

## Reinforcement Learning

- Reinforcement learning involves an agent, a set of states  $S$  and a set of actions  $A$  per state. By performing an action  $a \in A$ , the agent transits from one state to another, where state  $s \in S$ . An agent will obtain a reward  $r \in R$  for each action when it is in a particular state.
- The goal of the agent is to maximize its total reward. Total reward can be calculated by adding the maximum potential reward to the reward for achieving its current state. This potential reward is a weighted sum of expected values of the rewards of all future states starting from the current state.

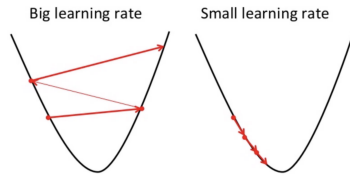
### Environment

There are mainly 2 types of environments that we are concerned with for a reinforcement learning problem:

1. **Deterministic environment:** For every state-action pair, immediate reward obtained by the agent due to this action remains constant.
2. **Stochastic environment:** For every state-action pair, immediate reward obtained by the agent due to this action is uncertain.

## Q-Learning

- Q-learning is a model-free reinforcement learning algorithm which can obtain the expected rewards for each action taken when the agent is at a given state.
- For any Finite Markov Decision Process (FMDP), Q-learning is able to identify the optimal action to take for at each state as  $t \rightarrow \infty$ .
- Q-value can be updated iteratively using the Bellman equation  $Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max(Q(s_{t+1}, a_{t+1})))$ .
  - $\leftarrow$ : symbol for assignment. For example, if  $x = 2$ ,  $x \leftarrow x + 1$  would make  $x = 3$ .
  - $t$ : current time.
  - $\alpha$ : learning rate,  $0 \leq \alpha \leq 1$ . The higher the learning rate, the larger the extent Q-values are updated. Learning rate should not be too big, as it may fluctuate about the minima (optimal value), and the minima for some state-action pairs may not be reached even after a long time. Learning rate should also not be too small, as it may take a long time until the Q-value converges to the minima. As such, a decay function is usually used for learning rate where it will decrease over time.



- If  $\alpha = 0$ , Q-values will never be updated.
- If  $\alpha = 1$ , updated Q-values do not consider previous experiences and only factor in the most recent experience.
- $\gamma$ : discount factor,  $0 \leq \gamma \leq 1$ . The higher the discount factor, the higher the importance that is placed on future rewards when calculating the Q-value at time  $t$ .  $\gamma < 1$  also guarantees convergence of the Bellman equation and prevents Q-values from reaching infinity.
  - If  $\gamma = 0$ , only the immediate rewards are considered in the calculation of Q-values for each state-action pair.
  - If  $\gamma = 1$ , future rewards are just as important as immediate rewards in the calculation of Q-values for each state-action pair.
- $r_t$ : reward earned for taking action  $a_t$  after observing state  $s_t$ .
- $Q(s_t, a_t)$ : Q-value for taking action  $a_t$  after observing state  $s_t$ .
- Since Q-value is updated iteratively, we would need to assume a value for the initial Q-value  $Q_0$  for any state-action pair that the agent has not encountered previously. We would generally set  $Q_0 = 0$ , but sometimes setting  $Q_0 > 0$  (also known as optimistic initial conditions) may be better in the long run as it encourages exploration.
- If  $(s_{final}, a_{final})$  is the last state-action pair for the episode, then  $Q(s_{final}, a_{final}) = 0$ , assuming initial Q-value  $Q_0 = 0$ . This is because no action is usually taken when the terminal state  $s_{final}$  is reached, and  $s_{final+1}$  would not exist. This leads to final reward obtained  $r_{final} = 0$ , and set of expected Q-values  $\{Q(s_{final+1}, a_{final+1}) | a_{t+1} \in A_{t+1}\} = \{0\}$ .
- Q-values are stored in a Q-table, where they are mapped to their respective state-action pairs.

