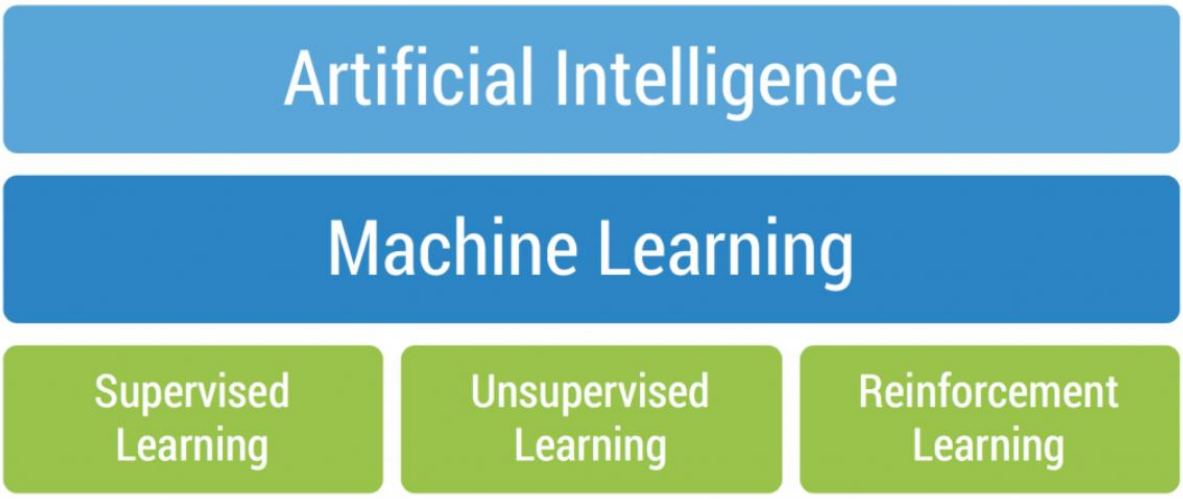
Q-LEARNING

BY:

M. ASHISH REDDY

DSWP -> BATCH – 5



Reinforcement Learning (RL) can be defined as the study of taking optimal decisions utilizing experiences. It is mainly intended to solve a specific kind of problem where the decision making is successive and the goal or objective is long-term, this includes robotics, game playing, or even logistics and resource management.

In simple words, unlike the other machine learning algorithms, the ultimate goal of RL is not to be greedy all the time by looking for quick immediate rewards, rather optimize for maximum rewards over the complete training process.

MUST-KNOW TERMINOLOGIES:

Agent: The agent in RL can be defined as the entity which acts as a learner and decision-maker. It is empowered to interact continually, select its own actions, and respond to those actions.

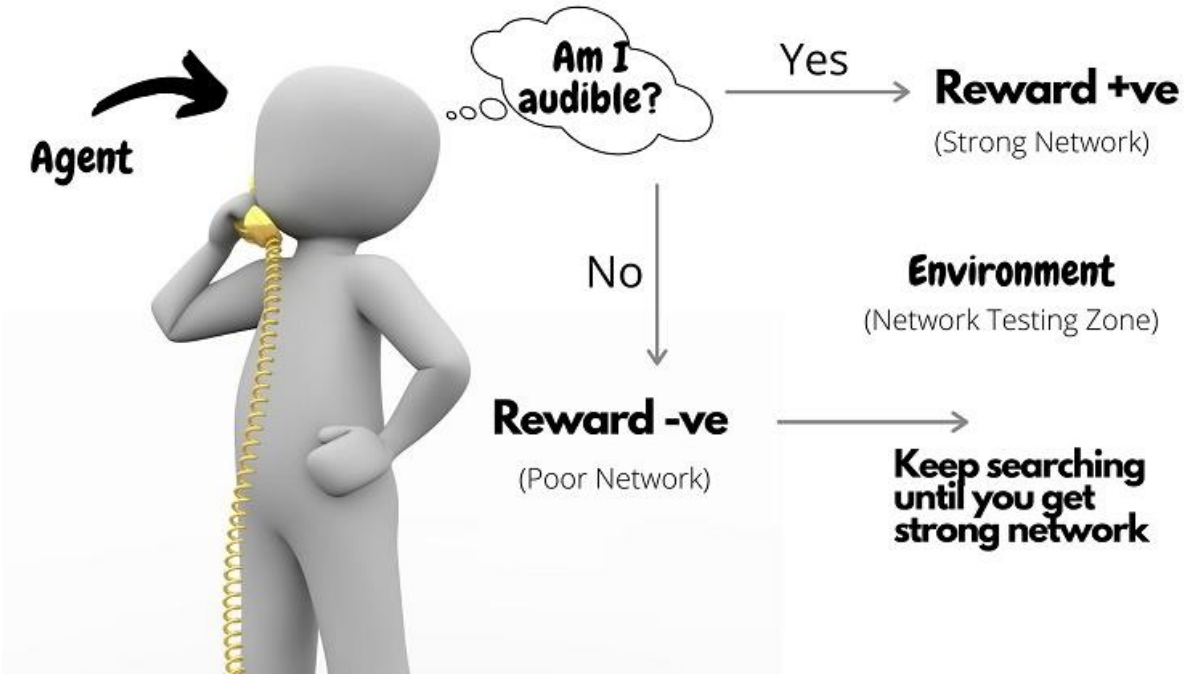
Environment: It is the abstract world through which the agent moves. The Environment takes the current state and action of the agent as input and returns its next state and appropriate reward as the output.

States: The specific place at which an agent is present is called a state. This can either be the current situation of the agent in the environment or any of the future situations.

Actions: This defines the set of all possible moves an agent can make within an environment.

Reward or Penalty: This is nothing but the feedback by means of which the success or failure of an agent's action within a given state can be measured. The rewards are used for the effective evaluation of an agent's action.

Policy or Strategy: It is mainly used to map the states along with their actions. The agent is said to use a strategy to determine the next best action based on its current state.



The Q – Learning Algorithm in RL!

Of all the types of algorithms available in reinforcement learning, ever wondered why this Q-learning has always been the sought-after one?

Q-learning is a model-free, value-based, off-policy learning algorithm.

- **Model-free:** The algorithm that estimates its optimal policy without the need for any transition or reward functions from the environment.
- **Value-based:** Q learning updates its value functions based on equations, (say Bellman equation) rather than estimating the value function with a greedy policy.
- **Off-policy:** The function learns from its own actions and doesn't depend on the current policy.

Having known the basics, let's now jump right away into the implementation part using simple python code. I suppose y'all would have gone through plenty of examples using the OpenAI baselines for illustrating different reinforcement learning algorithms.

Yet, this example can give you a clear-cut explanation about the exact working of the Q-learning algorithm in an effortless way!

Scenario – Robots in a Warehouse

A growing e-commerce company is building a new warehouse, and the company would like all of the picking operations in the new warehouse to be performed by warehouse robots.

- In the context of e-commerce warehousing, “picking” is the task of gathering individual items from various locations in the warehouse in order to fulfill customer orders.

After picking items from the shelves, the robots must bring the items to a specific location within the warehouse where the items can be packaged for shipping.

In order to ensure maximum efficiency and productivity, the robots will need to learn the shortest path between the item packaging area and all other locations within the warehouse where the robots are allowed to travel.

- We will use Q-learning to accomplish this task!

