Policy Gradients

Pawan Jayakumar

May 2019

1 Introduction

1.1 What is Reinforcement Learning?

Recall that the goal of reinforcement learning is to train an agent to maximize the return (the cumulative sum of discounted future rewards) it receives from the environment. There are many methods to approach this rather general problem. One method which we will explore in depth today is to create a policy that maps states to actions. In any given state, the policy returns the action that will maximize the future rewards received by the agent. Note that some algorithms involve creating a model of the environment. Sometimes this is beneficial if the environment is well known such as a chess board, but in other cases its impractical to create a model such as when the environment is really large or variable.

1.2 How to optimize the policy

There are two popular ways to optimize the policy. One way is to optimize the state value $V(s)$. Which state is the best to be in to receive the max return? The other is to optimize the action value $Q(s,a)$. Regardless of what state we are in, which action is most beneficial to maximize the reward? This is known as Q learning.

1.3 Deep RL

Notice that for complex environments there are an infinite amount of states and actions. It is impractical to have a table to store all the state/action values. This is where deep reinforcement learning comes in. Using neural networks, we can approximate the state/action value. This was a major breakthrough for RL and it is what allowed agents to be trained to beat Atari games such as pong.

1.4 What are policy gradients?

The policy function is just like any other mathematical function in that it can be optimized. Just as we did for neural networks, we can find the gradient of the reward function and use that to make incremental adjustments to our policy in order to maximize the return. This is known as gradient ascent.

2 Theory

2.1 The reward function

The policy function is a a parametric function of two variables, the state and the value. The function is parameterized with respect to theta. For learning purposes, think of this policy as some neural network where you input the state and it gives you the probability of choosing an action. Theta represents the weights and biases of the network. By computing the gradient of the reward function with respect to theta, we can find the best weights and biases for our neural network.

$$\pi_\theta(a|s) \tag{1}$$

We also parametrize the reward function to make it easier to compute the gradient. The parameterized reward function is

$$J(\theta) = \sum_{s\in\mathcal{S}} d^\pi(s)V^\pi(s) = \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s)Q^\pi(s,a) \tag{2}$$

This equation will seem very confusing at first because of the heavy notation, but the underlying principle is not as bad.

$d^\pi(s)$ is the stationary distribution of the Markov chain of the environment. If you took an agent and had it travel through the Markov chain (picking action a at state s and receiving reward r and going to state s' over and over again), at some point the agent will have found the optimal policy so the probability of it traveling to some state remains constant. At this point you can associate the probability that the agent will visit a certain state s.

Assuming the agent has found the optimal policy they will have also found the optimal value function $V(s)$. The total expected reward is the sum of the probability the agent will reach state s multiplied by the value of being in state s over all possible states.

$V(s)$ can be rewritten as the sum over all actions of the probability that the agent chooses action a multiplied by the Q value of that action. This way we have the reward function in terms of our policy function.

2.2 Policy Gradient Theorem

One thing you may have realized is that calculating $d^\pi(s)$ is very hard, in fact, there is actually no way to compute $d^\pi(s)$ unless you had the optimal policy. It goes without saying that this is all pointless if we already have the optimal policy.

Fortunately we can rewrite the gradient of the reward function without using $d^\pi(s)$ using the policy gradient theorem. Vanilla policy gradient is defined as:

$$E_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)] \tag{3}$$

To understand where this equation comes from, look in the attached resources. It is too complicated to explain in 20 minutes.

3 Implemented Algorithms

The biggest advantage these algorithms have over vanilla policy gradient is that they reduce the variance of the updates. This helps train the model faster.

3.1 Monte-Carlo Policy Gradient

This variation of policy gradient allows you to use monte-carlo methods to calculate the gradient. It is essentially forward propogation. You run the agent, which generates state,action,reward tuples. Using these tuples, you can estimate the return and update the weights using the following equation:

$$\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \ln \pi_\theta(A_t|S_t) \tag{4}$$

where G is the estimated return, alpha is the learning rate and γ^t is the discount factor

3.2 Actor Critic

This is more complicated because it uses two models that help each other learn. The critic updates the value function (either $Q_w(a)$ or $V_w(s)$). The actor updates the weights for the policy according to what the critic says. Here is the general algorithm:
Create two neural networks with random weights, θ and w and take an action a dictated by the policy at a random state s.
For each training step:
1.Get the reward r_t and next state s' from taking action a.
2.Take another action a' according to the policy
3.Update the policy with the following equation:

$$\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta \ln \pi_\theta(a|s) \tag{5}$$

4. Compute the TD error for action-value function:

$$\theta \leftarrow \theta + \alpha_\theta Q_w(s,a)\nabla_\theta \ln \pi_\theta(a|s) \tag{6}$$

5. Use this to update the weights of critic.

$$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a) \tag{7}$$

6. update $a \leftarrow a' s \leftarrow s'$

4 Resources

RL has a very high barrier of entry compared to traditional supervised machine learning because you actually have to know what is going on, there is no keras method for creating an agent. Implementing these algorithms is straightforward once you understand them but just one mistake and it is a nightmare to debug. Here are some resources to help you learn more about RL in general.

https://lilianweng.github.io/lil-log/
https://spinningup.openai.com/en/latest/index.html