Reinforcement Learning

Shageenth Sandrakumar
Department of Data Science and Analytics
Buffalo, NY 14226
shageent@buffalo.edu

Abstract

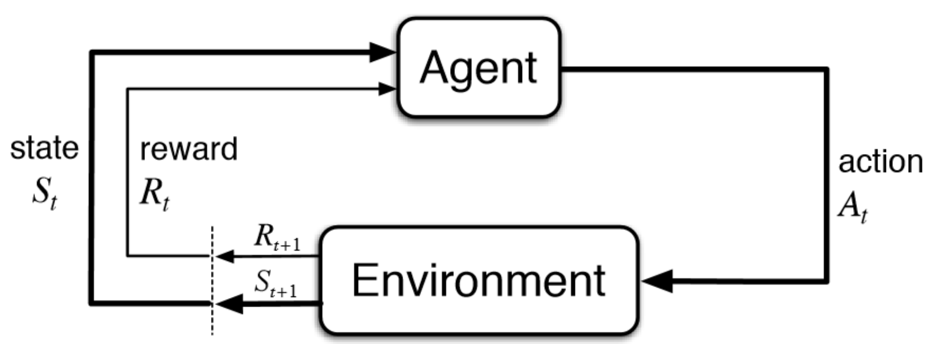
In this lab, we are required to work with Reinforcement Learning, a newer Machine Learning technique, that can train an agent in an environment. Thea gent will navigate the classic 4x4 grid-world environment to a specific goal. The agent will learn an optimal policy through Q-Learning which will allow it to take actions to reach the goal while avoiding the boundaries. We use a platform here called AI gym to facilitate the whole process of the construction of the agent and the environment.

1 Introduction

1.1 What is Reinforcment Learning?

Reinforcement Learning(RL) is one of the hottest research topics in the field of modern Artificial Intelligence and its popularity is only growing. Reinforcement Learning(RL) is a type of machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences. Though both supervised and reinforcement learning use mapping between input and output, unlike supervised learning where the feedback provided to the agent is correct set of actions for performing a task, reinforcement learning uses rewards and punishments as signals for positive and negative behavior.

As compared to unsupervised learning, reinforcement learning is different in terms of goals. While the goal in unsupervised learning is to find similarities and differences between data points, in the case of reinforcement learning the goal is to find a suitable action model that would maximize the total cumulative reward of the agent. The figure below illustrates the action-reward feedback loop of a generic RL model.



2. Open AI Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano.The gym library is a collection of test problems — environments — that you can use to work out your reinforcement learning algorithms. These environments have a shared interface, allowing you to write general algorithms.

3. The Envriomment

The Environment is a 4 by 4 grid environment described by two things the grid environment and the agent. An observation space which is defined as a vector of elements. This can be particularly useful for environments which return measurements, such as in robotic environments.

The core gym interface is `env`, which is the unified environment interface. The following are the `env` methods that would be quite helpful to us:

`env.reset`: Resets the environment and returns a random initial state.

`env.step(action)`: Step the environment by one timestep.

Returns observation: Observations of the environment

reward: If your action was beneficial or not

done: Indicates if we have successfully picked up and dropped off a passenger, also called one *episode*

info: Additional info such as performance and latency for debugging purposes

`env.render`: Renders one frame of the environment (helpful in visualizing the environment)

4. We have an Action Space of size 4

0 = down

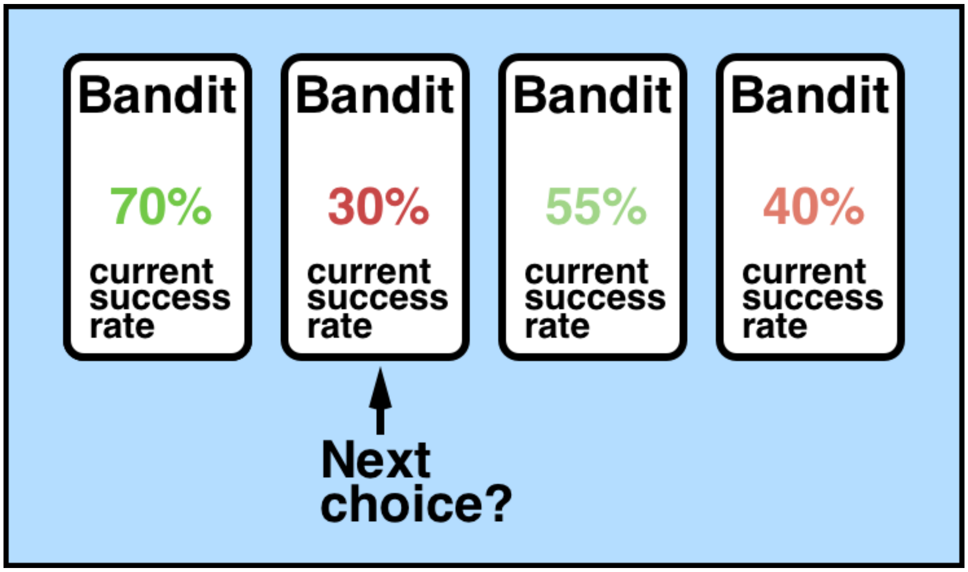
1 = up

2 = right

3 = left

5. The MULti-Bandit Problem (E-Greedy Algorithm)

The multi-armed bandit problem is a classic reinforcement learning example where we are given a slot machine with *n* arms (bandits) with each arm having its own rigged probability distribution of success. Pulling any one of the arms gives you a stochastic reward of either R=+1 for success, or R=0 for failure. Our objective is to pull the arms one-by-one in sequence such that we maximize our total reward collected in the long run.



The non-triviality of the multi-armed bandit problem lies in the fact that we (the agent) cannot access the true bandit probability distributions — all learning is carried out via the means of trial-and-error and value estimation. So the question is:

This is our goal for the multi-armed bandit problem, and having such a strategy would prove very useful in many real-world situations where one would like to select the “best” bandit out of a group of bandits.

In this project, we approach the multi-armed bandit problem with a classical reinforcement learning technique of an *epsilon-greedy agent* with a learning framework of *reward-average sampling* to compute the action-value Q(a) to help the agent improve its future action decisions for long-term reward maximization.

In a nutshell, the epsilon-greedy agent is a hybrid of a (1) completely-exploratory agent and a (2) completely-greedy agent. In the multi-armed bandit problem, a completely-exploratory agent will sample all the bandits at a uniform rate and acquire knowledge about every bandit over time; the caveat of such an agent is that this knowledge is never utilized to help itself to make better future decisions! On the other extreme, a completely-greedy agent will choose a bandit and stick with its choice for the rest of eternity; it will not make an effort to try out other bandits in the system to see whether they have better success rates to help it maximize its long-term rewards, thus it is very narrow-minded!

How do we perform this in our code?

We perform this by assigning a variable called epsilon. This epsilon switches between the exploratory and the greedy agent. We choose a random number between 0 and 1, if this number is less than epsilon, we tell the agent to explore if it is greater then we tell the agent to be greedy. This tactic is used in our policy part of our code.

Q-Learning Algorithm

Essentially, Q-learning lets the agent use the environment's rewards to learn, over time, the best action to take in a given state.

In our environment, we have the reward table,that the agent will learn from. It does thing by looking receiving a reward for taking an action in the current state, then updating a *Q-value* to remember if that action was beneficial.

The values store in the Q-table are called a *Q-values*, and they map to a (state,action) combination.

A Q-value for a particular state-action combination is representative of the "quality" of an action taken from that state. Better Q-values imply better chances of getting greater rewards.

Q-values are initialized to an arbitrary value, and as the agent exposes itself to the environment and receives different rewards by executing different actions, the Q-values are updated using the equation:

$$\mathbf{Q(state,action) \leftarrow (1-\alpha)Q(state,action)+\alpha(reward+\gamma maxaQ(nextstate,allactions))}$$

Where:

- α(alpha) is the learning rate ($0<\alpha\leq1$) - Just like in supervised learning settings, α is the extent to which our Q-values are being updated in every iteration.

- γ (gamma) is the discount factor ($0\leq\gamma\leq1$) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to 1) captures the long-term effective award, whereas, a discount factor of 0 makes our agent consider only immediate reward, hence making it greedy.

What is this saying?

