Arthur Juliani [Follow]

Deep Learning @Unity3D & Cognitive Neuroscience PhD student.

Aug 25, 2016 · 6 min read

Simple Reinforcement Learning with Tensorflow Part 0: Q-Learning with Tables and Neural Networks

We'll be learning how to solve the OpenAI FrozenLake environment. Our version is a little less photo-realistic.

For this tutorial in my Reinforcement Learning series, we are going to be exploring a family of RL algorithms called Q-Learning algorithms. These are a little different than the policy-based algorithms that will be looked at in the the following tutorials (Parts 1–3). Instead of starting with a complex and unwieldy deep neural network, we will begin by implementing a simple lookup-table version of the algorithm, and then show how to implement a neural-network equivalent using Tensorflow. Given that we are going back to basics, it may be best to think of this as Part-0 of the series. It will hopefully give an intuition into what is really happening in Q-Learning that we can then build on going forward when we eventually combine the policy gradient and Q-learning approaches to build state-of-the-art RL agents (If you are more interested in Policy Networks, or already have a grasp on Q-Learning, feel free to start the tutorial series here instead).

Unlike policy gradient methods, which attempt to learn functions which directly map an observation to an action, Q-Learning attempts to learn the value of being in a given state, and taking a specific action there. While both approaches ultimately allow us to take intelligent actions given a situation, the means of getting to that action differ significantly. You may have heard about DeepQ-Networks which can play Atari Games. These are really just larger and more complex implementations of the Q-Learning algorithm we are going to discuss here.

Tabular Approaches for Tabular Environments

```
SFFF        (S: starting point, safe)
FHFH        (F: frozen surface, safe)
FFFH        (H: hole, fall to your doom)
HFFG        (G: goal, where the frisbee is located)
```

The rules of the FrozenLake environment.

For this tutorial we are going to be attempting to solve the FrozenLake environment from the OpenAI gym. For those unfamiliar, the OpenAI gym provides an easy way for people to experiment with their learning agents in an array of provided toy games. The FrozenLake environment consists of a 4x4 grid of blocks, each one either being the start block, the goal block, a safe frozen block, or a dangerous hole. The objective is to have an agent learn to navigate from the start to the goal without moving onto a hole. At any given time the agent can choose to move either up, down, left, or right. The catch is that there is a wind which occasionally blows the agent onto a space they didn't choose. As such, perfect performance every time is impossible, but learning to avoid the holes and reach the goal are certainly still doable. The reward at every step is 0, except for entering the goal, which provides a reward of 1. Thus, we will need an algorithm that learns long-term expected rewards. This is exactly what Q-Learning is designed to provide.

In it's simplest implementation, Q-Learning is a table of values for every state (row) and action (column) possible in the environment. Within each cell of the table, we learn a value for how good it is to take a given action within a given state. In the case of the FrozenLake environment, we have 16 possible states (one for each block), and 4 possible actions

(the four directions of movement), giving us a 16x4 table of Q-values. We start by initializing the table to be uniform (all zeros), and then as we observe the rewards we obtain for various actions, we update the table accordingly.

We make updates to our Q-table using something called the <u>Bellman equation</u>, which states that the expected long-term reward for a given action is equal to the immediate reward from the current action combined with the expected reward from the best future action taken at the following state. *In this way, we reuse our own Q-table when estimating how to update our table for future actions!* In equation form, the rule looks like this:

$$Eq\ 1.\ Q(s,a) = r + \gamma(max(Q(s',a')))$$

This says that the Q-value for a given state (s) and action (a) should represent the current reward (r) plus the maximum discounted (γ) future reward expected according to our own table for the next state (s') we would end up in. The discount variable allows us to decide how important the possible future rewards are compared to the present reward. By updating in this way, the table slowly begins to obtain accurate measures of the expected future reward for a given action in a given state. Below is a Python walkthrough of the Q-Table algorithm implemented in the FrozenLake environment:

Q-Table Learning

In [ ]:

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

Load the environment

In [ ]:

```
env = gym.make('FrozenLake-v0')
```

Implement Q-Table learning
algorithm

(Thanks to <u>Praneet D</u> for finding the optimal hyperparameters for this approach)

Q-Learning with Neural Networks

Now, you may be thinking: tables are great, but they don't really scale, do they? While it is easy to have a 16x4 table for a simple grid world, the number of possible states in any modern game or real-world environment is nearly infinitely larger. For most interesting problems, tables simply don't work. We instead need some way to take a description of our state, and produce Q-values for actions without a table: that is where neural networks come in. By acting as a function approximator, we can take any number of possible states that can be represented as a vector and learn to map them to Q-values.

In the case of the FrozenLake example, we will be using a one-layer network which takes the state encoded in a one-hot vector (1x16), and produces a vector of 4 Q-values, one for each action. Such a simple network acts kind of like a glorified table, with the network weights serving as the old cells. The key difference is that we can easily expand the Tensorflow network with added layers, activation functions, and different input types, whereas all that is impossible with a regular table. The method of updating is a little different as well. Instead of directly updating our table, with a network we will be using backpropagation and a loss function. Our loss function will be sum-of-squares loss, where the difference between the current predicted Q-values, and the "target" value is computed and the gradients passed through the network. In this case, our Q-target for the chosen action is the equivalent to the Q-value computed in equation 1 above.

Eq2. Loss = Σ(Q-target - Q)²

Below is the Tensorflow walkthrough of implementing our simple Q-Network:

Q-Network Learning

In [1]:

```
import gym
import numpy as np
import random
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
```

Load the environment

In [2]:

```
env = gym.make('FrozenLake-v0')
```

While the network learns to solve the FrozenLake problem, it turns out it doesn't do so quite as efficiently as the Q-Table. While neural networks allow for greater flexibility, they do so at the cost of stability when it comes to Q-Learning. There are a number of possible extensions to our simple Q-Network which allow for greater performance and more robust learning. Two tricks in particular are referred to as Experience Replay and Freezing Target Networks. Those improvements and other tweaks were the key to getting Atari-playing Deep Q-Networks, and we will be exploring those additions in the future. For more info on the theory behind Q-Learning, see this great post by Tambet Matiisen. I hope this tutorial has been helpful for those curious about how to implement simple Q-Learning algorithms!

. . .

If this post has been valuable to you, please consider *donating* to help support future tutorials, articles, and implementations. Any contribution is greatly appreciated!

If you'd like to follow my work on Deep Learning, AI, and Cognitive Science, follow me on Medium @Arthur Juliani, or on Twitter @awjliani.

. . .

More from my Simple Reinforcement Learning with Tensorflow series: