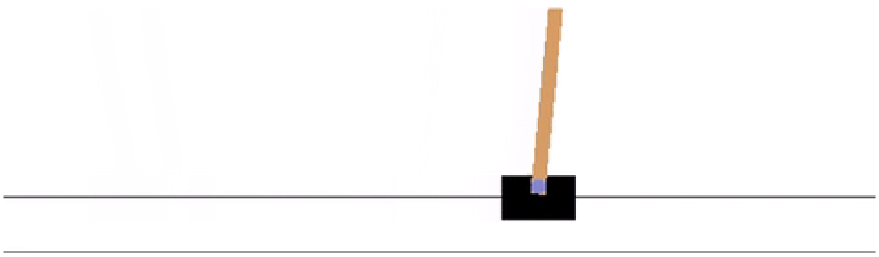
Arthur Juliani Follow
Deep Learning @Unity3D & Cognitive Neuroscience PhD student.
Jun 24, 2016 · 4 min read

Simple Reinforcement Learning with Tensorflow: Part 2 - Policy-based Agents



After a weeklong break, I am back again with part 2 of my Reinforcement Learning tutorial series. In Part 1, I had shown how to put together a basic agent that learns to choose the more rewarding of two possible options. In this post, I am going to describe how we get from that simple agent to one that is capable of taking in an *observation* of the world, and taking *actions* which provide the optimal *reward* not just in the present, but over the long run. With these additions, we will have a full reinforcement agent.

Environments which pose the full problem to an agent are referred to as Markov Decision Processes (MDPs). These environments not only provide rewards and state transitions given actions, but those rewards are also condition on the state of the environment and the action the agent takes within that state. These dynamics are also temporal, and can be delayed over time.

To be a little more formal, we can define a Markov Decision Process as follows. An MDP consists of a set of all possible states `S` from which our agent at any time will experience `s`. A set of all possible actions

`A` from which our agent at any time will take action `a`. Given a state action pair `(s, a)`, the transition probability to a new state `s’` is defined by `T(s, a)`, and the reward `r` is given by `R(s, a)`. As such, at any time in an MDP, an agent is given a state `s`, takes action `a`, and receives new state `s’` and reward `r`.

While it may seem relatively simple, we can pose almost any task we could think of as an MDP. For example, imagine opening a door. The state is the vision of the door that we have, as well as the position of our body and door in the world. The actions are our every movement our body could make. The reward in this case is the door successfully opening. Certain actions, like walking toward the door are essential to solving the problem, but aren’t themselves reward-giving, since only actually opening the door will provide the reward. In this way, an agent needs to learn to assign value to actions the lead eventually to the reward, hence the introduction of temporal dynamics.

Cart-Pole Task

In order to accomplish this, we are going to need a challenge that is more difficult for the agent than the two-armed bandit. To meet provide this challenge we are going to utilize the OpenAI gym, a collection of reinforcement learning environments. We will be using one of the classic tasks, the Cart-Pole. To learn more about the OpenAI gym, and this specific task, check out their tutorial here. Essentially, we are going to have our agent learn how to balance a pole for as long as possible without it falling. Unlike the two-armed bandit, this task requires:

- *Observations*—The agent needs to know where pole currently is, and the angle at which it is balancing. To accomplish this, our neural network will take an observation and use it when producing the probability of an action.

- *Delayed reward—K*eeping the pole in the air as long as possible means moving in ways that will be advantageous for both the present and the future. To accomplish this we will adjust the reward value for each observation-action pair using a function that weighs actions over time.

To take reward over time into account, the form of Policy Gradient we used in the previous tutorials will need a few adjustments. The first of which is that we now need to update our agent with more than one experience at a time. To accomplish this, we will collect experiences in a buffer, and then occasionally use them to update the agent all at once. These sequences of experience are sometimes referred to as rollouts, or experience traces. We can't just apply these rollouts by themselves however, we will need to ensure that the rewards are properly adjusted by a discount factor

Intuitively this allows each action to be a little bit responsible for not only the immediate reward, but all the rewards that followed. We now use this modified reward as an estimation of the advantage in our loss equation. With those changes, we are ready to solve CartPole!

Let's get to it!

awjuliani/DeepRL-Agents

DeepRL-Agents - A set of Deep Reinforcement Learning Agents implemented in Tensorflow.

github.com

And with that we have a fully-functional reinforcement learning agent. Our agent is still far from the state of the art though. While we are using a neural network for the policy, the network still isn't as deep or complex as the most advanced networks. In the next post I will be showing how to use Deep Neural Networks to create agents able to learn more complex relationships with the environment in order to play a more exciting game than pole balancing. In doing so, I will be diving into the kinds of representations that a network learns for more complex environments.

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.

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

More from my Simple Reinforcement Learning with Tensorflow series:

1. *Part 0—Q-Learning Agents*
2. *Part 1—Two-Armed Bandit*
3. *Part 1.5—Contextual Bandits*
4. ***Part 2—Policy-Based Agents***
5. *Part 3—Model-Based RL*
6. *Part 4—Deep Q-Networks and Beyond*
7. *Part 5—Visualizing an Agent's Thoughts and Actions*
8. *Part 6—Partial Observability and Deep Recurrent Q-Networks*
9. *Part 7—Action-Selection Strategies for Exploration*
10. *Part 8—Asynchronous Actor-Critic Agents (A3C)*