Q Learning – Reinforcement Learning

Introduction

One of my favorite algorithms that I learned while taking a reinforcement learning course was q-learning. Probably because it was the easiest for me to understand and code, but also because it seemed to make sense. In this quick post I'll discuss q-learning and provide the basic background to understanding the algorithm.

What is q-learning?

Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

What's 'Q'?

The 'q' in q-learning stands for quality. Quality in this case represents how useful a given action is in gaining some future reward.

Create a q-table

When q-learning is performed we create what's called a *q-table* or matrix that follows the shape of [state, action] and we initialize our values to zero. We then update and store our *q-values* after an episode. This q-table becomes a reference table for our agent to select the best action based on the q-value.

```
import numpy as np# Initialize q-table values to 0Q = np.zeros((state_size, action_size))
```

Q-learning and making updates

The next step is simply for the agent to interact with the environment and make updates to the state action pairs in our q-table

`Q[state, action]`.

Taking Action: Explore or Exploit

An agent interacts with the environment in 1 of 2 ways. The first is to use the q-table as a reference and view all possible actions for a given state. The agent then selects the action based on the max value of those actions. This is known as ***exploiting*** since we use the information we have available to us to make a decision.

The second way to take action is to act randomly. This is called ***exploring***. Instead of selecting actions based on the max future reward we select an action at random. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process. You can balance exploration/exploitation using epsilon (ε) and setting the value of how often you want to explore vs exploit. Here's some rough code that will depend on how the state and action space are setup.

```
import random# Set the percent you want to explore
epsilon = 0.2if random.uniform(0, 1) < epsilon:
    """
    Explore: select a random action    """
else:
    """
    Exploit: select the action with max value (future reward)    """
```

Updating the q-table

The updates occur after each step or action and ends when an episode is done. Done in this case means reaching some terminal point by the agent. A terminal state for example can be anything like landing on a checkout page, reaching the end of some game, completing some desired objective, etc. The agent will not learn much after a single episode, but eventually with enough exploring (steps and episodes) it will converge and learn the optimal q-values or q-star ($Q*$).

Here are the 3 basic steps:

1. Agent starts in a state (s1) takes an action (a1) and receives a reward (r1)
2. Agent selects action by referencing Q-table with highest value (max) **OR** by random (epsilon, ε)
3. Update q-values

Here is the basic update rule for q-learning:

```
# Update q valuesQ[state, action] = Q[state, action] + lr * (reward + gamma *
np.max(Q[new_state, :]) — Q[state, action])
```

In the update above there are a couple variables that we haven't mentioned yet. Whats happening here is we adjust our q-values based on the difference between the discounted new values and the old values. We

discount the new values using gamma and we adjust our step size using learning rate (lr). Below are some references.

Learning Rate: lr or learning rate, often referred to as *alpha* or α, can simply be defined as how much you accept the new value vs the old value. Above we are taking the difference between new and old and then multiplying that value by the learning rate. This value then gets added to our previous q-value which essentially moves it in the direction of our latest update.

Gamma: gamma or γ is a discount factor. It's used to balance immediate and future reward. From our update rule above you can see that we apply the discount to the future reward. Typically this value can range anywhere from 0.8 to 0.99.

Reward: reward is the value received after completing a certain action at a given state. A reward can happen at any given time step or only at the terminal time step.

Max: np.max() uses the numpy library and is taking the maximum of the future reward and applying it to the reward for the current state. What this does is impact the current action by the possible future reward. This is the beauty of q-learning. We're allocating future reward to current actions to help the agent select the highest return action at any given state.

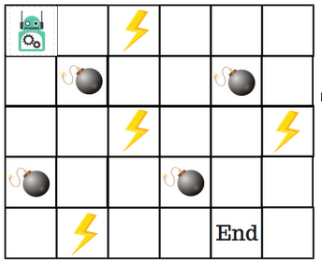
Q-Learning — a simplistic overview

Let's say that a **robot** has to cross a **maze** and reach the end point. There are **mines**, and the robot can only move one tile at a time. If the robot steps onto a mine, the robot is dead. The robot has to reach the end point in the shortest time possible.

The scoring/reward system is as below:

1. The robot loses 1 point at each step. This is done so that the robot takes the shortest path and reaches the goal as fast as possible.
2. If the robot steps on a mine, the point loss is 100 and the game ends.
3. If the robot gets power ⚡, it gains 1 point.
4. If the robot reaches the end goal, the robot gets 100 points.