○

	Action 1	Action 2
State 1	$Q(s_1, a_1)$	$Q(s_1, a_2)$

<b>State 2</b>	$Q(s_2, a_1)$	$Q(s_2, a_2)$
----------------	---------------	---------------

For the difference between Q-learning and SARSA, see this answer here: [What is the difference between Q-learning and SARSA?](#)

## Replay Memory

- During the training of the agent, experience replay is often used for deep Q-learning (DQN) to allow the agent to memorise and reuse past experiences. This is done by storing the agent's experiences at each time step in the replay memory (or buffer). For Q-learning, agent's experience at each time step  $e_t = (s_t, a_t, r_t, s_{t+1})$ . All of the agent's experiences at each time step over all episodes are stored in the replay memory.
  - We would generally set a limit to the number of experiences that can be stored in the replay memory, and when the replay memory is full, the older experiences will be replaced by newer experiences (acts like a queue, a FIFO data structure). This is to ensure efficient learning.
  - We would randomly retrieve a portion of the replay memory and use this as the sample to train the network. A random sample is used rather than providing the sequential experiences that occur in the environment. This is because Q-learning algorithms with function approximation such as DQN would often cause q-values to diverge rather than converge, and it is only through random sampling that we can break the correlation between sequential experiences.
- In the case of vanilla Q-learning without deep neural networks, a Q-table is used to store the various state-action pairs. While there is often a limit to the number of experiences in the replay memory, there is no such limit for the Q-table. However, since the Q-table would need to store the Q-values for all state-action pairs, vanilla Q-learning only works for discrete state-action pairs.
  - For each episode, an episodic memory is often used, where the agent's experience at each time step  $e_t = (s_t, a_t, r_t, s_{t+1})$  is stored. Episodic memory is reset after an episode is completed. At each time step, the agent's experience will be inserted to the episodic memory, and Q-values for each experience in this memory will be updated based on the Bellman

equation, starting from the most recent experience. The Q-table is then updated with these new Q-values.

## Epsilon-Greedy Algorithm

- **Exploration:** To learn what is the best action for each state, the agent has to explore the environment. This is often done by choosing a random action  $a_t$  at state  $s_t$ , where the agent can take an action that has not or rarely been experienced at that particular state to improve current knowledge.
- **Exploitation:** Prior knowledge learned from exploring can be exploited, where the agent would choose the action which maximises expected reward at that particular state.
- In reinforcement learning algorithms such as Q-learning, there is always the existence of exploration-exploitation trade-off.
  - A greedy algorithm that is only focused on exploiting would often get stuck because it would always choose the action with the highest Q-value, despite the possibilities of other actions that could have higher Q-value and yield higher future rewards which are not yet explored (since most Q-values in the Q-table have not converged to their optimal value).
  - A thrifty algorithm that is only focused on exploring is able to lead to convergence for all Q-values, but that can only be achieved in theory where  $t \rightarrow \infty$ . However, while the optimal values for Q-values have been achieved, the agent would still move randomly and would not deliberately take actions to maximise its expected rewards. Moreover, there is lower probability for the agent to reach convergence for Q-values for the set of state-action pairs that maximises total rewards if it were to use a thrifty algorithm, which leads to longer time taken for the agent to effectively solve the reinforcement learning problem.
  - As such,  $\epsilon$ -greedy algorithm is often used to balance exploration and exploitation, where  $\epsilon$  refers to the exploration rate and  $0 \leq \epsilon \leq 1$ . The agent has  $\epsilon$  probability of taking a random action, and  $1 - \epsilon$  probability of taking the action that maximises the expected reward obtained. As the agent gains more experience over time, it gains sufficient knowledge on what action to take for an increasing number of states. As such, a decay function is usually used for exploration rate where it will decrease over time.

## Categorical Encoding

- **Label encoding:** Each categorical variable can be represented as a unique integer label.

- Example:

Variable	Label
Land	1
Air	2
Sea	3

- **One-hot encoding:** Each categorical label can be converted into a column, and a binary value of 0 or 1 is assigned to the columns, where 0 refers to False and 1 refers to True in boolean expression.

- Example:

Variable	Land_Label	Air_Label	Sea_Label
Land	1	0	0
Air	0	1	0
Sea	0	0	1

- One-hot encoding is used more often than label encoding as label encoding may cause machine learning models to capture relationships that are untrue (e.g.  $\text{land} < \text{air} < \text{sea}$ ).

## Q-learning for Mazes

We can utilise the Q-learning algorithm to train the agent to navigate a maze. For the mazes I have generated, the existence of unknown tiles which require the agent to explore to know whether it is a wall or it is empty means that many maze-solving algorithms are not applicable. Whenever the agent moves from one tile to another, it will search up, down, left and right, and a new tile will appear on the screen until a wall is found.

