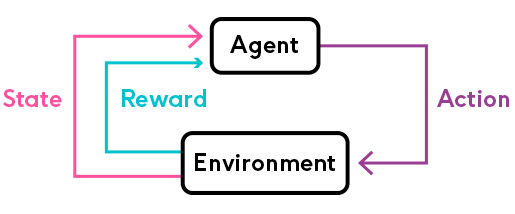
**Reinforcement Learning**

Reinforcement Learning (RL) is concerned with producing algorithms (agents) that try to achieve some predefined goal. The achievement of this objective is dependent on choosing a set of actions — receiving rewards for the good ones and punishment for the bad ones — the reinforcement bit comes from this. The agent act, in environments that have a state, gives rewards, and a set of actions.



**Markov Decision Processes**

Markov Decision Process (MDP) is a mathematical formulation of the RL problem. MDPs satisfy the Markov property:

Markov property — the current state completely represents the state of the environment (world). That is, the future depends only on the present.

An MDP can be defined by (S, A, R, P, γ) where:

* S — set of possible states
* A — set of possible actions
* R — probability distribution of reward given (state, action) pair
* P — probability distribution over how likely any of the states is to be the new states, given (state, action) pair. Also known as transition probability.
* γ — reward discount factor

The discount factor γ allows us to inject the heuristic that 100 bucks now are more valuable than 100 bucks in 30 days. A discount factor of 1 would make future rewards worth just as much as immediate rewards.

Another reason to discount future rewards:

“In the long run we are all dead “ - J. M. Keynes

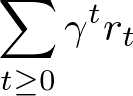
**Learning**

Here is how learning happens in RL context:

for time step t=0 until done:

1. The environment gives your agent a state
2. Your agent chooses an action (from a set of possible ones)
3. The environment gives a reward along with a new state
4. Continue until the goal or other condition is met

What is the objective of all this? Find a function π∗, known as optimal policy, that maximizes the cumulative discounted reward:

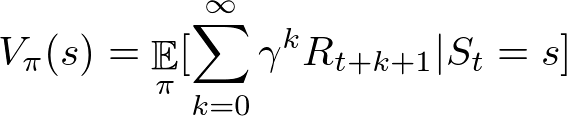


where rt​ is the reward received at step t and γt is the discount factor at step tt.

A policy π is a function that maps state s to action a, that our agent believes is the best given that state.

**Value function**

The value function gives you the maximum expected future reward the agent will get, starting from some state s and following some policy π.

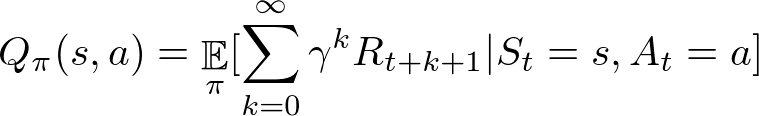


There exists an **optimal value function** that has the highest value for all states. Defined as:

https://miro.medium.com/max/633/1*W11hMqE83Su4eqhBUlCqog.png

**Q-function**

Similarly, let’s define another function known as Q-function (state-action value function) that gives the expected return starting from state s, taking action aa, and following policy π.



You can think of the Q-function as the “quality” of a certain action given a state. We can define the optimal Q-function as:

https://miro.medium.com/max/733/1*ArO7a1PAj_LrvLzGxKPJdw.png

There is a relationship between the two optimal functions V∗ and Q∗. It is given byhttps://miro.medium.com/max/679/1*o6Z8g3KpN4hnOjNR8bpzdQ.png

That is, the maximum expected total reward when starting at s is the maximum of Q∗(s, a) over all possible actions.

We can find the optimal policy π∗ by choosing the action a that gives maximum reward Q∗(s, a) for state s:

https://miro.medium.com/max/750/1*DE1o2hq2MAXTT_FmXsg-FA.png

There seems to be a synergy between all functions we defined so far. More importantly, we can now build an optimal agent for a given environment. Can we do it in practice?

**Q-learning**

Q-Learning is an off-policy algorithm for [Temporal Difference (TD) learning](https://en.wikipedia.org/wiki/Temporal_difference_learning). It is proven that with enough training, it converges with probability 1 to a close approximation of the action-value function for an arbitrary target policy. It learns the optimal policy even when actions are selected using exploratory (some randomness) policy (off-policy).

Given a state s and action a, we can express Q(s, a) in terms of itself (recursively):

https://miro.medium.com/max/635/1*jamiG5MkFVHLTFmLggemVg.png

This is the **Bellman equation**. It defines the maximum future reward as the reward r the agent received, for being at state s, plus the maximum future reward for state s′ for every possible action.

We can define Q-learning as an iterative approximation of Q∗ using the Bellman equation:

https://miro.medium.com/max/1494/1*2APPuDNSpWMlaiin7R0C9g.png

where α is the learning rate that controls how much the difference between previous and new Q value is considered.