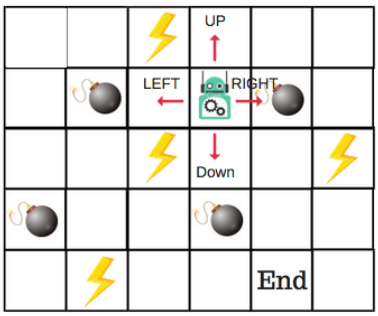
Now, the obvious question is: **How do we train a robot to reach the end goal with the shortest path without stepping on a mine?**



So, how do we solve this?

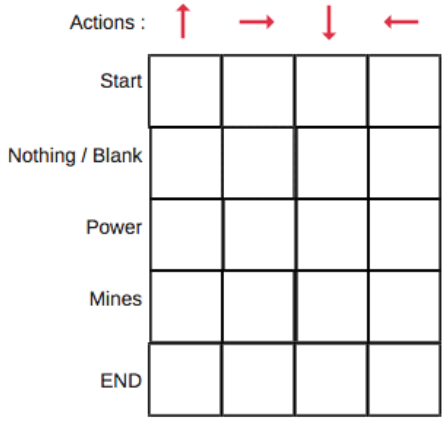
Introducing the Q-Table

Q-Table is just a fancy name for a simple lookup table where we calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state.



There will be four numbers of actions at each non-edge tile. When a robot is at a state it can either move up or down or right or left.

So, let's model this environment in our Q-Table. In the Q-Table, the columns are the actions and the rows are the states.



Each Q-table score will be the maximum expected future reward that the robot will get if it takes that action at that state. This is an iterative process, as we need to improve the Q-Table at each iteration.

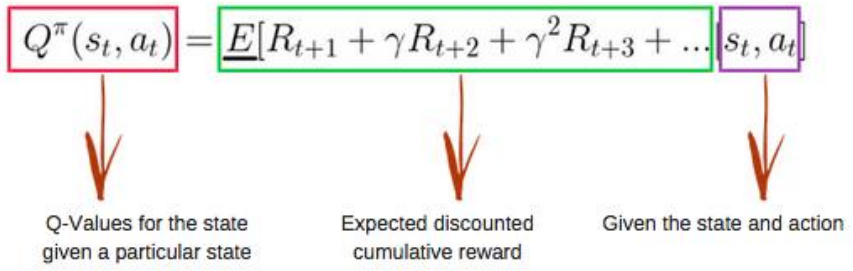
But the questions are:

- How do we calculate the values of the Q-table?
- Are the values available or predefined?

To learn each value of the Q-table, we use the **Q-Learning algorithm.**

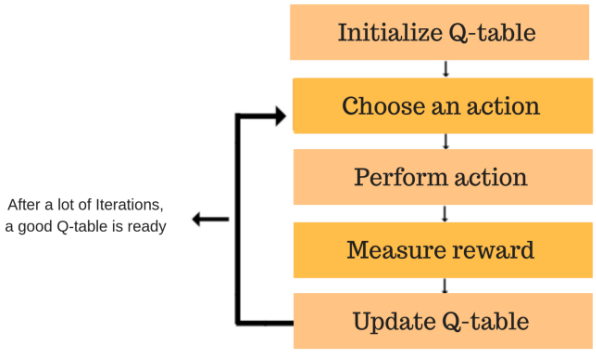
Mathematics: the Q-Learning algorithm

Q-function: The **Q-function** uses the Bellman equation and takes two inputs: state (**s**) and action (**a**).



Using the above function, we get the values of **Q** for the cells in the table. When we start, all the values in the Q-table are zeros. There is an iterative process of updating the values. As we start to explore the environment**,** the Q-function gives us better and better approximations by continuously updating the Q-values in the table.

Introducing the Q-learning algorithm process

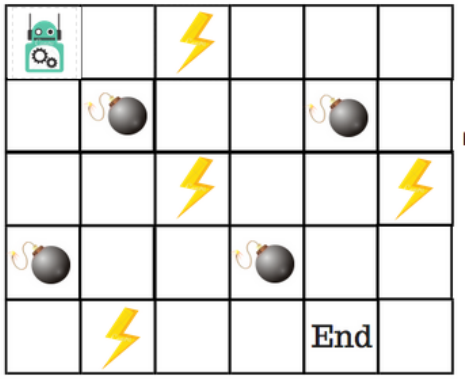


Each of the colored boxes is one step. Let's understand each of these steps in detail.