### Maze

For my code, I generated the maze using recursive division. The start and end of the maze would always be at the top left and bottom right of the maze respectively.




Explanation and visualisation on how a recursive division maze is generated can be found here: [Recursive Division Maze Generating](#)

### State-action pairs

For each episode, I would generate an entirely different maze for the agent to explore. This is so that the agent would gain enough experience to traverse and win every possible maze. My initial idea for the state-action pairs was as such:

- Set of states  $S = \{x \in X \text{ for each coordinate}\}$ 
  - Set of tiles  $X = \{EMPTY, WALL, PLAYER, END, UNKNOWN\}$ 
    - $EMPTY = 0$  (WHITE)
    - $WALL = 1$  (BLACK)
    - $PLAYER = 2$  (BLUE)
    - $END = 3$  (RED)
    - $UNKNOWN = 4$  (GREY)
- Set of actions  $A = \{LEFT, RIGHT, UP, DOWN\}$ 
  - $LEFT = 0$
  - $RIGHT = 1$
  - $UP = 2$
  - $DOWN = 3$

To provide an example of how the agent would act, assuming greedy policy is adopted:


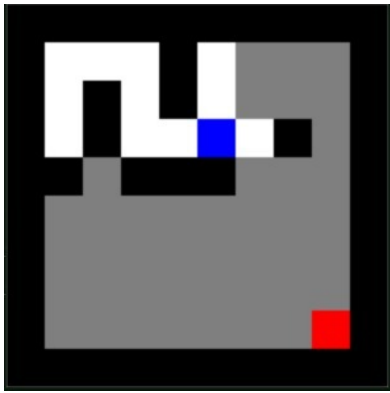

State at time step $t$ $s_t$	Action $a_t$	State at time step $t + 1$ $s_{t+1}$										
	<p><b>RIGHT</b> button is pressed, as it has maximum <math>Q(s_t, a_t)</math>. Reward <math>r_t</math> is given to the agent.</p>											
For $s_t$ :		For $s_t$ :										
<table> <tr> <th><math>a_t</math></th> <th><math>Q(s_t, a_t)</math></th> </tr> <tr> <td>LEFT</td> <td>0.4</td> </tr> <tr> <td>RIGHT</td> <td>1.2</td> </tr> <tr> <td>UP</td> <td>0.1</td> </tr> <tr> <td>DOWN</td> <td>0.2</td> </tr> </table>	$a_t$	$Q(s_t, a_t)$	LEFT	0.4	RIGHT	1.2	UP	0.1	DOWN	0.2		<ul style="list-style-type: none"> <li>Update Q-value in Q-table for <math>a_t = \text{DOWN}</math> for <math>s_t</math> via Bellman equation.</li> <li>Update Q-values in Q-table for previous state-action pairs before time step <math>t</math> via Bellman equation.</li> </ul>
$a_t$	$Q(s_t, a_t)$											
LEFT	0.4											
RIGHT	1.2											
UP	0.1											
DOWN	0.2											

However, by considering each tile for each coordinate in the maze as the state, such a method is computationally expensive. For example, for a 8 by 8 maze, there are approximately  $4^{64}$  state-action pairs, and as there is a very low probability of reaching each state-action pair, it would also take very long for the Q-values to converge and reach optimal values.

As such, I decided to use a new set of states. Each state  $s$  in the second set of states  $S$  would include:

- $x$  refers to the x-coordinate of the agent.
- $y$  refers to the y-coordinate of the agent.
- *LEFT* refers to the number of empty tiles to the left of the agent.
- *RIGHT* refers to the number of empty tiles to the right of the agent.
- *UP* refers to the number of empty tiles above the agent.
- *DOWN* refers to the number of empty tiles below the agent.

To provide an example of how the agent would act for the second set of states, assuming greedy policy is adopted:

