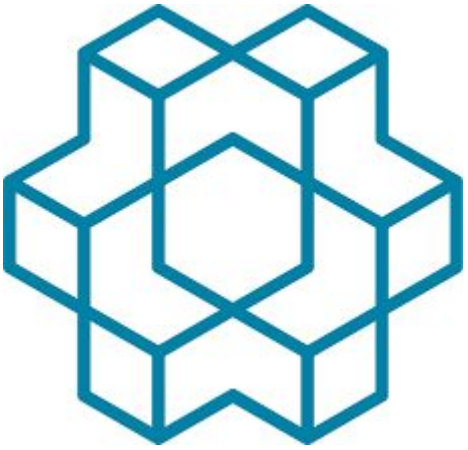
OpenAI GYM



Index

What's OpenAI Gym?

OpenAI is a non-profit research company that is focussed on building out AI in a way that is good for everybody. It was founded by Elon Musk and Sam Altman. OpenAI's mission as stated on their website is to "build safe AGI, and ensure AGI's benefits are as widely and evenly distributed as possible".

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like pong or pinball. Gym is an open source interface to reinforcement learning tasks. Gym provides an environment and its is upto the developer to implement any reinforcement learning algorithms.

Among the vast quantity of environments available for OpenAI Gym, there are a lot of ATARI 2600 games for you to take the challenge of achieving the highest score possible.

But enough boring talk about the background of OpenAI Gym, let's go to the fun part!

Installation and use of OpenAI Gym

The installation guide of the toolkit that we will provide is meant to be followed by a computer with Ubuntu SO, and python3 or greater.
First of all, you have to install python 3, the numpy library, python3-pip, and some other libraries and tools, with this commands:

```
sudo apt-get install -y python3-numpy python3-dev python3-
pip cmake zlib1g-dev libjpeg-dev xvfb libav-tools xorg-dev
python-opengl libboost-all-dev libsdl2-dev swig
```

After that, you will clone the github repository to your computer, and proceed to install the toolkit:

```
cd ~
git clone https://github.com/openai/gym.git
cd gym
sudo pip3 install -e '.[all]'
```

With those two steps, the installation will be completed, so let's see how you can use OpenAI Gym to develop some projects.

Now you have to open your python editor, and begin importing gym, and numpy libraries like this:

```
import gym
import numpy as np
```

Then you create an environment, and after that, reset to initialize it:

```
env = gym.make("Taxi-v2")
env.reset()
```

In this case, the environment created will be a big square, with a yellow square inside it which represents the taxi, "|" character that represent a wall, and 4 letters, one blue colored that represent the pick-up location, and another one purple where is the drop-off location.
When the taxi has a passenger, the color will turn green.

After this, you can see the initial state of the problem with this line:

```
env.render()
```

Now you should start making some actions to see how the problem changes, the available actions in the environment, can be seen with that line:

```
env.action_space.n
env.env.get_action_meanings()
```

The first line just tell you how many actions can be done with an integer, and the second one will tell you to what action corresponds each number.

So, in this case of the taxi environment the number 1 corresponds to 'move up one position', if you want to make this movement you have to use the step() method like this:

```
env.step(1)
env.render()
```

The step() method will return four variables that you will call: state, reward, done and info.

```
state, reward, done, info = env.step(1)
```

The 'state' variable corresponds to the new state that will take the environment, 'reward' makes reference to what will earn our agent making that action, 'done' will be a boolean stating whether the environment is terminated or done, and 'info' will be extra information for debugging.

When the taxi successfully pick up a passenger and drop them off at their desired location, you will receive a reward of 20 pints and 'done' will equal 'True', and for each time the agent incorrectly pick up or drop off a passenger, the environment will give a -10 reward.

To solve the environment, and evaluate the agent's performance, you have to compare it to a completely random agent, and to choose a random action you can use 'env.action_space.sample()' like this:

```
state, reward, done, info = env.step(env.action_space.sample())
```

And now you can create a loop to solve randomly the environment.
You can also put a 'counter' variable to see how many steps it takes to solve it.

```
state = env.reset()
counter = 0
reward = None
while reward != 20:
    state, reward, done, info = env.step(env.action_space.sample())
    counter += 1

print(counter)
```

This method will take 2000+ steps on average, to solve the environment, so in order to improve it, and maximize the reward, we will have to have the algorithm remember its actions and their associated rewards.

The algorithm memory is going to be a Q action value table.

```
import numpy as np
Q = np.zeros([env.observation_space.n, env.action_space.n])
```

You also have to define some variables like ‘G’ where you will accumulate the total reward, and ‘alpha’ that is a learning rate, and each one can use what he likes, in my case 0.618.

```
G = 0
alpha = 0.618
```

And then we can implement a basic Q learning algorithm like that:

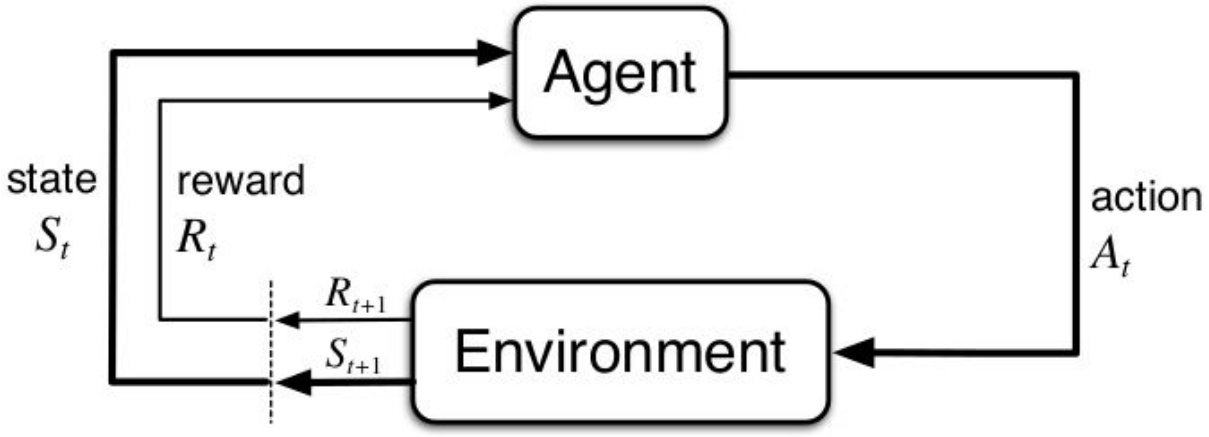
```
for episode in range(1,251):
    done = False
    G, reward = 0,0
    state = env.reset()
    while done != True:
            action = np.argmax(Q[state]) #1
            state2, reward, done, info = env.step(action) #2
            Q[state,action] += alpha * (reward + np.max(Q[state2]) - Q[state,action]) #3
            G += reward
            state = state2
            env.render()
    if episode % 50 == 0:

        print('Episode {} Total Reward: {}'.format(episode,G))
```

This algorithm will solve the problems, basing each action on a estimated reward, thanks to the formula which decides what action to make in each state.

Reinforced Learning

Reinforcement learning, explained simply, is a computational approach where an agent interacts with an environment by taking actions in which it tries to maximize an accumulated reward. Here is a simple graph, which I will be referring to often:



An agent in a current state (S_t) takes an action (A_t) to which the environment reacts and responds, returning a new state(S_{t+1}) and reward (R_{t+1}) to the agent. Given the updated state and reward, the agent chooses the next action, and the loop repeats until an environment is solved or terminated.

These four variables are: the new **state** (S_{t+1} = 14) (S_t is a number in the range 0..n-1, where n is the number of possible states in the environment.), **reward** (R_{t+1} = -1), a boolean stating whether the environment is terminated or **done**, and extra **info** for debugging. Every Gym environment will return these same four variables after an action is taken, as they are the core variables of a reinforcement learning problem.

You may luck out and solve the environment fairly quickly, but on average, a completely random policy will solve this environment in about 2000+ steps, so in order to maximize our reward, we will have to have the algorithm remember its actions and their associated rewards. In this case, the algorithm's memory is going to be a Q action value table. To manage this Q table, we will use a NumPy array. The size of this table will be the number of states (500) by the number of possible actions (6).

Number of possible states (s)	← Number of possible actions (a) →		
	Q(0, 0)	...	Q(0, a)
	...	...	...
	Q(s, 0)	...	Q(s, a)

Over multiple episodes of trying to solve the problem, we will be updating our Q values, slowly improving our algorithm's efficiency and performance. We will also want to track our total accumulated reward for each episode, which we will define as **G**.

Similar to most machine learning problems, we will need a learning rate as well. I will use my personal favorite of 0.618, also known as the mathematical constant phi.

```
for episode in range(1,1001):
        done = False
        G, reward = 0,0
        state = env.reset()
        while done != True:
                action = np.argmax(Q[state]) #1
                state2, reward, done, info = env.step(action) #2
                Q[state,action] += alpha * (reward + np.max(Q[state2]) - Q[state,action]) #3
                G += reward
                state = state2
        if episode % 50 == 0:
print('Episode {} Total Reward: {}'.format(episode,G))
```

This code alone will solve the environment. There is a lot going on in this code, so I will try and break it down.
First (#1): The agent starts by choosing an **action** with the highest **Q** value for the current state using argmax. Argmax will return the index/action with the highest value for that state. Initially, our Q table will be all zeros. But, after every step, the Q values for state-action pairs will be updated.
Second (#2): The agent then takes **action** and we store the future state as **state2** (S_{t+1}). This will allow the agent to compare the previous state to the new state.
Third (#3): We update the state-action pair (S_t , A_t) for **Q** using the **reward**, and the max **Q** value for **state2** (S_{t+1}). This update is done using the action value formula (based upon the Bellman equation) and allows state-action pairs to be updated in a recursive fashion (based on future values). See Figure 2 for the value iteration update.

$$Q_{t+1}(s_t,a_t) = \underbrace{Q_t(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t,a_t)}_{\text{learning rate}} \times \left[\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q_t(s_{t+1},a_t)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q_t(s_t,a_t)}_{\text{old value}} \right]$$

References

Introduction to reinforcement learning and OpenAI Gym

Github

Understanding OpenAI Gym

Complete OpenAI Guide

Official Webpage