Import Required Libraries

```
#import libraries
import numpy as np
```

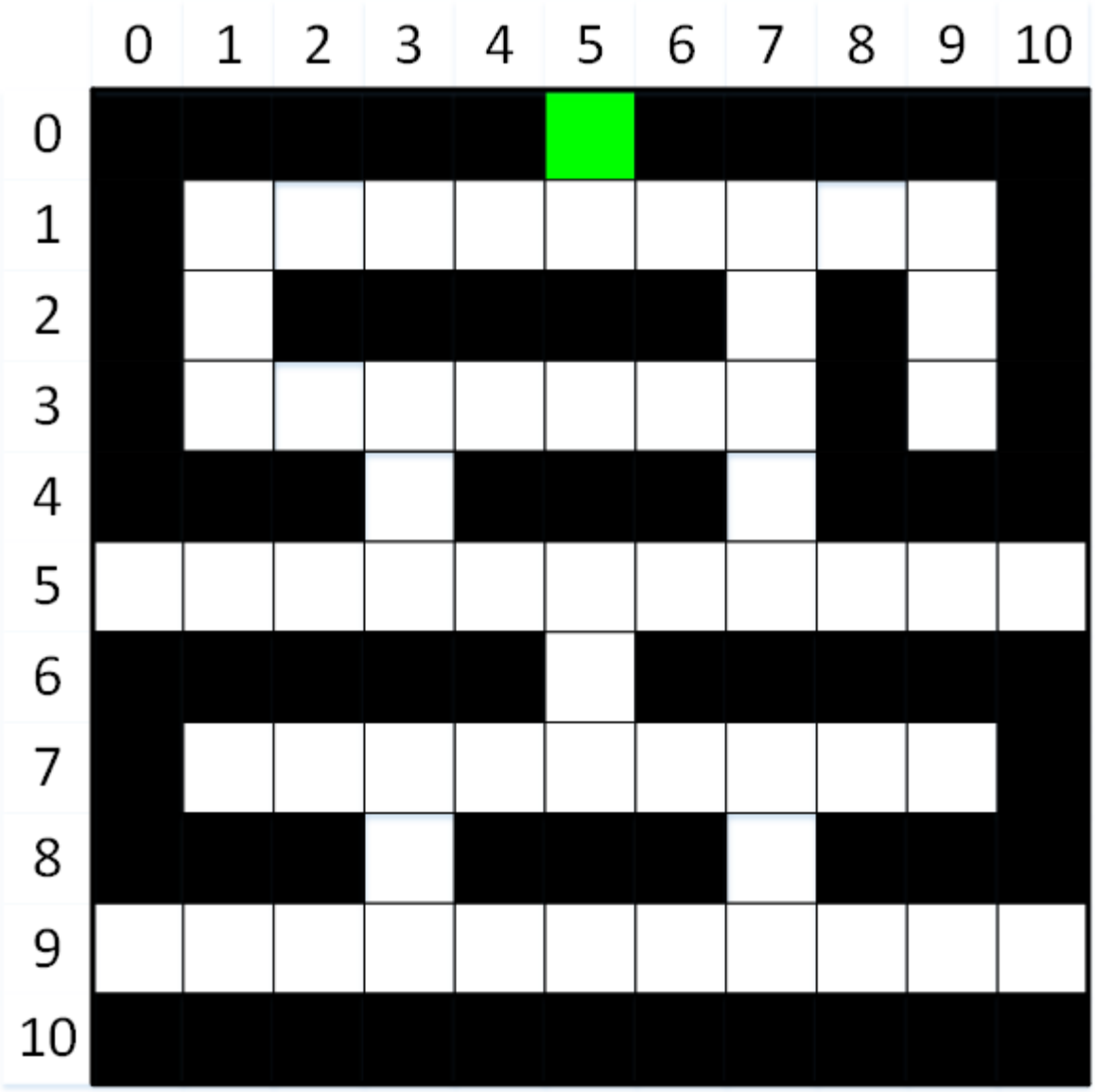
Define the Environment

The environment consists of **states**, **actions**, and **rewards**. States and actions are inputs for the Q-learning agent, while the possible actions are the agent’s outputs.

States

The states in the environment are all of the possible locations within the warehouse. Some of these locations are for storing items (**black squares**), while other locations are aisles that the robot can use to travel throughout the warehouse (**white squares**). The **green square** indicates the item packaging and shipping area.

