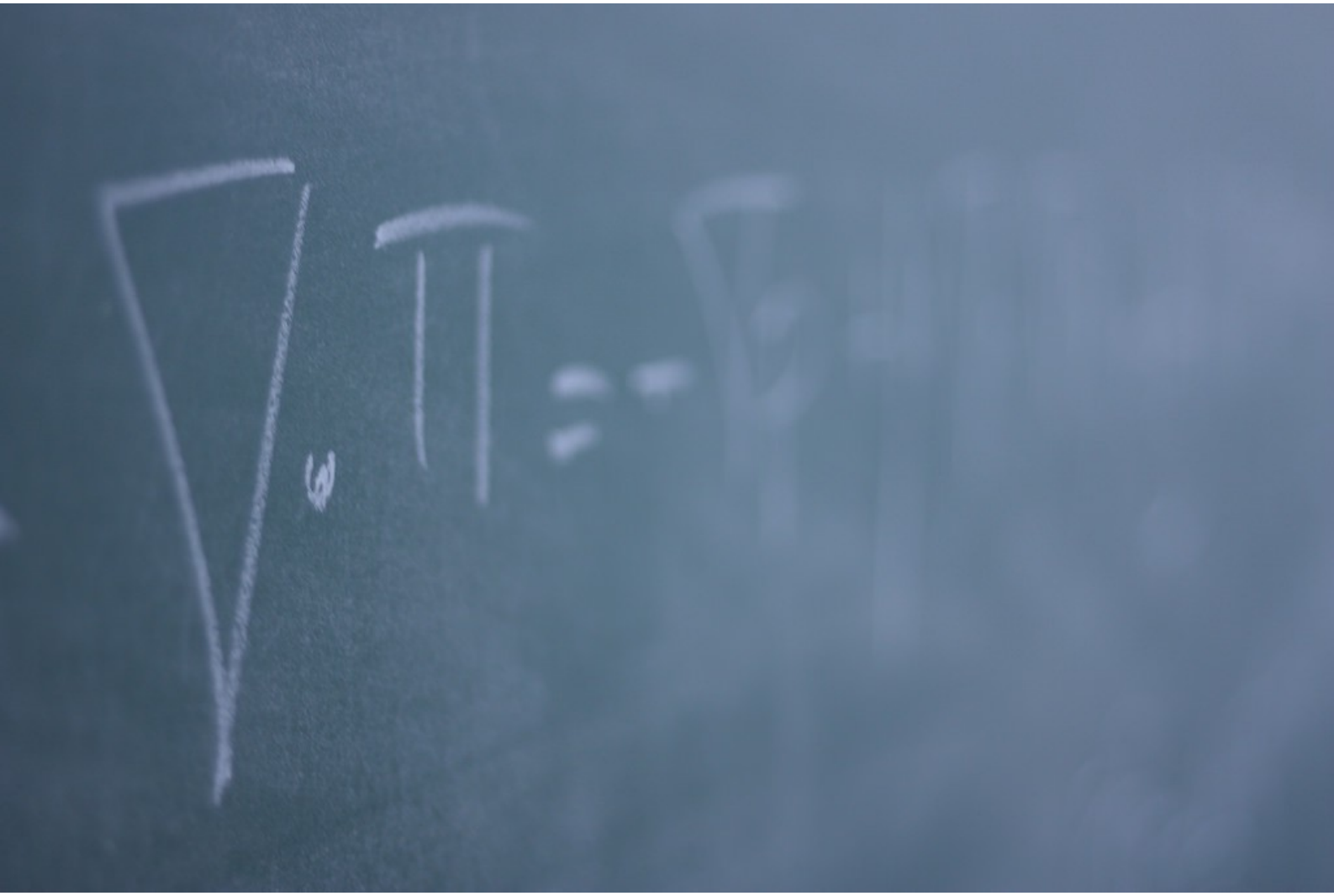
An Intuitive Explanation of Policy Gradient

Adrien Lucas Ecoffet Follow

Oct 5, 2018 · 13 min read



Source: https://hiveminer.com/User/Julian%20Veron

This is part 1 of a series of tutorials which I expect to have 2 or 3 parts. The next part will be on A2C and, time providing, I hope to complete a part on various forms of off-policy policy gradients.

The notebooks for these posts can be found in this git repo.

Introduction

The Problem with Policy Gradient

Some of today's most successful reinforcement learning algorithms, from A3C to TRPO to PPO belong to the **policy gradient** family of algorithm, and often more specifically to the **actor-critic** family. Clearly as an RL enthusiast, you owe it to yourself to have a good understanding of the policy gradient method, which is why so many tutorials out there attempt to describe them.

Yet, if you've ever tried to follow one of these tutorials, you were probably faced with an equation along the following lines:

or perhaps the related update rule:

or perhaps even the loss function:

Most likely this was accompanied by a very handwavy explanation, a ton of complicated math, or no explanation at all.

Indeed, some of my favorite tutorials on Reinforcement Learning are guilty of this. Jaromiru's Let's make an A3C states:

> The second term inside the expectation, $\nabla\theta\log\pi(a|s)$, tells us a direction in which logged probability of taking action a in state s rises. Simply said, how to make this action in this context more probable.

While Arthur Juliani states in Simple Reinforcement Learning with Tensorflow:

> 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.

Both of these explanations are unsatisfactory. Specifically, they completely fail to explain the presence of the mysterious log function in these formulas. If you only remembered

their intuitive explanations, you would not be able to write down the loss function yourself. This is not OK.

Amazingly, there actually **is** a perfectly intuitive explanation of this formula! However, as far as I know the only clear statement of this intuition is hidden in Chapter 13, section 3 of Sutton & Barto's Reinforcement Learning: An Introduction.

Once I describe it to you in painstaking details, you will be able to account for the presence of the loglog function and you should be able to write down the formula from first principles. Indeed, you will also be able to come up with your own variants policy gradient!

In this post, I will assume that you already have some basic notions of Reinforcement Learning, and in particular that you are familiar with Q-Learning and, ideally, DQNs.

If you aren't yet familiar with Q-Learning, I suggest you read at least the introduction of one of the many great DQN tutorials out there. Some suggestions:

- Beat Atari with Deep Reinforcement Learning by yours truly
- Let's Make a DQN by Jaromiru
- Simple Reinforcement Learning with Tensorflow by Arthur Juliani

Motivating Policy Gradient

Before going any further, let's learn about some of the advantages of policy gradient methods over Q-Learning, as extra motivation to try to genuinely understand policy gradient:

- The policy implied by Q-Learning is deterministic. This means that Q-Learning can't learn stochastic policies, which can be useful in some environments. It also means that we need to create our own exploration strategy since following the policy will not perform any exploration. We usually do this with $\epsilon\epsilon$-greedy exploration, which can be quite inefficient.
- There is no straightforward way to handle continuous actions in Q-Learning. In policy gradient, handling continous actions is relatively easy.
- As its name implies, in policy gradient we are following gradients with respect to the policy itself, which means we are constantly improving the policy. By contrast, in Q-Learning we are improving our estimates of the values of different actions, which only implicitely improves the policy. You would think that improving the policy directly would be more efficient, and indeed it very often is.

In general, policy gradient methods have very often beaten value-based methods such as DQNs on modern tasks such as playing Atari games.

Explaining Policy Gradient

Simple Policy Gradient Ascent

Although this article aims to provide intuition on policy gradients and thus aims to not be too mathematical, our goal is still to explain an equation, and so we need to use some mathematical notation, which I will introduce as needed throughout the article. Let's start with our first fundamental pieces of notation:

The letter $\pi\pi$ will symbolize a policy. Let's call $\pi\theta(a|s)$ the probability of taking action a in state s. θ represents the parameters of our policy (the weights of our neural network).

Our goal is to update θ to values that make $\pi\theta$ the optimal policy. Because θ will change, we will use the notation θt to denote θ at iteration t. We want to find out the update rule that takes use from θt to $\theta t+1$ in a way that we eventually reach the optimal policy.

Typically, for a discrete action space, $\pi\theta$ would be a neural network with a **softmax** output unit, so that the output can be thought of as the probability of taking each action.

Clearly, if action $a*$ is the optimal action, we want $\pi\theta(a*|s)$ to be as close to 1 as possible.

