Member-only story

Policy Gradients In Reinforcement Learning Explained

Learn all about policy gradient algorithms based on likelihood ratios (REINFORCE): the intuition, the derivation, the 'log trick', and update rules for Gaussian and softmax policies.

Wouter van Heeswijk, PhD · Following

Published in Towards Data Science · 15 min read · Apr 9, 2022

Photo by Scott Webb on Unsplash

When I first studied policy gradient algorithms, I did not find them particularly easy to fathom. Intuitively they seemed straightforward enough — **sample actions, observe rewards, tweak the policy** — but after the initial idea followed many lengthy derivations, calculus tricks I had long forgotten, and an overwhelming amount of notation. At a certain point, it just became a blur of probability distributions and gradients.

In this article, I try to explain the concept step by step, including key thought processes and mathematical operations. Admittedly, it's a bit of a long read and requires a certain preliminary knowledge on Reinforcement Learning (RL), but hopefully it sheds some light on the idea behind policy gradients.

The focus is on **likelihood ratio policy gradients**, which is the foundation of classical algorithms such as REINFORCE/vanilla policy gradient.

Given the length, let's structure this article up front:

1. *Value approximation: learning deterministic policies*
2. *Policy approximation methods: Moving to stochastic policies*
3. *Establishing the objective function*
4. *Defining trajectory probabilities*
5. *Introducing the policy gradient*
6. *Deriving the policy gradient*
7. *Gradient of the log probability function*
8. *Approximating the gradient*
9. *Defining the update rule*
10. *Examples: Softmax and Gaussian policies*
11. *Loss functions and automated gradient calculations*
12. *Algorithmic implementation (REINFORCE)*

I. Value approximation: learning deterministic policies

The objective of RL is to learn a good decision-making policy π that maximizes rewards over time. Although the notion of a (deterministic) policy π might seem a bit abstract at first, it is simply a function that returns an action a based on the problem state s, $\pi : s \rightarrow a$.

If you have some experience with RL, you probably started with **value approximation**. This class of RL stays close to the Dynamic Programming paradigm, aiming to *approximate* value functions — Q-values that reflect downstream rewards — rather than recursively solving Bellman equations to optimality.

For value approximation methods, deterministic policies works just fine. Typically, we select the best action (given our Q-values) with probability 1-ϵ and a random action with probability ϵ, allowing some exploration of new actions. We compare r(t)+Q_t+1 to Q_t, and use the observed errors to improve the value functions. The entire concept stays quite close to Bellman's optimality conditions.

Want to read more about the various classes of RL? Check out this article:

The Four Policy Classes of Reinforcement Learning

towardsdatascience.com

II. Policy approximation methods: Moving to stochastic policies

In policy approximation methods, we omit the notion of learning value functions, instead **tuning the policy directly**. We **parameterize the policy** with a set of parameters θ — these could be neural network weights, for instance — and tweak θ to improve the policy π_θ.

This sounds reasonable enough, but how do we evaluate the quality of a given policy? How do we update θ? Without the ability to contrast the

corresponding performance to something else, there is no way to tell. Like the ϵ-greedy approach for value approximation, we need some **exploration mechanism**.

There's a number of policy approximation methods (e.g., genetic algorithms, hill climbing), but **policy gradients** are used by far the most due to their efficiency. There are various forms of policy gradient algorithms (e.g., *finite difference methods* add small perturbations to θ and measure the differences), but this article focuses exclusively on **likelihood ratio policy gradients**.

The core idea is to replace the deterministic policy $\pi:s\rightarrow a$ with a parameterized probability distribution $\pi_\theta(a|s) = P(a|s; \theta)$. Instead of returning a single action, we **sample actions from a probability distribution** tuned by θ.

A stochastic policy might seem inconvenient, but it provides the foundation to optimize the policy. We'll get into the mathematics soon, but let's start with a hand-wavy intuition. The various policy samples enable us to contrast rewards associated with certain actions. We use these samples to alter θ, **increasing probabilities of obtaining high rewards**. This is the essence of likelihood ratio gradient policies.

III. Establishing the objective function

When moving through a sequential decision-making process, we follow a state-action *trajectory* $\tau=(s_1,a_1,\ldots,s_T,a_T)$. By sampling actions, the policy influences the probability with which we observe each possible **sequence of states and actions over the time horizon**. Each trajectory has a corresponding probability of $P(\tau)$ and a cumulative reward $R(\tau)=\sum\gamma^t R_t$ (sequence of rewards R_t discounted by γ).

> *For simplicity (not a necessity), we assume a finite set of trajectories, such that we can sum over probabilities rather than integrate.*

With these ingredients, we can formalize our **objective**, which is to maximize the expected reward over time. Here, τ~π_θ formalizes the distribution of trajectories under the prevailing policy. Equivalently, we may sum over all trajectory probabilities and multiply them with corresponding rewards.