The black and green squares are **terminal states**!



The agent's goal is to learn the shortest path between the item packaging area and all of the other locations in the warehouse where the robot is allowed to travel.

As shown in the image above, there are 121 possible states (locations) within the warehouse. These states are arranged in a grid containing 11 rows and 11 columns. Each location can hence be identified by its row and column index.

```
#define the shape of the environment (i.e., its states)

environment_rows = 11

environment_columns = 11

#Create a 3D numpy array to hold the current Q-values for each state and action pair: Q(s, a)

#The array contains 11 rows and 11 columns (to match the shape of the environment), as well as a third "action" dimension.

#The "action" dimension consists of 4 layers that will allow us to keep track of the Q-values for each possible action in

#each state (see next cell for a description of possible actions).

#The value of each (state, action) pair is initialized to 0.

q_values = np.zeros((environment_rows, environment_columns, 4))
```

Actions

The actions that are available to the agent are to move the robot in one of four directions:

- Up
- Right
- Down
- Left

Obviously, the agent must learn to avoid driving into the item storage locations (e.g., shelves)!

```
#define actions

#numeric action codes: 0 = up, 1 = right, 2 = down, 3 = left

actions = ['up', 'right', 'down', 'left']
```

Rewards

The last component of the environment that we need to define is the **rewards**. To help the agent learn, each state (location) in the warehouse is assigned a reward value. The agent may begin at any white square, but its goal is always the same: ***to maximize its total rewards***! Negative rewards (i.e., **punishments**) are used for all states except the goal.

- This encourages the agent to identify the *shortest path* to the goal by *minimizing its punishments*!

	0	1	2	3	4	5	6	7	8	9	10
0	-100	-100	-100	-100	-100	100	-100	-100	-100	-100	-100
1	-100	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100
2	-100	-1	-100	-100	-100	-100	-100	-1	-100	-1	-100
3	-100	-1	-1	-1	-1	-1	-1	-1	-100	-1	-100
4	-100	-100	-100	-1	-100	-100	-100	-1	-100	-100	-100
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	-100	-100	-100	-100	-100	-1	-100	-100	-100	-100	-100
7	-100	-1	-1	-1	-1	-1	-1	-1	-1	-1	-100
8	-100	-100	-100	-1	-100	-100	-100	-1	-100	-100	-100
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100	-100

To maximize its cumulative rewards (by minimizing its cumulative punishments), the agent will need to find the shortest paths between the item packaging area (green square) and all of the other locations in the warehouse where the robot is allowed to travel (white squares). The agent will also need to learn to avoid crashing into any of the item storage locations (black squares)!

```
#Create a 2D numpy array to hold the rewards for each state.
#The array contains 11 rows and 11 columns (to match the shape of the environment), and each value is initialized to -100.
rewards = np.full((environment_rows, environment_columns), -100.)
rewards[0, 5] = 100. #set the reward for the packaging area (i.e., the goal) to 100
#define aisle locations (i.e., white squares) for rows 1 through 9
aisles = {} #store locations in a dictionary
aisles[1] = [i for i in range(1, 10)]
aisles[2] = [1, 7, 9]
aisles[3] = [i for i in range(1, 8)]
aisles[3].append(9)
aisles[4] = [3, 7]
aisles[5] = [i for i in range(11)]
aisles[6] = [5]
aisles[7] = [i for i in range(1, 10)]
aisles[8] = [3, 7]
aisles[9] = [i for i in range(11)]
#set the rewards for all aisle locations (i.e., white squares)
for row_index in range(1, 10):
  for column_index in aisles[row_index]:
    rewards[row_index, column_index] = -1.
#print rewards matrix
for row in rewards:
  print(row)
```

Output:

```
[-100. -100. -100. -100. -100. 100. -100. -100. -100. -100. -100.]
[-100. -1. -1. -1. -1. -1. -1. -1. -1. -1. -100.]
```

```
[-100. -1. -100. -100. -100. -100. -100. -1. -100. -1. -100.]
[-100. -1. -1. -1. -1. -1. -1. -1. -100. -1. -100.]
[-100. -100. -100. -1. -100. -100. -100. -1. -100. -100. -100.]
[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
[-100. -100. -100. -100. -100. -1. -100. -100. -100. -100. -100.]
[-100. -1. -1. -1. -1. -1. -1. -1. -1. -1. -100.]
[-100. -100. -100. -1. -100. -100. -100. -1. -100. -100. -100.]
[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
[-100. -100. -100. -100. -100. -100. -100. -100. -100. -100. -100.]
```

Train the Model

Our next task is for our agent to learn about its environment by implementing a Q-learning model. The learning process will follow these steps:

1. Choose a random, non-terminal state (white square) for the agent to begin this new episode.
2. Choose an action (move *up*, *right*, *down*, or *left*) for the current state. Actions will be chosen using an *epsilon greedy algorithm*. This algorithm will usually choose the most promising action for the agent, but it will occasionally choose a less promising option in order to encourage the agent to explore the environment.
3. Perform the chosen action, and transition to the next state (i.e., move to the next location).
4. Receive the reward for moving to the new state, and calculate the temporal difference.
5. Update the Q-value for the previous state and action pair.
6. If the new (current) state is a terminal state, go to #1. Else, go to #2.

This entire process will be repeated across 1000 episodes. This will provide the agent sufficient opportunity to learn the shortest paths between the item packaging area and all other locations in the warehouse where the robot is allowed to travel, while simultaneously avoiding crashing into any of the item storage locations!

Define Helper Functions

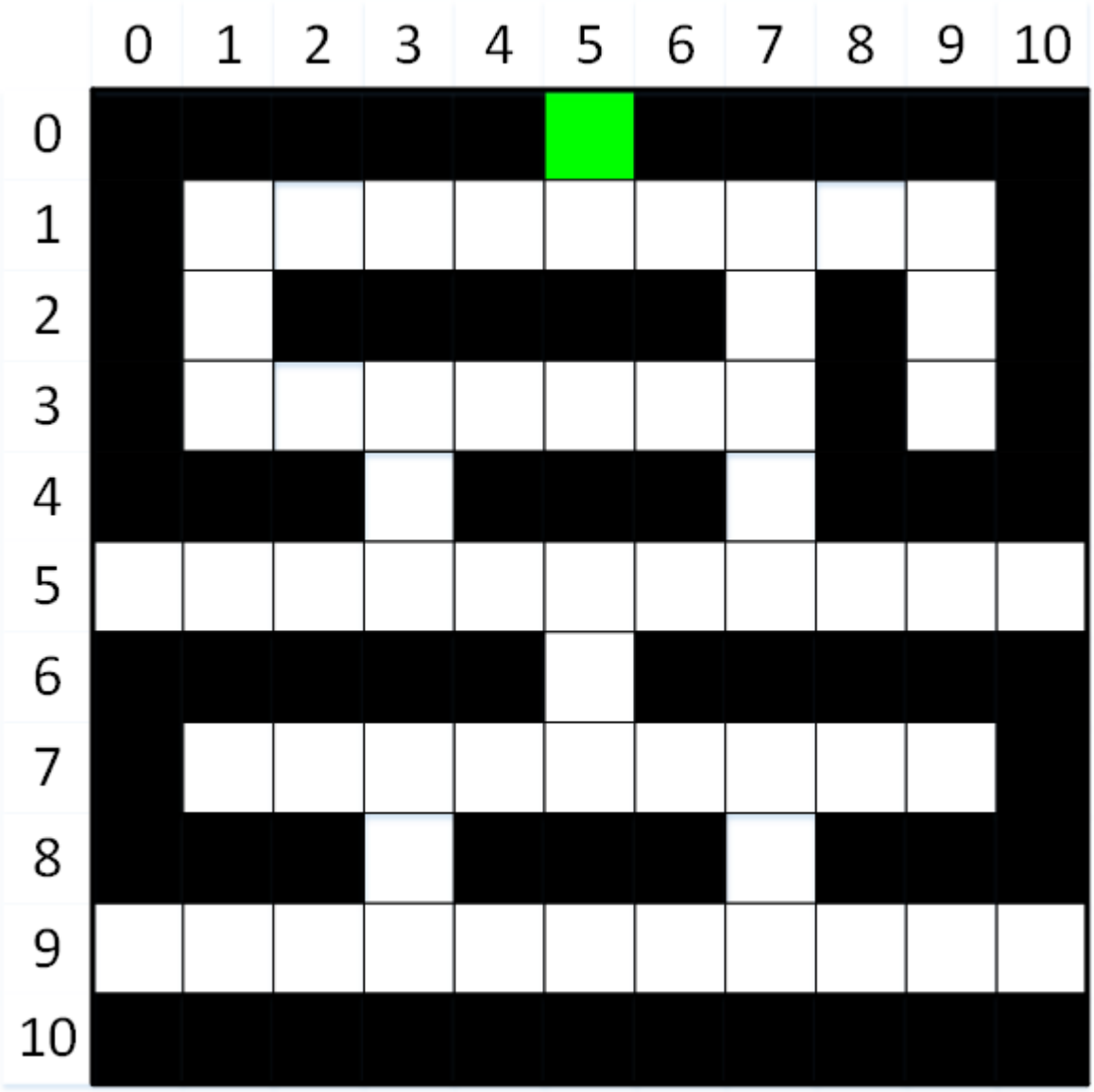
Train the Agent using Q-Learning Algorithm