State at time step $t$ $s_t$	Action $a_t$	State at time step $t + 1$ $s_{t+1}$																								
 <table><tr><td><b>x</b></td><td>4</td></tr><tr><td><b>y</b></td><td>3</td></tr><tr><td><b>LEFT</b></td><td>1</td></tr><tr><td><b>RIGHT</b></td><td>2</td></tr><tr><td><b>UP</b></td><td>0</td></tr><tr><td><b>DOWN</b></td><td>0</td></tr></table>	<b>x</b>	4	<b>y</b>	3	<b>LEFT</b>	1	<b>RIGHT</b>	2	<b>UP</b>	0	<b>DOWN</b>	0	<p><b>RIGHT</b> button is pressed, as it has maximum <math>Q(s_t, a_t)</math>. Reward <math>r_t</math> is given to the agent.</p>	 <table><tr><td><b>x</b></td><td>5</td></tr><tr><td><b>y</b></td><td>3</td></tr><tr><td><b>LEFT</b></td><td>2</td></tr><tr><td><b>RIGHT</b></td><td>1</td></tr><tr><td><b>UP</b></td><td>2</td></tr><tr><td><b>DOWN</b></td><td>0</td></tr></table>	<b>x</b>	5	<b>y</b>	3	<b>LEFT</b>	2	<b>RIGHT</b>	1	<b>UP</b>	2	<b>DOWN</b>	0
<b>x</b>	4																									
<b>y</b>	3																									
<b>LEFT</b>	1																									
<b>RIGHT</b>	2																									
<b>UP</b>	0																									
<b>DOWN</b>	0																									
<b>x</b>	5																									
<b>y</b>	3																									
<b>LEFT</b>	2																									
<b>RIGHT</b>	1																									
<b>UP</b>	2																									
<b>DOWN</b>	0																									
For $s_t$ : <table><tr><td><math>a_t</math></td><td><math>Q(s_t, a_t)</math></td></tr></table>	$a_t$	$Q(s_t, a_t)$		For $s_t$ : <ul style="list-style-type: none"><li>• Update Q-value in Q-table for</li></ul>																						
$a_t$	$Q(s_t, a_t)$																									



LEFT	0.4		$a_t = DOWN$ for $s_t$ via Bellman equation. • Update Q-values in Q-table for previous state-action pairs before time step $t$ via Bellman equation.
RIGHT	1.2		
UP	0.1		
DOWN	0.2		

I did consider distance from the exit to be a state on its own, but rejected that idea afterwards as there are some cases where closest distance to exit is actually the worst route. For example, the agent at coordinate (6, 8) may be 2 blocks away from the exit (8, 8), but the route that it is going leads to a dead end as there is a wall at (7, 8) that blocks its path.

### Rewards

```
REWARDS = {"NEW": 1, "OLD": -1, "WALL": -10, "WIN": 1000}
```

- *NEW*: New tile discovered. (+1)
- *OLD*: No new tile discovered. This means the agent is walking backwards. (-1)
- *WALL*: Hitting the wall wastes an action. (-10)
- *WIN*: Reach the end. (+1000)
  - As having a static win reward does not penalise the agent for taking a longer route, reward for win
$$r_t = 1000 \cdot (2 - \text{number\_of\_actions} \div \text{MAX\_ACTIONS}).$$

### Hyperparameters

```
INITIAL_Q_VALUE = 0.0
MAX_ACTIONS = WIDTH * HEIGHT

DISCOUNT_FACTOR = 0.95
LEARNING_RATE = 0.05

EXPLORATION_MIN = 0.10
EXPLORATION_MAX = 0.95
EXPLORATION_DECAY = 0.995
```

Note that for state-action pairs not in Q-table, it is implicit that the Q-value for that state-action pair is equal to the initial Q-value which I have set to 0.

You are free to edit these values to optimise the training of the agent. The values that I have set for the hyperparameters are merely arbitrary values which have not been tuned as of now.

I have set the maximum number of actions that can be taken for each episode to be  $MAX\_ACTIONS = WIDTH \cdot HEIGHT$  such that the agent would not get stuck in an infinite loop, while also ensuring that the agent would always be able to reach the end of the maze within this maximum number of actions.

## Results

- First set of states  $S$ 
  - **Random mazes, vanilla Q-learning:** 6 wins out of 240 episodes, 1st win on 39th episode.
- Second set of states  $S$ 
  - **Same maze, vanilla Q-learning:** 48 wins out of 100 episodes, 1st win on 25th episode, 1 loss from 76th to 100th episode.
  - **Random mazes, vanilla Q-learning:** 49 wins out of 500 episodes, 1st win on 30th episode, 12 wins from 401st to 500th episode.
  - **Random mazes, Q-learning + Random Forest:** 7 wins out of 100 episodes, 1st win on 30th episode.
    - Takes way longer time than vanilla Q-learning as all memory from all episodes need to be fit to the model, and the model is refitted at every time step.