We are assigning ($\leftarrow$), or updating, the Q-value of the agent's current *state* and *action* by first taking a weight ($1-\alpha$) of the old Q-value, then adding the learned value. The learned value is a combination of the reward for taking the current action in the current state, and the discounted maximum reward from the next state we will be in once we take the current action.

Basically, we are learning the proper action to take in the current state by looking at the reward for the current state/action combo, and the max rewards for the next state. This will eventually cause our taxi to consider the route with the best rewards strung together.

The Q-value of a state-action pair is the sum of the instant reward and the discounted future reward (of the resulting state). The way we store the Q-values for each state and action is through a Q-table

Q-Table

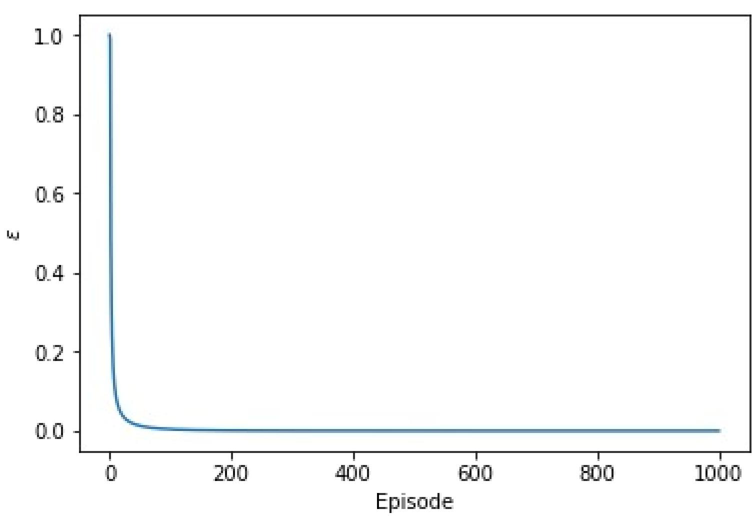
The Q-table is a matrix where we have a row for every state and a column for every action. It's first initialized to 0, and then values are updated after training.

6. Epsilon

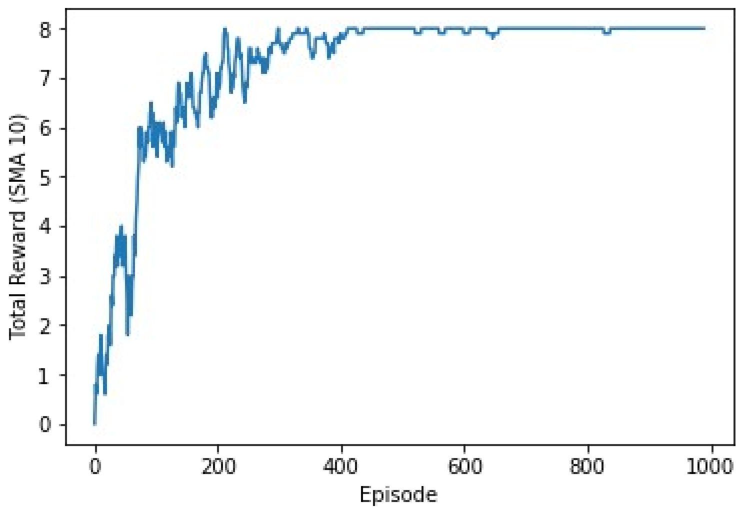
   We want the odds of the agent exploring to decrease as time goes on. One way to do this is by updated the epsilon every moment of the agent during the training phase. Choosing a decay rate was the difficult part. We want episilon not to decay too soon, so the agent can have time to explore. The way I went about this is I want the agent the ability to explore the region for at least the size of the grid movements. So in 25 movements the agent should still have a good chance of being in exploratory phase. I chose decay = 1/1.01 because (1/1.01)^25 = 75% which is a good chance and still being in the exploratory phase within 0-25 moves. After 50 moves episilon goes to 0.6 which is good odds of agent doing both exploring and action orientated.

7. Results and Charts

   Here we plot epsilon vs episode.
   We see that we have a exponential decay going on here.



Then we plot rewards vs episode



Here we see that the total rewards slowly goes up then plateus after hitting 8. This is because 8 is the optimal path for our algrotihm.

CONCLUSION

Our Reinforcement Algorithm had done fairly well. Our agent has been able to learn the material in an effective manner. It has been able to reach its goal in 8 steps.