For this we can simply perform gradient *ascent* on $\pi\theta(a*|s)$, so at each iteration we update θ in the following way:

We can view the gradient $\nabla\pi\theta t(a*|s)$ as being "the direction in which to move θt so as to *increase* the value of $\pi\theta t(a*|s)$ the fastest". Note that we are indeed using gradient *ascent* since we want to *increase* a value, not *decrease* it as is usual in deep learning.

Thus one way to view this update is that we keep "pushing" towards more of action a* in our policy, which is indeed what we want.

In the example below, we know that the first action is the best action, so we run gradient ascent on it. For this running example we will assume a single state s so that the evolution of the policy is easier to plot. We will generalize to multiple states later in this series when we introduce A2C.

The gif above shows the result of our algorithm. The height of each bar is the probability of taking each action as our algorithm runs, and the arrow shows the gradient that we are following each iteration. In this case, we are only applying the gradient on the first action, which happens to have the largest value (10), which is why it is the only one that increases at the expense of others.

Weighing the Gradients

Of course, in practice, we won't know which action is best… After all that's what we're trying to figure out in the first place!

To get back to the metaphor of "pushing", if we don't know which action is optimal, we might "push" on suboptimal actions and our policy will never converge.

One solution would be to "push" on actions in a way that is *proportional to our guess of the value* of these actions. These guesses can be highly approximate, but as long as they are somewhat grounded in reality, more overall pushing will happen on the optimal action $a*$. This way it is guaranteed that eventually our policy will converge to $a*=1$!

We will call our guess of the value of action a in state s $\hat{Q}$ (s,a). Indeed, this is very similar to the Q function that we know from Q-Learning, though there is a subtle and

important difference that will make it easier to learn and that we will learn about later. For now, let's just assume that this Q function is a given.

We get the following gradient ascent update, that we can now apply to each action in turn instead of just to the optimal action:

Let's weigh our updates to the simulated policy gradient using a noisy version of the Q values of the different actions (10 for action a, 5 for action b, 2.5 for action c), and this time, we will randomly update each of the different possible actions, since we assume that we don't know which one is best. As we see, the first action does still end up winning, even though our estimates of the values (as shown by the lengths of the arrows) varies significantly across iterations.

On-Policy Correction

Of course, in practice, our agent is not going to choose actions uniformly at random, which is what we implicitly assumed so far. Rather, we are going to follow the very policy

$\pi\theta$ that we are trying to train! This is called training **on-policy**. There are two reasons why we might want to train on-policy:

- We accumulate more rewards even as we train, which is something we might value in some contexts.
- It allows us to explore more promising areas of the state space by not exploring purely randomly but rather closer to our current guess of the optimal actions.

This creates a problem with our current training algorithm, however: although we are going to "push" stronger on the actions that have a better value, we are also going to "push" *more often* on whichever actions happen to have higher values of $\pi\theta$ to begin with (which could happen due to chance or bad initialization)! These actions might end up winning the race to the top in spite of being bad.

Let's illustrate this phenomenon by using the same update rule but sampling actions according to their probability instead of uniformly:

Here we can see that the third action, in spice of having a lower value than the other two, ends up winning because it starts out initialized much higher.

This means that we need to compensate for the fact that more probable actions are going to be taken more often. How do we do this? Simple: we *divide* our update by the

probability of the action. This way, if an action is 4x more likely to be taken than another, we will have 4x more gradient updates to it but each will be 4x smaller.

This gives us the following update rule:

Let's try it:

We now see that even though action 3 starts out with a significant advantage, action 1 eventually wins out because its updates are much larger when its probability is small. Exactly the behavior that we wanted!

Basic REINFORCE

And we are now done explaining the intuition behind policy gradient! All the rest of this post is simply filling out some details. This is a significant accomplishment and you should feel proud of yourself for understanding this far!

> *“But, wait a minute”, you say, “I thought you would tell us how to understand the mysterious update rule*

> *but our update rule looks completely different!”*

Ah yes, indeed our update rule looks different, but it is in fact essentially the same! As a reminder, the key point in our rule is

while the rule we are trying to explain contains

The next section will explain the difference between $\hat{A}$ and $\hat{Q}$, but suffice it to say for now that both work perfectly fine, but that using $\hat{A}$ is just an optimization.

