# Policy Gradients in a Nutshell

Sanyam Kapoor · Follow
Published in Towards Data Science
11 min read · Jun 3, 2018



. . .

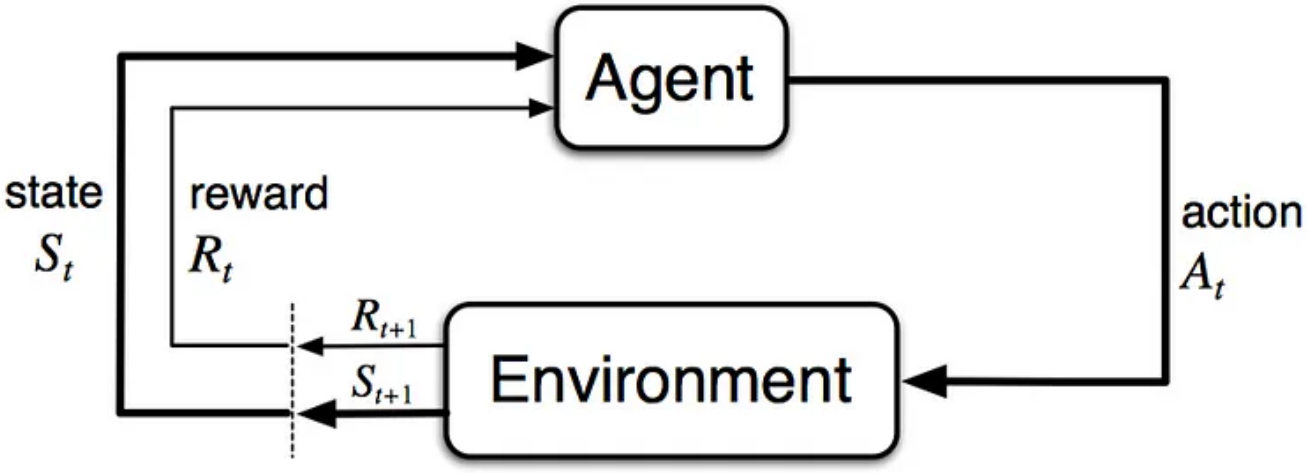
*This article aims to provide a concise yet comprehensive introduction to one of the most important class of control algorithms in Reinforcement Learning — Policy Gradients. I will discuss these algorithms in progression, arriving at well-known results from the ground up. It is aimed at readers with a reasonable background as for any other topic in Machine Learning. By the end, I hope that you'd be able to attack a vast amount of (if not all) Reinforcement Learning literature.*

. . .

## Introduction

Reinforcement Learning (RL) refers to both the learning problem and the sub-field of machine learning which has lately been in the news for great reasons. RL based systems have now beaten world champions of Go, helped operate datacenters better and mastered a wide variety of Atari games. The research community is seeing many more promising results. With enough motivation, let us now take a look at the Reinforcement Learning problem.

Reinforcement Learning is the most general description of the learning problem where the aim is to maximize a long-term objective. The system description consists of an *agent* which interacts with the *environment* via its actions at discrete time steps and receives a *reward*. This transitions the agent into a new *state*. A canonical agent-environment feedback loop is depicted by the figure below.




The Canonical Agent-Environment Feedback Loop

The Reinforcement Learning flavor of the learning problem is strikingly similar to how humans effectively behave — experience the world, accumulate knowledge and use the learnings to handle novel situations. Like many people, this attractive nature (although a harder formulation) of the problem is what excites me and hope it does you as well.

## Background and Definitions

A large amount of theory behind RL lies under the assumption of *The Reward Hypothesis* which in summary states that all goals and purposes of an agent can be explained by a single scalar called the *reward*. This is still subject to debate but has been fairly hard to disprove yet. More formally, the reward hypothesis is given below

> ***The Reward Hypothesis****: That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

As an RL practitioner and researcher, one's job is to find the right set of rewards for a given problem known as *reward shaping*.

The *agent* must formally work through a theoretical framework known as a Markov Decision Process which consists of a decision (what action to take?) to be made at each state. This gives rise to a sequence of states, actions and rewards known as a *trajectory,*

$$S_0, A_0, R_1, S_1, A_1, R_2, \dots$$

and the objective is to maximize this set of rewards. More formally, we look at the Markov Decision Process framework.

