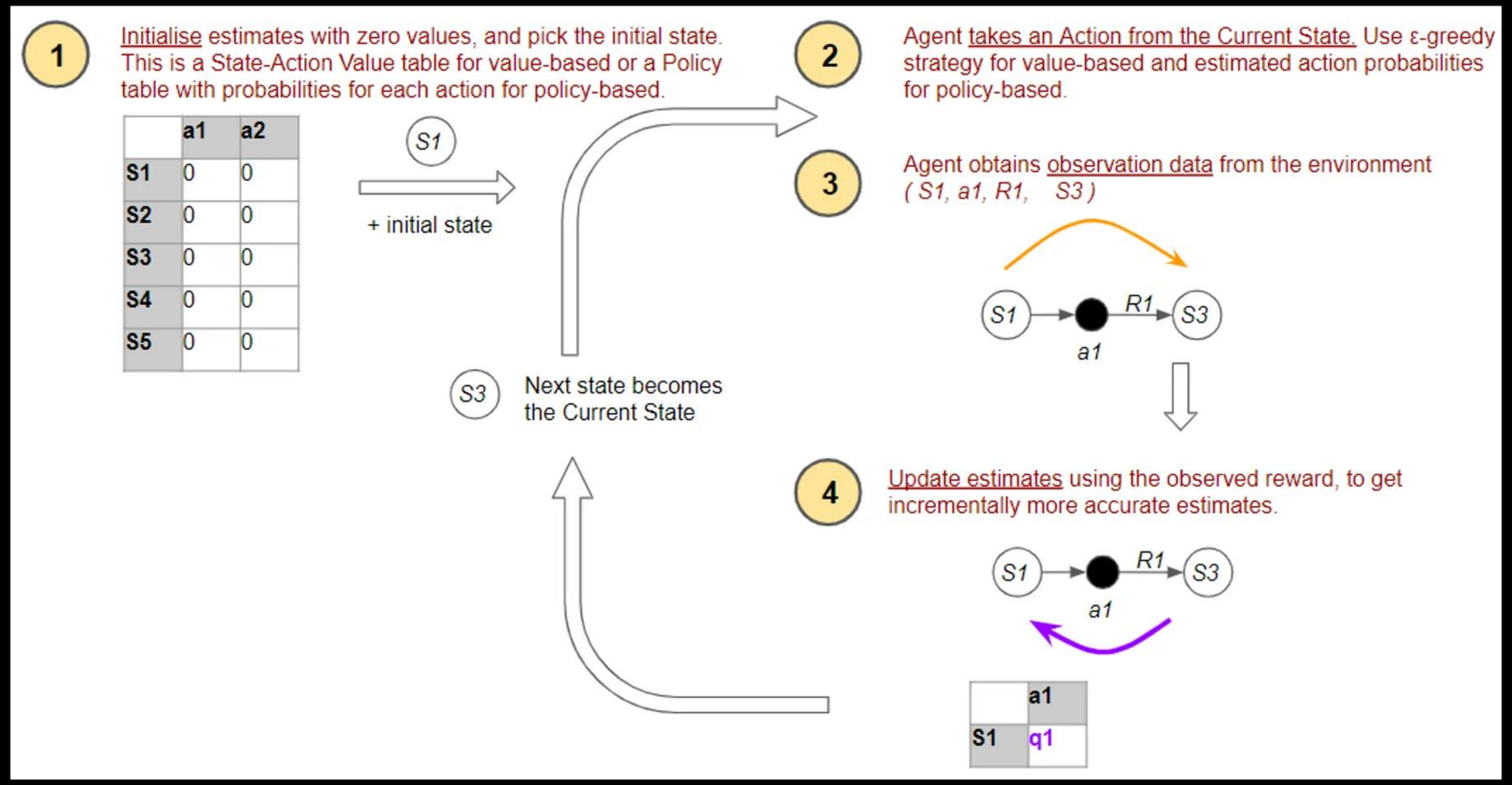
This process is then repeated for the next state after we take the particular action till CONVERGENCE.



Here, a3 is the action which has the highest Q value for all the actions from state S3



Step 3: Putting all together



Note that when we compare the two ways of computing the Q value for a particular state-action, we take the action in the next state which has maximum Q value, but it is not necessary that in the next iteration we actually follow that action, because we can also be exploring with a probability of ε. Hence, as it is not following exactly the same policy as it is using to compute the Q values, it is called an OFF-POLICY algorithm.

THANK YOU!