Arthur Juliani Follow
Deep Learning @Unity3D & Cognitive Neuroscience PhD student.
Jun 15, 2016 · 5 min read

Simple Reinforcement Learning in Tensorflow: Part 1 - Two-armed Bandit

Introduction

Reinforcement learning provides the capacity for us not only to teach an artificial agent how to act, but to allow it to learn through it's own interactions with an environment. By combining the complex representations that deep neural networks can learn with the goal-driven learning of an RL agent, computers have accomplished some amazing feats, like beating humans at over a dozen Atari games, and defeating the Go world champion.

Learning how to build these agents requires a bit of a change in thinking for anyone used to working in a supervised learning setting though. Gone is the ability to simply get the algorithm to pair certain stimuli with certain responses. Instead RL algorithms must enable the agent to learn the correct pairings itself through the use of *observations, rewards, and actions*. Since there is no longer a "True" correct action for an agent to take in any given circumstance that we can just tell it, things get a little tricky. In this post and those to follow, I will be walking through the creation and training of reinforcement learning agents. The agent and task will begin simple, so that the concepts are clear, and then work up to more complex task and environments.

Two-Armed Bandit

The simplest reinforcement learning problem is the n-armed bandit. Essentially, there are n-many slot machines, each with a different fixed payout probability. The goal is to discover the machine with the best payout, and maximize the returned reward by always choosing it. We are going to make it even simpler, by only having two possible slot machines to choose between. In fact, this problem is so simple that it is more of a precursor to real RL problems than one itself. Let me explain. Typical aspects of a task that make it an RL problem are the following:

Different actions yield different rewards. For example, when looking for treasure in a maze, going left may lead to the treasure, whereas going right may lead to a pit of snakes.

Rewards are delayed over time. This just means that even if going left in the above example is the right things to do, we may not know it till later in the maze.

Reward for an action is conditional on the state of the environment. Continuing the maze example, going left may be ideal at a certain fork in the path, but not at others.

The n-armed bandit is a nice starting place because we don't have to worry about aspects #2 and 3. All we need to focus on is learning which rewards we get for each of the possible actions, and ensuring we chose the optimal ones. In the context of RL lingo, this is called learning a *policy.* We are going to be using a method called policy gradients, where our simple neural network learns a policy for picking actions by adjusting it's weights through gradient descent using feedback from the environment. There is another approach to reinforcement learning where agents learn *value functions.* In those approaches, instead of learning the optimal action in a given state, the agent learns to predict how good a given state or action will be for the agent to be in. Both approaches allow agents to learn good behavior, but the *policy gradient* approach is a little more direct.

Policy Gradient

The simplest way to think of a Policy gradient network is one which produces explicit outputs. In the case of our bandit, we don't need to condition these outputs on any state. As such, our network will consist of just a set of weights, with each corresponding to each of the possible arms to pull in the bandit, and will represent how good our agent thinks it is to pull each arm. If we initialize these weights to 1, then our agent will be somewhat optimistic about each arm's potential reward.

To update our network, we will simply try an arm with an e-greedy policy (See Part 7 for more on action-selection strategies). This means that most of the time our agent will choose the action that corresponds to the largest expected value, but occasionally, with e probability, it will choose randomly. In this way, the agent can try out each of the different arms to continue to learn more about them. Once our agent has taken an action, it then receives a reward of either 1 or -1. With this reward, we can then make an update to our network using the policy loss equation:

```
Loss = -log(π)*A
```

A is advantage, and is an essential aspect of all reinforcement learning algorithms. Intuitively it corresponds to how much better an action was than some baseline. In future algorithms, we will develop more complex baselines to compare our rewards to, but for now we will assume that the baseline is 0, and it can be thought of as simply the reward we received for each action.

π is the policy. In this case, it corresponds to the chosen action's weight.

Intuitively, this loss function allows us to increase the weight for actions that yielded a positive reward, and decrease them for actions that yielded a negative reward. In this way the agent will be more or less likely to pick that action in the future. By taking actions, getting rewards, and updating our network in this circular manner, we will quickly converge to an agent that can solve our bandit problem! Don't take my word for it though. Try it out yourself.

Simple Reinforcement Learning in Tensorflow Part 1:

The Multi-armed bandit

This tutorial contains a simple example of how to build a policy-gradient based agent that can solve the multi-armed bandit problem. For more information, see this Medium post (https://medium.com/@awjuliani/super-simple-reinforcement-learning-tutorial-part-1-fd544fab149).

For more Reinforcement Learning algorithms, including DQN and Model-based learning in Tensorflow, see my

(Update 09/10/2016): I rewrote the iPython walkthrough for this tutorial today. The loss equation used previously was less intuitive than I would have liked. I have replaced it with a more standard and interpretable version that will definitely be more useful for those interested in applying policy gradient methods to more complex problems.)

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)*