> ***Markov Decision Process****: A (Discounted) Markov Decision Process (MDP) is a tuple (S,A,R,*p*,*γ*), such that*

$$p(s', r|s, a) = Pr\left[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\right]$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

> *where* S_t, S_(t+*1*) ∈ *S (state space),* A_(t+*1*) ∈ *A (action space),* R_(t+*1*), R_t ∈ *R (reward space),* p *defines the dynamics of the process and* G_t *is the discounted return.*

In simple words, an MDP defines the probability of transitioning into a new state, getting some reward given the current state and the execution of an action. This framework is mathematically pleasing because it is First-Order Markov. This is just a fancy way of saying that anything that happens next is dependent only on the present and not the past. It does not matter how one arrives at the current state as long as one does. Another important part of this framework is the discount factor *γ*. Summing these rewards over time with a varying degree of importance to the rewards from the future leads to a notion of discounted returns. As one might expect, a higher *γ* leads to higher sensitivity for rewards from the future. However, the extreme case of *γ*=0 doesn't consider rewards from the future at all.

The dynamics of the environment *p* are outside the control of the agent. To internalize this, imagine standing on a field in a windy environment and taking a step in one of the four directions at each second. The winds are so strong, that it is hard for you to move in a direction perfectly aligned with north, east, west or south. This probability of

landing in a new state at the next second is given by the dynamics $p$ of the windy field. It is certainly not in your (agent's) control.

However, what if you somehow understand the dynamics of the environment and move in a direction other than north, east, west or south. This *policy* is what the agent controls. When an agent follows a policy $\pi$, it generates the sequence of states, actions and rewards called the *trajectory*.

> *Policy: A policy is defined as the probability distribution of actions given a state*

$$\pi(A_t = a | S_t = s)$$
$$\forall A_t \in \mathcal{A}(s), S_t \in \mathcal{S}$$

With all these definitions in mind, let us see how the RL problem looks like formally.

## Policy Gradients

The objective of a Reinforcement Learning agent is to maximize the "expected" reward when following a policy $\pi$. Like any Machine Learning setup, we define a set of parameters $\theta$ (e.g. the coefficients of a complex polynomial or the weights and biases of units in a neural network) to parametrize this policy — $\pi\_\theta$ (also written a $\pi$ for brevity). If we represent the total reward for a given trajectory $\tau$ as $r(\tau)$, we arrive at the following definition.

> ***Reinforcement Learning Objective**: Maximize the "expected" reward following a parametrized policy*

$$J(\theta) = \mathbb{E}_\pi \left[ r(\tau) \right]$$

All finite MDPs have at least one optimal policy (which can give the maximum reward) and among all the optimal policies at least one is stationary and deterministic.

Like any other Machine Learning problem, if we can find the parameters $\theta\star$ which maximize $J$, we will have solved the task. A standard approach to solving this maximization problem in Machine Learning Literature is to use Gradient Ascent (or

Descent). In gradient ascent, we keep stepping through the parameters using the following update rule

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

Here comes the challenge, how do we find the gradient of the objective above which contains the expectation. Integrals are always bad in a computational setting. We need to find a way around them. First step is to reformulate the gradient starting with the expansion of expectation (with a slight abuse of notation).

$$\begin{aligned}
\nabla \mathbb{E}_\pi \left[ r(\tau) \right] &= \nabla \int \pi(\tau) r(\tau) d\tau \\
&= \int \nabla \pi(\tau) r(\tau) d\tau \\
&= \int \pi(\tau) \nabla \log \pi(\tau) r(\tau) d\tau \\
\nabla \mathbb{E}_\pi \left[ r(\tau) \right] &= \mathbb{E}_\pi \left[ r(\tau) \nabla \log \pi(\tau) \right]
\end{aligned}$$

> ***The Policy Gradient Theorem****: The derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy* π_θ.

$$\nabla \mathbb{E}_{\pi_\theta} \left[ r(\tau) \right] = \mathbb{E}_{\pi_\theta} \left[ r(\tau) \nabla \log \pi_\theta(\tau) \right]$$

Now, let us expand the definition of $\pi\_\theta(\tau)$.

$$\pi_\theta(\tau) = \mathcal{P}(s_0) \Pi_{t=1}^{T} \pi_\theta(a_t | s_t) p(s_{t+1}, r_{t+1} | s_t, a_t)$$

To understand this computation, let us break it down — P represents the ergodic distribution of starting in some state *s*_0. From then onwards, we apply the product rule of probability because each new action probability is independent of the previous one (remember Markov?). At each step, we take some action using the policy $\pi\_\theta$ and the environment dynamics *p* decide which new state to transition into. Those are

multiplied over $T$ time steps representing the length of the trajectory. Equivalently, taking the log, we have

$$\log \pi_\theta(\tau) = \log \mathcal{P}(s_0) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \sum_{t=1}^{T} \log p(s_{t+1}, r_{t+1}|s_t, a_t)$$

$$\nabla \log \pi_\theta(\tau) = \sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)$$

$$\implies \nabla \mathbb{E}_{\pi_\theta}\left[r(\tau)\right] = \mathbb{E}_{\pi_\theta}\left[r(\tau)\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)\right)\right]$$