Here’s the general algorithm we’re going to implement:

1. Initialize a Q-values table
2. Observe initial state s
3. Choose action a and act
4. Observe reward r and a new state s′
5. Update the Q table using r and the maximum possible reward from s′
6. Set the current state to the new state and repeat from step 2 until a terminal state

**Exploration vs exploitation**

You might’ve noticed that we glanced over the strategy on choosing an action. Any strategy you implement will have to choose how often to try something new or use something it already knows. This is known as the exploration/exploitation tradeoff.

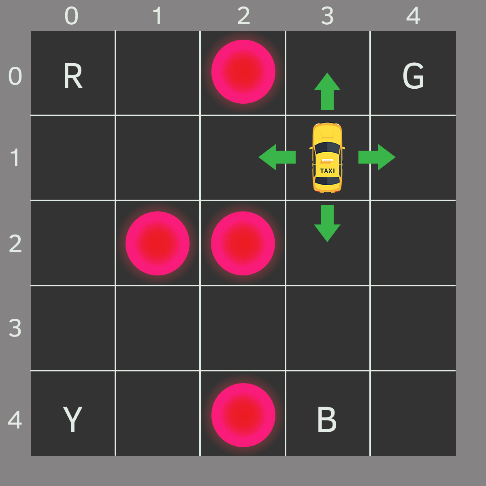
* Exploration — finding new information about the environment
* Exploitation — using existing information to maximize the reward

Remember, the goal of our RL agent is to maximize the expected cumulative reward. What does this mean for our self-driving taxi?

Initially, our driving agent knows pretty much nothing about the best set of driving directions for picking up and dropping off passengers. It should learn to avoid anomalies too since they are bad for business (passengers tend to disappear or worse in those)! During this time, we expect to make a lot of exploration.

After obtaining some experience, the agent can use it to choose an action more and more often. Eventually, all choices will be based on what is learned.

**Driving in a post-apocalyptic world**



Your partner sketched this map. Each block represents a small region of the city. Anomalies are marked in bright circles. The four letters are “safe zones”, where pickups and drop-offs happen.

Let’s assume your car is the only vehicle in the city (not much of a stretch). We can break it up into a 5x5 grid, which gives us 25 possible taxi locations.

The current taxi location is (3, 1) in (row, col) coordinates. The pickup and drop off locations are: [(0,0), (0,4), (4,0), (4,3)]. Anomalies are at [(0, 2), (2, 1), (2, 2), (4, 2)].

**Environment**

We’re going to encode our city map into an environment for the self-driving agent using [OpenAI’s Gym](https://gym.openai.com/" \t "_blank). What is this Gym thing anyways?

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from [walking](https://gym.openai.com/envs/Humanoid-v2/) to playing games like [Pong](https://gym.openai.com/envs/Pong-ram-v0/) or [Pinball](https://gym.openai.com/envs/VideoPinball-ram-v0/).

The most important entity in Gym is the Environment. It provides a unified and easy to use interface. Here are the most important methods:

* reset - resets the environment and returns a random initial state
* step(action) - take action and advance one timestep. It returns:

observation — the new state of the environment

reward — reward received from taking the action

done — most environments are divided into well-defined episodes and if done is True it indicates the episode has been completed

info — additional information about the environment (might be useful for debugging)

* render - renders one frame of the environment (useful for visualization)

The complete source code of the environment is in the notebook. Here, we’ll take a look at the registration to Gym:

Note that we set a timestep\_limit which limits the number of steps in an episode.

**Action Space**

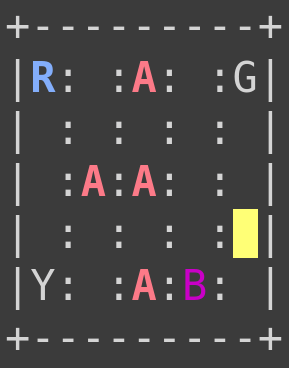
Our agent encounters one of the 500 states (5 rows x 5 columns x 5 passenger locations x 4 destinations) and it chooses an action. Here are the possible actions:

1. south
2. north
3. east
4. west
5. pickup
6. dropoff

You might notice that in the illustration above, the taxi cannot perform certain actions when near the boundaries of the city. We will penalize this with -1 and won’t move the taxi if such situation occurs.

**Getting a taste of the environment**

Let’s have a look at our encoded environment:



Action Space Discrete(6)  
State Space Discrete(500)

Here is the action to number mapping:

* 0 = south
* 1 = north
* 2 = east
* 3 = west
* 4 = pickup
* 5 = dropoff

**Building an agent**

Our agent has a simple interface for interacting with the environment. Here it is:

We’re going to use the choose\_action method when we want our agent to make a decision and act. Then, after the reward and new state from the environment are observed, our agent will learn from its actions using the learn method.