Output:

Training complete!

Get Shortest Paths

Now that the agent has been fully trained, we can see what it has learned by displaying the shortest path between any location in the warehouse where the robot is allowed to travel and the item packaging area.



Try out a few different starting locations!

```
#display a few shortest paths
print(get_shortest_path(3, 9)) #starting at row 3, column 9
print(get_shortest_path(5, 0)) #starting at row 5, column 0
print(get_shortest_path(9, 5)) #starting at row 9, column 5
```

Output:

[[3, 9], [2, 9], [1, 9], [1, 8], [1, 7], [1, 6], [1, 5], [0, 5]]

[[5, 0], [5, 1], [5, 2], [5, 3], [4, 3], [3, 3], [3, 4], [3, 5], [3, 6], [3, 7], [2, 7], [1, 7], [1, 6], [1, 5], [0, 5]]

[[9, 5], [9, 6], [9, 7], [8, 7], [7, 7], [7, 6], [7, 5], [6, 5], [5, 5], [5, 6], [5, 7], [4, 7], [3, 7], [2, 7], [1, 7], [1, 6], [1, 5], [0, 5]]

Finally…

It's great that our robot can automatically take the shortest path from any 'legal' location in the warehouse to the item packaging area. **But what about the opposite scenario?**

Put differently, our robot can currently deliver an item from anywhere in the warehouse ***to*** the packaging area, but after it delivers the item, it will need to travel ***from*** the packaging area to another location in the warehouse to pick up the next item!

Don't worry — this problem is easily solved simply by ***reversing the order of the shortest path!***

```
#display an example of reversed shortest path
path = get_shortest_path(5, 2) #go to row 5, column 2
path.reverse()
print(path)
```

Output:

[[0, 5], [1, 5], [1, 6], [1, 7], [2, 7], [3, 7], [3, 6], [3, 5], [3, 4], [3, 3], [4, 3], [5, 3], [5, 2]]

What are the advantages of Q-Learning?

One of the strengths of Q-Learning is that **it is able to compare the expected utility of the available actions without requiring a model of the environment**. Reinforcement Learning is an approach where the agent needs no teacher to learn how to solve a problem.