This result is beautiful in its own right because this tells us, that we don't really need to know about the ergodic distribution of states P nor the environment dynamics $p$. This is crucial because for most practical purposes, it hard to model both these variables. Getting rid of them, is certainly good progress. As a result, all algorithms that use this result are known as "*Model-Free Algorithms*" because we don't "model" the environment.

The "expectation" (or equivalently an integral term) still lingers around. A simple but effective approach is to sample a large number of trajectories (I really mean LARGE!) and average them out. This is an approximation but an unbiased one, similar to approximating an integral over continuous space with a discrete set of points in the domain. This technique is formally known as Markov Chain Monte-Carlo (MCMC), widely used in Probabilistic Graphical Models and Bayesian Networks to approximate parametric probability distributions.

One term that remains untouched in our treatment above is the reward of the trajectory $r(\tau)$. Even though the gradient of the parametrized policy does not depend on the reward, this term adds a lot of variance in the MCMC sampling. Effectively, there are $T$ sources of variance with each $R\_t$ contributing. However, we can instead make use of the returns $G\_t$ because from the standpoint of optimizing the RL objective, rewards of the past don't contribute anything. Hence, if we replace $r(\tau)$ by the discounted return $G\_t$, we arrive at the classic algorithm Policy Gradient algorithm called *REINFORCE*. This doesn't totally alleviate the problem as we discuss further.

## REINFORCE (and Baseline)

To reiterate, the REINFORCE algorithm computes the policy gradient as

**REINFORCE Gradient**

$$\nabla \mathbb{E}_{\pi_\theta}\left[r(\tau)\right] = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} G_t \nabla \log \pi_\theta(a_t|s_t)\right)\right]$$

We still have not solved the problem of variance in the sampled trajectories. One way to realize the problem is to reimagine the RL objective defined above as *Likelihood Maximization* (Maximum Likelihood Estimate). In an MLE setting, it is well known that data overwhelms the prior — in simpler words, no matter how bad initial estimates are, in the limit of data, the model will converge to the true parameters. However, in a setting where the data samples are of high variance, stabilizing the model parameters can be notoriously hard. In our context, any erratic trajectory can cause a sub-optimal shift in the policy distribution. This problem is aggravated by the scale of rewards.

Consequently, we instead try to optimize for the difference in rewards by introducing another variable called baseline $b$. To keep the gradient estimate unbiased, the baseline independent of the policy parameters.

**REINFORCE with Baseline**

$$\nabla \mathbb{E}_{\pi_\theta}\left[r(\tau)\right] = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} (G_t - b) \nabla \log \pi_\theta(a_t|s_t)\right)\right]$$

To see why, we must show that the gradient remains unchanged with the additional term (with slight abuse of notation).

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} b\nabla \log \pi_\theta(a_t|s_t)\right)\right] &= \int \sum_{t=1}^{T} \pi_\theta(a_t|s_t) b\nabla \log \pi_\theta(a_t|s_t) d\tau \\
&= \int \sum_{t=1}^{T} \nabla b\pi_\theta(a_t|s_t) d\tau \\
&= \int \nabla b\pi_\theta(\tau) d\tau \\
&= b\nabla \int \pi_\theta(\tau) d\tau \\
&= b\nabla 1 \\
\mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} b\nabla \log \pi_\theta(a_t|s_t)\right)\right] &= 0
\end{aligned}$$

Using a baseline, in both theory and practice reduces the variance while keeping the gradient still unbiased. A good baseline would be to use the state-value current state.

> ***State Value**: State Value is defined as the expected returns given a state following the policy* π_θ .

$$V(s) = \mathbb{E}_{\pi_\theta}[G_t|S_t = s]$$

## Actor-Critic Methods

Finding a good baseline is another challenge in itself and computing it another. Instead, let us make approximate that as well using parameters $\omega$ to make $V^\omega(s)$. All algorithms where we bootstrap the gradient using learnable $V^\omega(s)$ are known as *Actor-Critic* Algorithms because this value function estimate behaves like a "*critic*" (good v/s bad values) to the "*actor*" (agent's policy). However this time, we have to compute gradients of both the actor and the critic.