Step 1: initialize the Q-Table

We will first build a Q-table. There are n columns, where n= number of actions. There are m rows, where m= number of states. We will initialise the values at 0.

Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

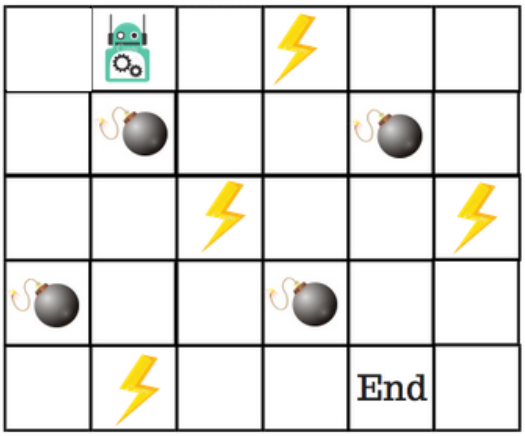


In our robot example, we have four actions (a=4) and five states (s=5). So we will build a table with four columns and five rows.

Steps 2 and 3: choose and perform an action

This combination of steps is done for an undefined amount of time. This means that this step runs until the time we stop the training, or the training loop stops as defined in the code. We will choose an action (a) in the state (s) based on the Q-Table. But, as mentioned earlier, when the episode initially starts, every Q-value is 0. So now the concept of exploration and exploitation trade-off comes into play. We'll use something called the **epsilon greedy strategy**. In the beginning, the epsilon rates will be higher. The robot will explore the environment and randomly choose actions. The logic behind this is that the robot does not know anything about the environment. As the robot explores the environment, the epsilon rate decreases and the robot starts to exploit the environment. During the process of exploration, the robot progressively becomes more confident in estimating the Q-values.

For the robot example, there are four actions to choose from: up, down, left, and right. We are starting the training now — our robot knows nothing about the environment. So the robot chooses a random action, say right.



Actions :	↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

We can now update the Q-values for being at the start and moving right using the Bellman equation.

Steps 4 and 5: evaluate

Now we have taken an action and observed an outcome and reward. We need to update the function Q(s,a).

$$\text{New } Q(s,a) = Q(s,a) + \alpha\,[R(s,a) + \gamma\, \text{maxQ}'(s',a') - Q(s,a)]$$

- New Q Value for that state and the action
- Learning Rate
- Reward for taking that action at that state
- Current Q Values
- Maximum expected future reward given the new state (s') and all possible actions at that new state.
- Discount Rate

In the case of the robot game, to reiterate the scoring/reward structure is:

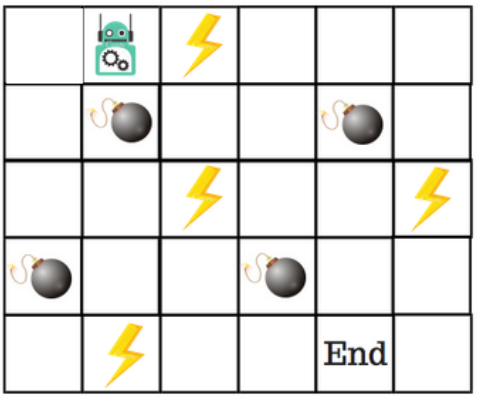
- **power** = +1
- **mine** = -100
- **end** = +100

New Q(start,right) = Q(start,right) + α[some ... Delta value]

Some ... Delta value = R(start,right) + max(Q`(nothing,down),Q`(nothing,left),Q`(nothing,right)) - Q(start,right)

Some ... Delta value = 0 + 0.9 * 0 - 0 = 0

New Q(start,right) = 0 + 0.1* 0 = 0



	Actions : ↑	→	↓	←
Start	0	0	0	0
Nothing / Blank	0	0	0	0
Power	0	0	0	0
Mines	0	0	0	0
END	0	0	0	0

We will repeat this again and again until the learning is stopped. In this way the Q-Table will be updated.

At last…let us recap

- Q-Learning is a value-based reinforcement learning algorithm which is used to find the optimal action-selection policy using a Q function.
- Our goal is to maximize the value function Q.
- The Q table helps us to find the best action for each state.
- It helps to maximize the expected reward by selecting the best of all possible actions.
- Q(state, action) returns the expected future reward of that action at that state.
- This function can be estimated using Q-Learning, which iteratively updates Q(s,a) using the **Bellman equation.**
- Initially we explore the environment and update the Q-Table. When the Q-Table is ready, the agent will start to exploit the environment and start taking better actions.