The **objective function** J(θ) is as follows:

$$J(\theta) = E_{\tau \sim \pi_\theta} R(\tau) = \sum_{\tau} P(\tau;\theta) R(\tau)$$

The objective function for the likelihood ratio policy gradient method. As sampling provides an unbiased estimate of the expectation, we can approximate it using simulation.

The corresponding **maximization problem** is denoted by:

$$\max_{\theta} J(\theta) = \max_{\theta} \sum_{\tau} P(\tau;\theta) R(\tau)$$

Maximization problem for the policy approximation. By adjusting θ, we aim to increase the probability of following trajectories τ that yield high rewards.

IV. Defining trajectory probabilities

From the maximization problem, it is clear that adjusting θ impacts the trajectory probabilities. The next question is: **how to compute the probabilities P(τ;θ)?** Recall that this objective term is influenced by the

policy π_θ. By increasing the probabilities of high reward trajectories, we improve our expected reward.

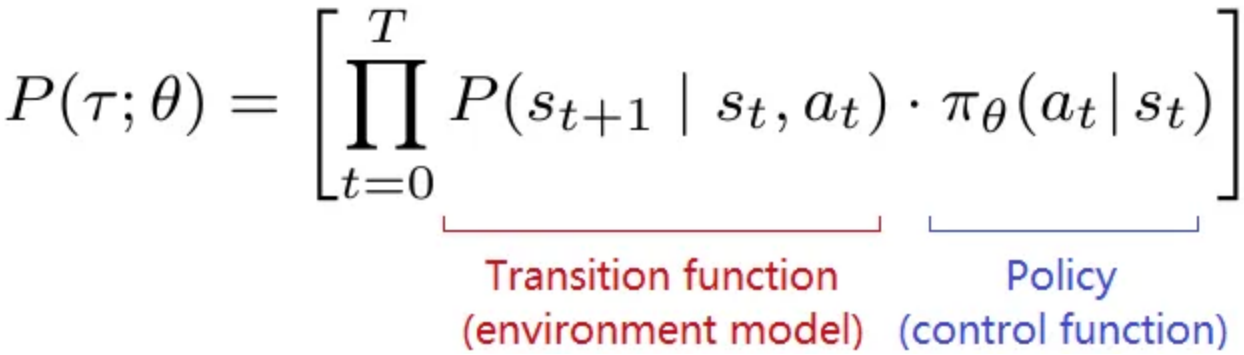
> *If you want more theoretical background on this part, it is worth reading up on* ***likelihood ratio methods, score functions*** *and* ***importance sampling****. For the purpose of this article, the current level of detail suffices.*

In a fully deterministic environment, we could compute the trajectory yielded by each policy π_θ and find the policy yielding the highest cumulative reward. However, most RL problems are not purely deterministic, but have a (substantial) stochastic component. As such, trajectory probabilities are *influenced by* — but not completely *determined by* — the policy. We compute the **probability of a certain reward trajectory** occurring given our policy.

To summarize, we deal with two probability distributions:

- **The policy itself is a probability distribution** $\pi_\theta(a|s) = P(a|s; \theta)$. The policy prescribes the probability with which each action is chosen in a given state, and depends on the parameter settings θ.
- A **transition probability distribution** $P(s_{t+1}|s_t,a_t)$ describes state transitions in the environment. Note that this probability distribution is partially influenced by π_θ (action selection) and partially by exogeneous information.

With this information, let's try to compute the probability of trajectory τ occurring under policy $\pi_\theta(a|s)$. Each time step **multiplies the probability** of sampling action $\pi_\theta(a_t|s_t)$ with transition probability $P(s_{t+1}|s_t,a_t)$. Subsequently, we multiply these probabilities for each time step to find the probability for the full trajectory materializing:



The trajectory probability is the product of action probabilities (dictated by policy π_θ) and state transition probabilities (governed by transition function P(s_t+1)).

The troublesome part is the transition function P(s_t+1). This is the model of the environment, which is complex to work with at best and completely unknown at worst.

An additional downside of this model is that it is a **product of probabilities.** For lengthy time horizons and small probabilities, trajectory probabilities get extremely small. As computer languages only offer finite precision floats, this causes numerical instability.

Let's worry about those things later, and first revisit the function we actually aim to optimize.

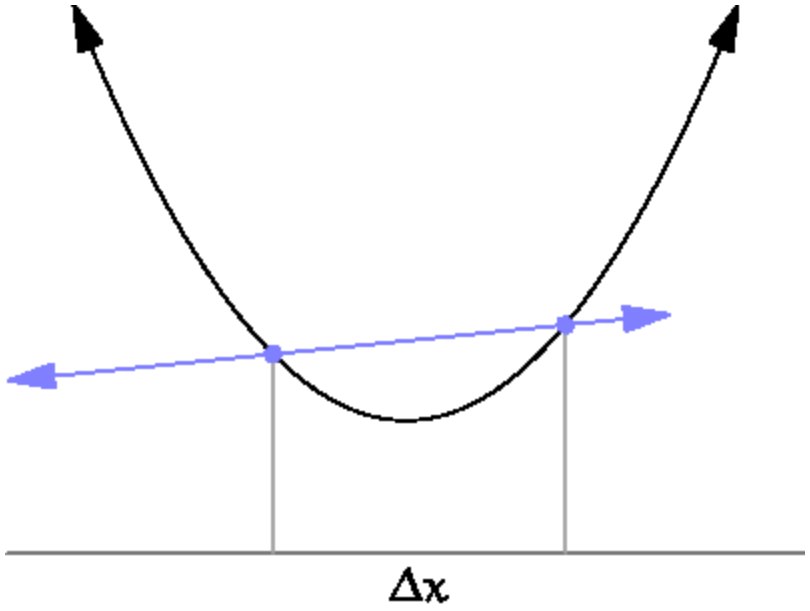
V. Introducing the policy gradient

As established, we seek to maximize our expected reward J(θ). How can optimize this function, e.g., identify the parameters θ that maximize the objective function?

Well, we have made a few helpful observations by now. By adopting a *stochastic* policy, we sample a variety of actions that yield different trajectories τ, enabling us to see which actions yield the best rewards. In

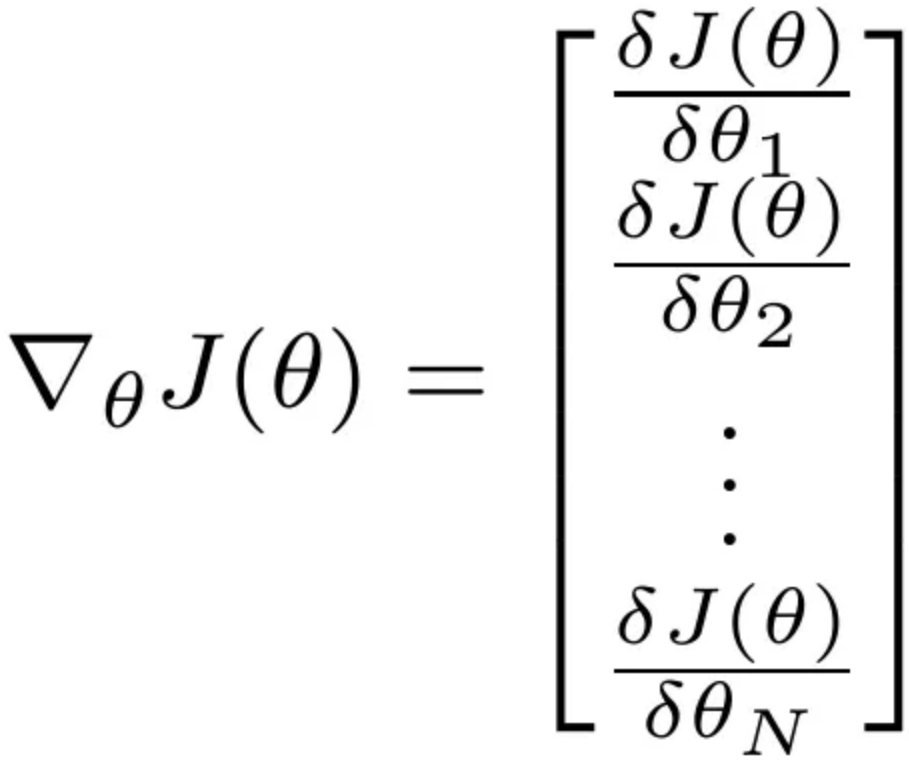
other terms, **the sampled observations provide an update direction** for the policy parameters θ.

Time to concretize this 'direction'. In high school, we learned how to take the **derivative** δf/δx of a function f(x) with respect to x to move towards its maximum (or minimum). If the derivative is large, a steep slope is implied — we are likely far from the optimum. If it is 0, we landed at a local optimum. We could do the same for J(θ), taking the derivative of the objective function J with respect to θ.



An animation of derivative slopes using a tangent line [image from Wikipedia by Brnbrnz]

A **gradient** generalizes the notion of a derivative, simply being a **vector of partial derivatives**. As θ is typically a vector θ=[θ_1,θ_2,…θ_N], we see how a partial derivate can be computed for each parameter (e.g., δJ(θ)/δ θ_1):



$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\delta J(\theta)}{\delta \theta_1} \\ \frac{\delta J(\theta)}{\delta \theta_2} \\ \vdots \\ \frac{\delta J(\theta)}{\delta \theta_N} \end{bmatrix}$$

Gradient of the parameterized cumulative reward function J(θ). The gradient is a vector of partial derivatives for each parameter θ_n in the vector θ.

To compute the gradient, we must be able to **differentiate the function** J(θ). We saw that changing π_θ(a|s) impacts trajectory probabilities P(τ;θ). However, we have not resolved *how* to compute these trajectory probabilities; remember we might not even *know* the transition function P(s_t+1|s_t,a_t)!

Time for some *mathemagics*.

VI. Deriving the policy gradient

We need to retrieve an explicit gradient of the objective function. Let's go through it step by step. We start by taking the gradient of the expected rewards:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta} R(\tau)$$

Step 1: express as gradient of expected reward

As seen before, we can rewrite this to the **sum over all trajectory probabilities** multiplied by trajectory rewards:

$$= \nabla_\theta \sum_\tau P(\tau;\theta) R(\tau)$$

Step 2: express as gradient of probability-weighted reward trajectories

The *gradient of a sum* equals the **sum of gradients**, so we can move the gradient within the summation.

$$= \sum_\tau \nabla_\theta P(\tau;\theta) R(\tau)$$

Step 3: rewrite as sum of gradients

The next step requires some more attention. Let's review an identity that is crucial to the gradient policy theorem — the **log derivative trick**.

> ***The log derivative trick***

> *A common technique in mathematics is to multiply an expression by 1, which obviously does not change it. In this case, we multiply with* $P(\tau;\theta)/P(\tau;\theta)$*, such that the expression becomes:*

$$= \sum_{\tau} \frac{P(\tau;\theta)}{P(\tau;\theta)} \nabla_{\theta} P(\tau;\theta) R(\tau)$$

Step 4a: Multiplying the expression by $P(\tau;\theta)/P(\tau;\theta)$ (which equals 1)

> *which we can rearrange to:*

$$= \sum_{\tau} \frac{P(\tau;\theta) \nabla_{\theta} P(\tau;\theta)}{P(\tau;\theta)} R(\tau)$$

Step 4b: Rearranging the expression

> *and subsequently to:*

$$= \sum_{\tau} P(\tau;\theta) \frac{\nabla_{\theta} P(\tau;\theta)}{P(\tau;\theta)} R(\tau)$$

Step 4c: Rearranging the expression once more

> *Why bother? Well, it is because we can now apply the following identity:*

$$\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} = \nabla_\theta \log P(\tau;\theta)$$

The log identity. This crucial result — also known as the likelihood ratio or the score function — facilitates a critical transformation of the trajectory probability function.

> *This identity is the cornerstone of the entire theorem. In fact,* ***this result is the likelihood ratio****.*
>
> *I will refrain from explaining in more detail — there's already enough math in this article as it is — but the identity can be reconstructed using (i) the derivative of a logarithmic function and (ii) the chain rule.*

Using the log derivative trick, we rewrite the expression to

$$= \sum_\tau P(\tau;\theta) \nabla_\theta \log P(\tau;\theta) R(\tau)$$

Step 4: rewrite using the log derivative trick

This expression is good news, for multiple reasons! We will restrict ourselves to a single reason right now, and that is the term ∑_θ(P(τ;θ). Why is this good news? Well, before the log transformation we summed over the *gradients* of the probabilities, now we sum over the probabilities themselves. With this

result, we can **rewrite our gradient as an expectation.** The final step becomes

$$= \mathbb{E}_{\tau \sim \pi_\theta} \nabla_\theta \log P(\tau;\theta) R(\tau)$$

Step 5: rewrite as expectation

It is worth pausing a moment to assess what we achieved. We already knew we could write the *objective function* as an expectation, but now it turns out **we can also write the gradient itself as an expectation.** This insight is crucial in applying sampling-based methods such as RL.

Unfortunately, we still don't know how to deal with P(τ;θ), (i.e., how to actually compute the gradient) but we'll get there shortly.

VII. Gradient of the log probability function

Remember that we struggled with the trajectory probability function earlier, because it required an explicit model of the environment (the P(s_t+1|s_t,a_t) part)? It turns out we already resolved that using the log trick!

Let's write out the gradient part ∇_θ log P(τ;θ). First, we simply substitute the lengthier version of P(τ;θ) we identified before:

$$\nabla_\theta \log P(\tau;\theta) = \nabla_\theta \log \left[\prod_{t=0}^{T} P(s_{t+1} \mid s_t, a_t) \cdot \pi_\theta(a_t \mid s_t) \right]$$

Step 1: substitute trajectory probability for explicit expression

Instead of taking the logarithm of the entire product of probabilities, we can rewrite to *logarithmic* probabilities:

$$= \nabla_\theta \left[\sum_{t=0}^{T} \log P(s_{t+1} \mid s_t, a_t) + \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t) \right]$$

Step 2: Rewrite using logarithmic probabilities

A convenient property of logarithmic probabilities is that they are **additive rather than multiplicative.** For numerical stability, this is very pleasant (given the finite precision of floats). It also helps to prevent exploding/vanishing gradients that commonly plague RL algorithms.

However, the main result is that we effectively **decoupled the transition function** P(s_t+1) **from the policy** π_θ(a_t). As the gradient is taken with respect to θ and P(s_t+1|s_t,a_t) does not depend on θ, we simply strike it!

$$= \nabla_\theta \left[\sum_{t=0}^{T} \log P(s_{t+1} \mid s_t, a_t) + \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t) \right]$$

Step 3a: remove the state transition function. As it is separate from the second term and does not depend on θ, it has no influence on the gradient.

The resulting outcome looks much cleaner, with our gradient depending only on the policy:

$$= \nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t \mid s_t)$$

Step 3b: gradient after removing the transition function

As a final step, we bring the gradient sign inside the summation (*'gradient of sum = sum of gradients'*), such that we can compute the gradient for each time step (e.g., for individual actions). The final result:

$$= \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

Step 4: expressing as sum of gradients

VIII. Approximating the gradient

We are nearly there. However, the computations so far concerned the true expectation, involving all possible trajectories. Typical RL problems are computationally intractable (otherwise we could just apply Dynamic Programming), so we need to work with a **sample of trajectories**.

In fact, we compute an **approximate gradient** based on a finite number of samples. Fortunately, we already established the gradient is also an expectation, which we can estimate using simulation. The expression looks like this:

$$\nabla_\theta J(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P\left(\tau^{(i)}; \theta\right) R\left(\tau^{(i)}\right)$$

Approximation of gradients, based on a finite number of trajectory samples m.

As you can see, this expression is **fully tractable.** We can sample the trajectories and corresponding rewards, even if we don't know the environment model. All we need is an explicitly defined policy that is differentiable w.r.t. θ.

For simplicity, I'll continue talking about *gradients* in the remainder of this article, but keep in mind we actually use *approximate gradients*.

IX. Defining the update rule

Once we compute the (approximate) gradient, how do we apply it to update our policy?

Based on sampled observations, we want to **update the policy parameters** gradually. For this, we define a learning rate $\alpha \in (0,1]$, indicating the weight placed on the computed gradient to update the existing parameter vector. The corresponding update rule looks like this:

$$\theta \leftarrow\!\shortmid \theta + \alpha \nabla_\theta J(\theta)$$

Policy gradient update rule. The new policy parameters are a weighted combination of the old parameters and the gradient over the objective function.

Often, the update rule is expressed by the delta with which θ should change: $\Delta\theta=\alpha\nabla_\theta J(\theta)$.

Remember that the gradient is simply a vector of partial derivatives. As such, the update rule provides a unique weight update for each element in θ. This way, updates align with the impact of individual features.

X. Examples: Softmax and Gaussian policies

We have arrived at an explicit update rule, but it is understandable if you feel a bit dazed by all the twists and turns.

Let's see if we can concretize our abstract results, providing the canonical policies for **discrete- and continuous action spaces.**

Some notation. Let $\phi(s,a)$ be a vector of basis functions (e.g., a vector of explanatory variables derived from the state action pair). As before, θ is the corresponding set of parameters, which may be interpreted as the weight per variable. We assume a simple linear function $\phi(s,a)^\top\cdot\theta$ — note we could easily replace this with, e.g., a neural network parameterized by θ.

For discrete actions, the **softmax policy** is used the most. It is defined by:

$$\pi_\theta(s,a)=\frac{e^{\phi(s,a)^\top\theta}}{\sum_{a'\in A}e^{\phi(s,a')^\top\theta}}$$

Softmax policy. This policy is commonly used for discrete action spaces

and its gradient (derivation here) is as follows:

$$\nabla_\theta \log \pi_\theta(a \mid s) = \phi(s,a) - \sum_{a' \in A} \phi(s,a')\pi_\theta(a \mid s)$$

Gradient of softmax policy

The result can be interpreted as the *observed* feature vector (from the sampled action) minus the *expected* feature vector over all actions. Thus, if the reward signal is high and the observed vector differs much from the expected vector, a strong incentive to update the probability of this action is provided.

For continuous action spaces, the **Gaussian policy** is most common. Here, we draw actions from a parameterized normal distribution. The distribution's mean is represented by μ_θ=ϕ(s,a)^⊤·θ.

$$\pi_\theta(a \mid s) = \frac{1}{\sqrt{2\pi}\sigma_\theta} e^{-\frac{a-\mu_\theta}{2\sigma_\theta^2}}$$

Gaussian policy. This policy is commonly used for continuous action spaces

The corresponding gradient:

$$\nabla_\theta \log(\pi_\theta(a \mid s)) = \frac{(a-\mu_\theta)\phi(s)}{\sigma_\theta^2}$$

Gradient of Gaussian policy

Also here, it can be seen that actions far from the mean trigger a strong update signals in case of high rewards. As probabilities must sum to 1, increasing probabilities for certain trajectories implies reducing those of others. As such, adjusting the policy impacts the expected rewards.

XI. Loss functions and automated gradient calculations

Although the policy needs to be differentiable and gradients can be computed using calculus, manually computing partial derivatives is rather cumbersome. Especially when the policy is a deep neural network — where θ represents the network weights — we typically rely on **automated gradient calculations**.

With automated gradients, we simply define a **loss function** and let the computer solve out all the derivations. The loss function effectively represents the update signal. We add a minus sign (as training relies on gradient *descent* rather than -*ascent*) and define the canonical loss function as follows:

$$\mathcal{L}(a, s, v) = -\log(\pi_\theta(a \mid s))v$$

Loss function for policy gradient algorithms. Most implementations offer automated differentiation, such that gradients are computed for you.

XII. Algorithmic implementation (REINFORCE)

The information provided in this article explains the background to likelihood ratio policy gradient methods, such as Williams' classical

REINFORCE algorithm.

Let's put it all together:

Input: $\theta \leftarrow \mathbb{R}^{|\theta|}$ ▶ Initialize θ
for $n = 1$ **to** N **do**:
  $\tau \sim \pi_\theta$ ▶ Generate state-action trajectory with π_θ
  for $t = 1$ **to** T **do**
    $R(\tau \mid t) = R_t + R_{t+1} + \ldots + R_T$ ▶ Compute cum. reward
    $\theta \leftarrow \theta + \alpha R(\tau \mid t) \nabla_\theta \log(\pi_\theta(a \mid s)$ ▶ Update θ

REINFORCE algorithm, also known as vanilla policy gradient or the likelihood ratio policy gradient [image by author, based on Williams (1992)]

Although it took some mathematics to get here, the actual algorithm is concise and straightforward. All we need is sampled rewards and the gradients of our policy.

Python implementations (e.g., using policy networks defined using TensorFlow) do not require much code either — check out my articles below for examples on both the continuous and discrete variant.

A Minimal Working Example for Discrete Policy Gradients in TensorFlow 2.0

A multi-armed bandit example for training discrete actor networks. With the aid of the GradientTape functionality, the...

towardsdatascience.com

A Minimal Working Example for Continuous Policy Gradients in TensorFlow 2.0

A simple example for training Gaussian actor networks. Defining a custom loss function and applying the GradientTape...

towardsdatascience.com

Cliff-Walking Problem With The Discrete Policy Gradient Algorithm

A full implementation of the REINFORCE algorithm in Python. The steps are performed manually to illustrate the inner...

towardsdatascience.com

If you want to take things a notch further, also read my articles on natural gradients, TRPO and PPO, which form the foundation of contemporary policy gradient algorithms:

Natural Policy Gradients In Reinforcement Learning Explained

Traditional policy gradient methods are fundamentally flawed. Natural gradients converge quicker and better, forming...

towardsdatascience.com

Trust Region Policy Optimization (TRPO) Explained

The Reinforcement Learning algorithm TRPO builds upon natural policy gradient algorithms, ensuring updates remain...

towardsdatascience.com

Proximal Policy Optimization (PPO) Explained

The journey from REINFORCE to the go-to algorithm in continuous control

towardsdatascience.com

If you made it all the way through, congratulations! It takes some time to get a grip on policy gradient algorithm, but once mastered they open the door to hybrid methods (e.g., actor-critic methods) as well as more advanced methods such as Proximal Policy Optimization. As such, their understanding is essential for any RL practitioner.

Summary

- The expected reward under a given policy is defined by the probability of a state-action trajectory multiplied with the corresponding reward. **Likelihood ratio policy gradients** build onto this definition by increasing the probabilities of high-reward trajectories, deploying a stochastic policy parameterized by θ.
- We may not know the transition- and reward functions of the environment. However, after a **log transformation**, we work with additive (logarithmic) probabilities rather than multiplications of probabilities. This transformation decouples the policy from the (possibly unknow) state transition function.
- If we have a differentiable policy w.r.t. its parameterization θ, we can compute its **gradient**. As we may express this gradient as an expectation, we can approximate it using simulation. Typical RL implementations utilize loss functions, from which gradients are derived automatically.
- In algorithms such as **REINFORCE**, we sample transitions and rewards from the environment (using the stochastic policy), and multiply trajectory rewards with the gradient of the log policy to update parameters θ.

Further reading

REINFORCE algorithm

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3), 229–256.

Explanation of different policy gradient methods

Riedmiller, M., Peters, J., & Schaal, S. (2007, April). Evaluation of policy gradient methods and variants on the cart-pole benchmark. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning* (pp. 254–261). IEEE.

Description of finite difference-, likelihood ratio-, and natural policy gradients

http://www.scholarpedia.org/article/Policy_gradient_methods

Intro to policy gradients (Jonathan Hui)

https://jonathan-hui.medium.com/rl-policy-gradients-explained-9b13b688b146

Policy gradient derivation (Chris Yoon)

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

Explanation on the log derivative trick (David Meyer):

https://davidmeyer.github.io/ml/log_derivative_trick.pdf

Likelihood ratio policy gradients (David Meyer):

https://davidmeyer.github.io/ml/policy_gradient_methods_for_robotics.pdf

Proof of policy gradient theorem and a large number of policy gradients algorithms (Lilian Weng):

https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

Likelihood ratio gradient (Tim Vieira):

https://timvieira.github.io/blog/post/2019/04/20/the-likelihood-ratio-gradient/

Sergey Levine (Berkeley) lecture slides on policy gradients

http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_4_policy_gradient.pdf

David Silver (Deepmind) lecture slides on policy gradients

https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf

Deep RL Bootcamp Lecture 4A by Pieter Abeel: Policy Gradients

Berkeley lecture on policy gradients

RL Course by David Silver — Lecture 7: Policy Gradient Methods

DeepMind lecture on policy gradients

Policy Gradient | Reinforcement Learning | Derivation | Reinforce | Deep Dives

Written by Wouter van Heeswijk, PhD

Following

1.5K Followers · Writer for Towards Data Science

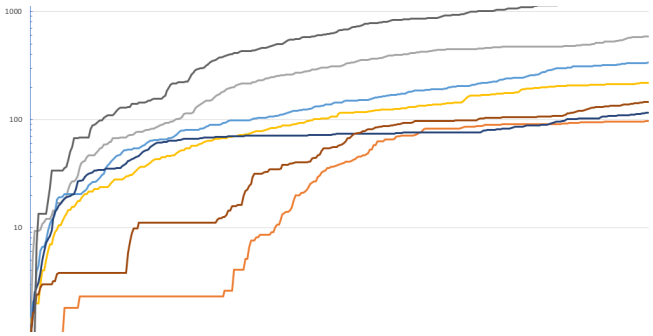
Assistant professor in Financial Engineering and Operations Research. Writing about reinforcement learning, optimization problems, and data science.

More from Wouter van Heeswijk, PhD and Towards Data Science

Wouter van Heeswijk, PhD in Towards Data Science

Five Ways To Handle Large Action Spaces in Reinforcement Learning

Action spaces, particularly in combinatorial optimization problems, may grow unwieldy i...



Pau Blasco i Roca in Towards Data Science

My Life Stats: I Tracked My Habits for a Year, and This Is What I...

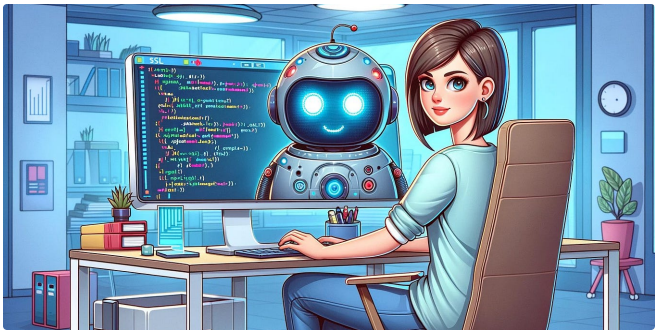
I measured the time I spent on my daily activities (studying, doing sports, socializing...

· 14 min read · Aug 18

36

12 min read · Nov 20

4.6K 82



Mariya Mansurova in Towards Data Science

LMQL—SQL for Language Models

Yet another tool that could help you with LLM applications

17 min read · Nov 27

683 10

Wouter van Heeswijk, PhD in Towards Data Science

Python's Map(), Filter(), and Reduce() Functions Explained

Demonstrating the building blocks of functional programming in Python with code...

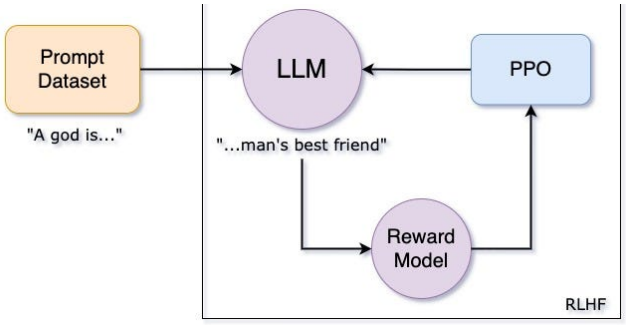
· 8 min read · Aug 10, 2022

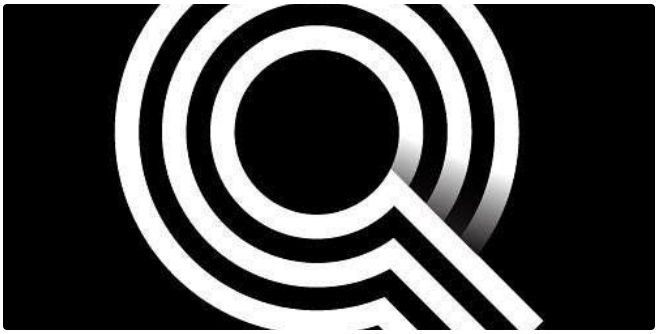
44

See all from Wouter van Heeswijk, PhD

See all from Towards Data Science

Recommended from Medium





Oleg Latypov

A Comprehensive Guide to Proximal Policy Optimization (PP...

OpenAI: We're releasing a new class of reinforcement learning algorithms, Proximal...

8 min read · Aug 27

12 1

Navneet Singh

Understanding Q-Learning: A Powerful Reinforcement Learning...

In the field of machine learning, specifically in the realm of reinforcement learning, Q-...

4 min read · Jul 10

8

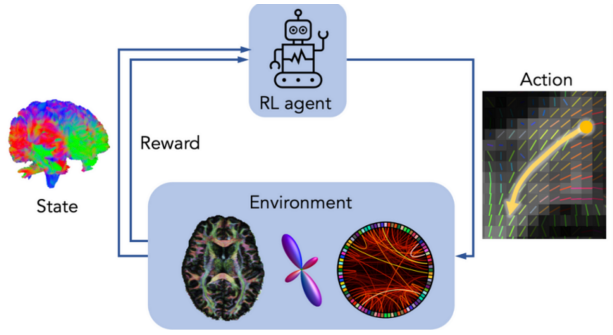
Lists

Stories to Help You Grow as a Software Developer
19 stories · 631 saves

Stories to Help You Level-Up at Work
19 stories · 368 saves

Natural Language Processing
986 stories · 475 saves

Staff Picks
535 stories · 535 saves



$$
\begin{aligned}
^{,\lambda)} &:= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \dots\right) \\
&= (1-\lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) \\
&= (1-\lambda)(\delta_t^V(1+\lambda+\lambda^2+\dots) + \gamma\delta_{t+1}^V(\lambda+\lambda^2+\lambda^3+\dots) \\
&\quad + \gamma^2\delta_{t+2}^V(\lambda^2+\lambda^3+\lambda^4+\dots) + \dots) \\
&= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right)\right. \\
&= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V
\end{aligned}
$$

Ankush k Singal in AI Artistry

Introduction to Reinforcement Learning Using Unity ML Agents-...

Overview Of Reinforcement Learning

5 min read · Aug 27

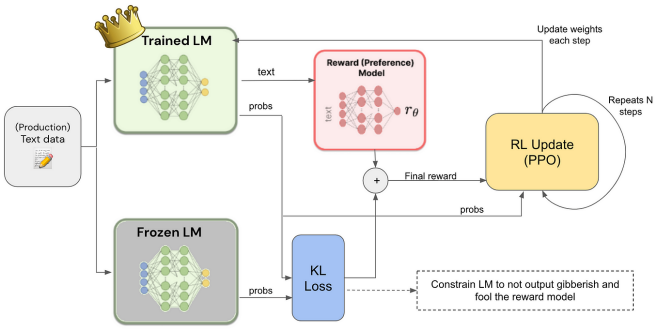
Zhirui Xia

Coding PPO from Scratch with PyTorch (Part 4/4)

Welcome to Part 4 of our series, where we will briefly discuss some of the most common...

8 min read · Aug 16

4



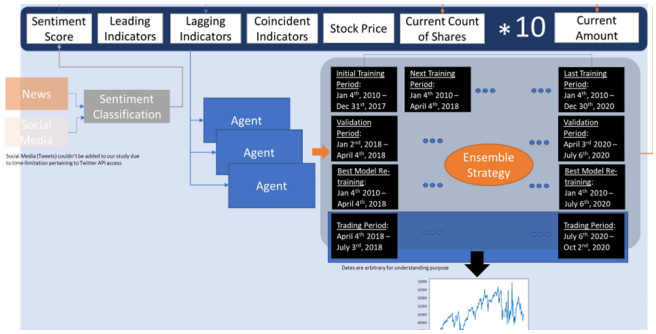
João Lages in Towards AI

Reinforcement Learning from Human Feedback (RLHF)

A Simplified Explanation

5 min read · Oct 31

118 1



Ayeshanasim

Portfolio Management Using Multi-Agent Reinforcement Learning

9 min read · Oct 14

58

See more recommendations