> ***One-Step Bootstrapped Return**: A single step bootstrapped return takes the immediate reward and estimates the return by using a bootstrapped value-estimate of the next state in the trajectory.*

$$G_t \simeq R_{t+1} + \gamma V^\omega(S_{t+1})$$

**Actor-Critic Policy Gradient**

$$\nabla \mathbb{E}_{\pi_\theta}\left[r(\tau)\right] = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T}(R_{t+1} + \gamma V^{\omega}(S_{t+1}) - V^{\omega}(S_t))\nabla \log \pi_\theta(a_t|s_t)\right)\right]$$

It goes without being said that we also need to update the parameters $\omega$ of the critic. The objective there is generally taken to be the Mean Squared Loss (or a less harsh Huber Loss) and the parameters updated using Stochastic Gradient Descent.

**Critic's Objective**

$$J(\omega) = \frac{1}{2}\left(R_{t+1} + \gamma V^{\omega}(S_{t+1}) - V^{\omega}(S_t)\right)^2$$
$$\nabla J(\omega) = R_{t+1} + \gamma V^{\omega}(S_{t+1}) - V^{\omega}(S_t)$$

## Deterministic Policy Gradients

Often times, in robotics, a differentiable control policy is available but the actions are not stochastic. In such environments, it is hard to build a stochastic policy as previously seen. One approach is to inject noise into the controller. More over, with increasing dimensionality of the controller, the previously seen algorithms start performing worse. Owing to such scenarios, instead of learning a large number of probability distributions, let us directly learn a deterministic action for a given state. Hence, in its simplest form, a greedy maximization objective is what we need

**Deterministic Actions**

$$\mu^{k+1}(s) = \underset{a}{argmax}\, Q^{\mu^k}(s, a)$$

However, for most practical purposes, this maximization operation is computationally infeasible (as there is no other way than to search the entire space for a given action-value function). Instead, what we can aspire to do is, build a function *approximator* to approximate this *argmax* and therefore called the *Deterministic Policy Gradient* (DPG).

We sum this up with the following equations.

### DPG Objective

$$J(\theta) = \mathbb{E}_{s \sim \rho^{\mu_\theta}} \left[ r(s, \mu_\theta(s)) \right]$$

### Deterministic Policy Gradient

$$\nabla J(\theta) = \mathbb{E}_{s \sim \rho^{\mu_\theta}} \left[ \nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu_\theta}(s, a) \big|_{a = \mu_\theta(s)} \right]$$

It shouldn't be surprising enough anymore that this value turned out to another expectation which we can again estimate using MCMC sampling.

## Generic Reinforcement Learning Framework

We can now arrive at a generic algorithm to see where all the pieces we've learned fit together. All new algorithms are typically a variant of the algorithm given below, trying to attack one (or multiple steps of the problem).

```
Loop:
    Collect trajectories (transitions - (state, action, reward, next state, terminated flag))
    (Optionally) store trajectories in a replay buffer for sampling
    Loop:
        Sample a mini batch of transitions
        Compute Policy Gradient
        (Optionally) Compute Critic Gradient
        Update parameters
```

general_rl_algo hosted with ❤ by GitHub view raw

## Code

For the readers familiar with Python, these code snippets are meant to be a more tangible representation of the above theoretical ideas. These have been taken out of the learning loop of real code.

### Policy Gradients (Synchronous Actor-Critic)

```
# Compute Values and Probability Distribution
values, prob = self.ac_net(obs_tensor)

# Compute Policy Gradient (Log probability x Action value)
advantages = return_tensor - values
action_log_probs = prob.log().gather(1, action_tensor)
actor_loss = -(advantages.detach() * action_log_probs).mean()

# Compute L2 loss for values
critic_loss = advantages.pow(2).mean()

# Backward Pass
loss = actor_loss + critic_loss
loss.backward()
```

pg_learn.py hosted with ❤ by GitHub view raw

### Deep Deterministic Policy Gradients

```
# Get Q-values for actions from trajectory
current_q = self.critic(obs_tensor, action_tensor)

# Get target Q-values
target_q = reward_tensor + self.gamma * self.target_critic(next_obs_tensor, self.target_acto

# L2 loss for the difference
critic_loss = F.mse_loss(current_q, target_q)

critic_loss.backward()

# Actor loss based on the deterministic action policy
actor_loss = - self.critic(obs_tensor, self.actor(obs_tensor)).mean()

actor_loss.backward()
```

ddpg_learn.py hosted with ❤ by GitHub view raw

### Complete Implementations

Completed Modular implementations of the full pipeline can be viewed at *activatedgeek/torchrl*.