The only difference remaining is thus between

and

In fact, these two expressions are equivalent! This is due to the chain rule and the fact that the derivative of log x is 1/x, as you might know from calculus. So we have in general:

So now you know the origin of the mysterious log function in the policy gradient update! So why do people use the form using the log function instead of the more intuitive one which divides by $\pi(s|a)\pi(s|a)$? I can see two reasons for this:

1. It obfuscates the intuition behind policy gradient, thus making you seem more impressive for understanding it.
2. It makes it possible to express the update as a loss function when doing gradient descent, as in the equation $L = -\hat{A}(s,a)\log\pi\theta(s|a)$ we saw at the beginning (bringing $\hat{A}(s,a)$inside the loss function is fine since it is a constant with respect to θ). This way you can use your favorite deep learning library to train your policy (which we will see how to do soon!).

At this point you now understand the basic form of the REINFORCE algorithm, which, as I am sure you expect, stands for "REward Increment = Nonnegative Factor × Offset Reinforcement ×Characteristic Eligibility"… (yes I am serious, see the original paper). REINFORCE is the fundamental policy gradient algorithm on which nearly all the advanced policy gradient algorithms you might have heard of are based.

The Advantage Function and Baselines

Now the final thing left to explain, as promised, is the difference between $\hat{Q}$ and $\hat{A}$. You should already be familiar with Q from Q-Learning: Q(s,a) is the value (cumulative discounted reward, to be exact) obtained by taking action a in state s, and then following our policy π thereafter.

Note that it may be the case that following any action will give us a cumulative reward of, say, at least 100, but that some actions will be better than others, so that Q might be equal to, say, 101 for action a, 102 for action b and 100 for action c. As you can guess, this means that almost all of the weight of our update will tell us nothing about whether the current action is better, which is problematic.

You may also be familiar with the V(s) function, which simply gives us the value of following the policy starting in state s and all the way after that. In the example above, V(s) will be larger than 100 since all the Q(s, a) are larger than 100.

By subtracting V(s) from Q(s, a), we get the *advantage function* A(s, a). This function tells us how much better or worse taking action a in state is is compared to acting according to the policy. In the example above, it will subtract the extra 100 from the Q values of all the action, providing more signal to noise ratio.

In this tutorial I always wrote $\hat{Q}$ or $\hat{A}$, with a "hat" on top of Q and A to emphasize that we are not using the "real" Q or A but rather an **estimate** of them. How to get those estimates will be the subject of the next post in the series.

As it turns out, REINFORCE will still work perfectly fine if we subtract **any** function from $\hat{Q}(s,a)$ as long as that function does not depend on the action. This makes sense based on our previous intuition of course, since the only thing that matters is that we "push" *harder* on the actions that have a higher Q value, and subtracting any value that doesn't depend on the action will preserve the ranking of how hard we push on the various actions.

This means that using the $\hat{A}$ function instead of the $\hat{Q}$ function is perfectly allowable. In fact, it is widely recommended for the reasons described above and also because it is supposed to reduce the variance of the gradient updates. Note that $\hat{A}$ is not known for sure to be the best function to use instead of $\hat{Q}$, but in practice it generally works quite well.

Let's see what using $\hat{A}$ does in our simulation:

Output:

```
Running using A function
Done in 48 iterations
Variance of A gradients: 31.384320255389287
Running using Q function
Done in 30 iterations
Variance of Q gradients: 31.813126235495847
```

Unfortunately, the impact is not dramatic here: the variance does end up slightly lower, but the time to converge is longer. That said, I am comparing the same learning rate between the two, and using the advantage function should let you use a higher learning rate without diverging. Further, this is a toy example and the benefits of using the advantage function has been widely attested in practice on larger problems. Sadly, this is the one instance in this tutorial in which you'll have to take my word for it that something works well.

Conclusion

You now have a full intuitive understanding of basic policy gradients. However, you might be wondering how to use these in practice: how do you represent a policy with a neural network? How do you get your Â estimate in the first place? What other tricks are used in practice to make this work?

We will learn about all of this soon in part 2, which will explain the A2C algorithm.

)

Machine Learning | Reinforcement Learning | Deep Learning | Artificial Intelligence | Towards Data Science