Let’s take a look at the implementations:

The interesting part of the \_\_init\_\_ is the initialization of our QQ-table. Initially, it is all zeros. Can we initialize it in some other way?

Our strategy is rather simple. We draw a random number from a uniform distribution between 0 and 1. If this number is smaller than epsilon and we want to explore, we take a random action. Otherwise, we take the best action based on our current knowledge.

Learning involves the update of the Q-table using the Q-learning equation and reducing the exploration rate ϵ if the episode is complete.

**Training**

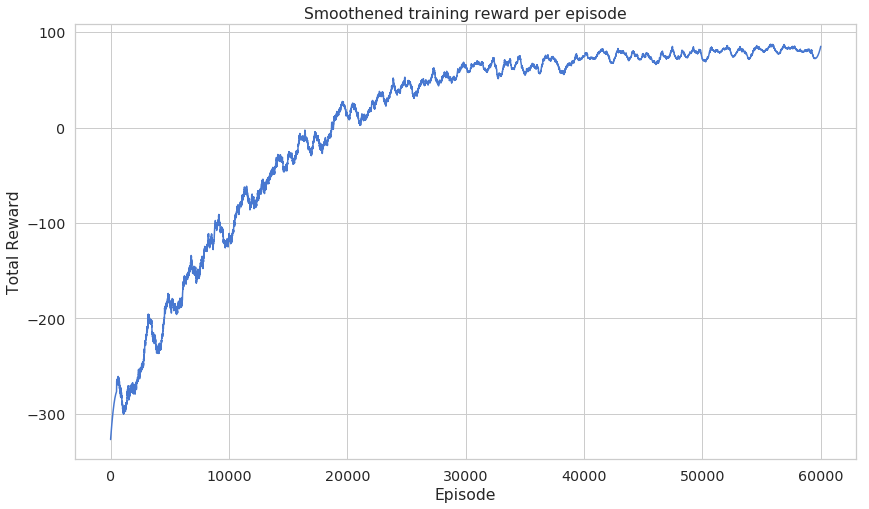
Now that our agent is ready for action, we can train it on the environment we’ve created. Let’s train our agent for 50k episodes and record episode rewards over time:

Let’s have a look at our agent driving before learning anything about the environment:

Recall that we set timestep\_limit when we registered the environment so our agent won’t stay in an episode infinitely.

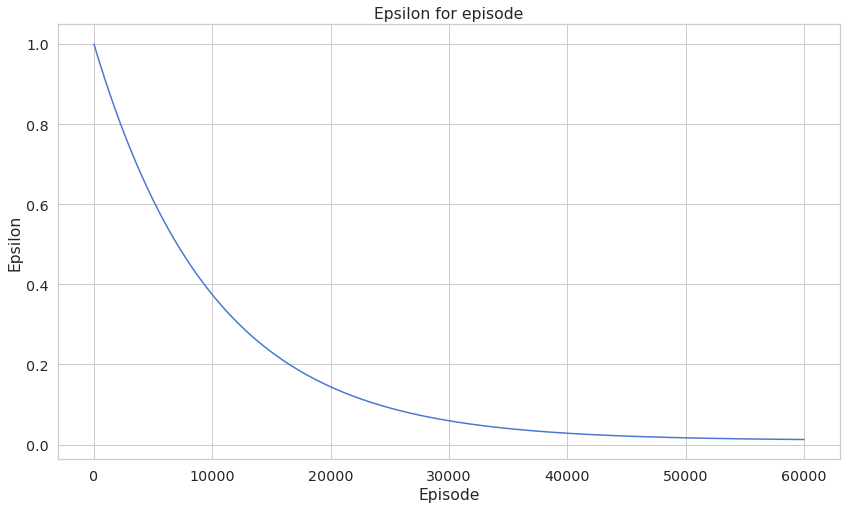
**Evaluation**

Let’s have a look at reward changes as the training progresses:



Note that the learning curve is smoothened using [savgol\_filter](https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.savgol_filter.html" \t "_blank)savgol\_filter(rewards, window\_length=1001, polyorder=2)

Recall that our exploration rate should decrease as our agent is learning. Have a look:



Here’s how we’re going to test our agent and record the progress:

Note that we want our agent to use only the experience it has so we set explore=False. Here’s what the total reward for each episode looks like:



I know that this chart might not give you a good idea of what the agent is capable of.

# Conclusion

Congratulations on building a self-driving taxi agent. You’ve learned how to

* Build your own environment based on one provided by OpenAI’s Gym
* Implement and apply Q-learning
* Build an agent that learns to pick up, drop off passengers and avoid danger areas