On Gradient Descent and its Variants

Ozaner Hansha

April 12, 2021

Abstract

We examine the gradient descent (GD) algorithm, with respect to arbitrary real functions $f : \mathbb{R}^n \to \mathbb{R}$. In particular, we elaborate on its formulation, complexity, and convergence conditions. We then introduce and do the same with two variants of GD, namely stochastic gradient descent (SGD) and Adam.

Background

First developed by French mathematician Cauchy in 1847 [1], **gradient descent** (GD) is an iterative optimization algorithm used to find a minimum, specifically a *local* minimum, of a multivariate real function. GD has become more prominent recently due to its use in optimizing machine learning models, which have become particularly useful in the last decade or so. It is for this very reason that stochastic gradient descent, a modification of GD, is often cited as 'the workhorse of machine learning.'

And so, in an effort to better inform the usage of this algorithm, this paper will attempt to provide both a intuitive and mathematical basis for GD. We then consider how this basis changes, if at all, when considering two of GD's modern incarnations: stochastic gradient descent (SGD), and Adam.

Gradient Descent

Overview

Before we dive into the formulation of GD, let us first get an intuitive view of how the algorithm works. Consider a function $f(x_1, x_2)$ which we wish to minimize. This function takes two parameters, and so we can represent it via the following 3D plot:

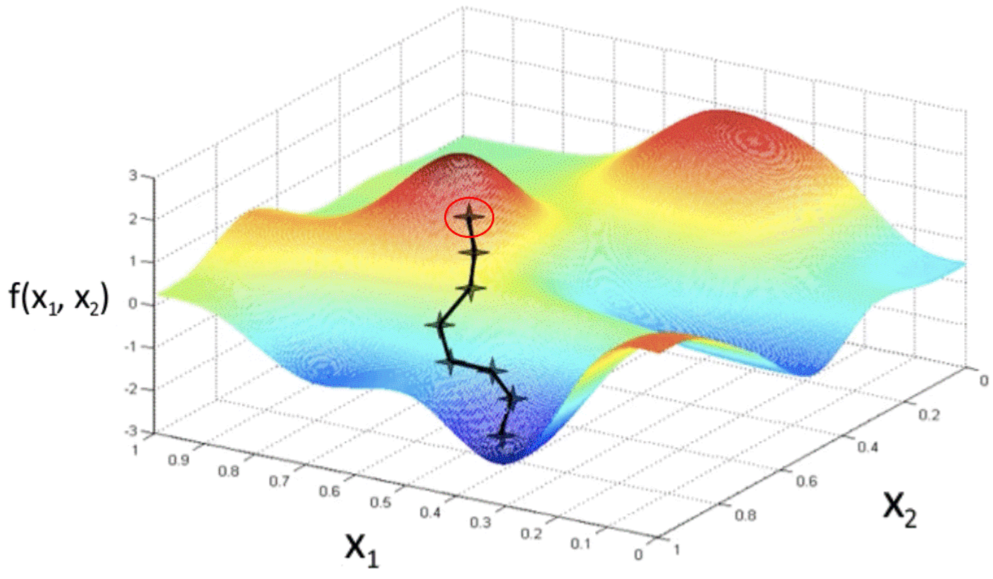


Figure 1: An arbitrary 2D surface. Red-Blue $\iff$ High-Low.

With this setup, we can now describe the GD algorithm in geometric terms. Given some initial guess, here circled in red, GD prescribes:

1. At the point you are on, find the direction which slopes downwards the most.
2. Step down this direction a small amount.
3. Repeat 1 & 2 until there is no direction which slopes downards at our position.

We can picture this like rolling a ball down a hill, sans the momentum. The ball will keep rolling down the hill until it reaches a valley which, in this case, is our minimum. Note however that the minimum our point/ball ends up in is not necessarily the global minimum, just a local one. After all, there may always be a lower valley to reach if we could travel uphill slightly more, as in this 2D example:

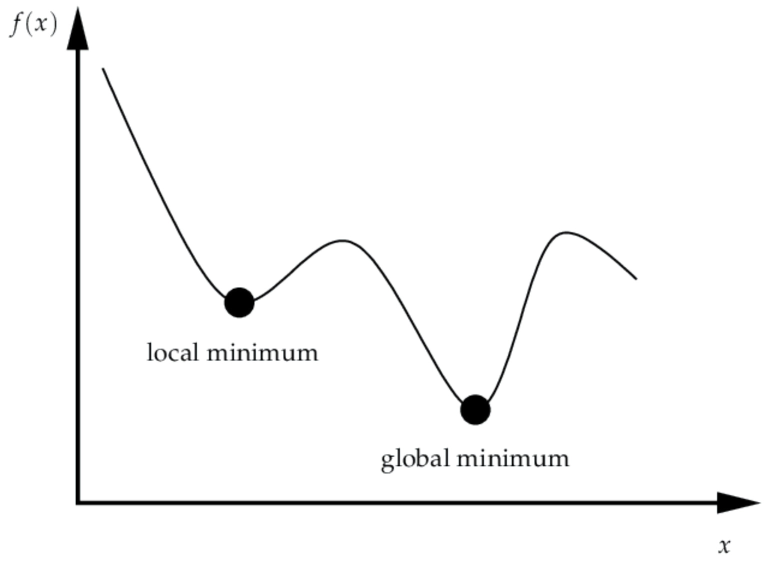


Figure 2: An arbitrary 1D surface.

Note that this property of GD of only considering its immediate surroundings and potentially missing out on a global minimum is called **greediness**. While such a property may not seem desirable, it is impossible, at least in general, to consider an entire function at once to ascertain its minimum. If it was, the problem of optimization wouldn't be much of a problem at all!

Formulation

Now let us convert the intuitive description of the algorithm given above into a mathematical one. We'll start by first laying out the algorithm:

```
function GradientDescent($f, \gamma, iter$)
    $\mathbf{x}_0 \leftarrow$ initial value
    for $i \in \{1, 2, \cdots, iter\}$ do
        $\mathbf{x}_i \leftarrow \mathbf{x}_{i-1} - \gamma \nabla f(\mathbf{x}_{i-1})$
    end for
    return $\mathbf{x}_{iter}$
end function
```

Our goal is to minimize $f : \mathbb{R}^n \to \mathbb{R}$, an n dimensional multivariate function. You'll note that in formulating GD mathematically, we have generalized it to $n > 3$. That said, while it is intuitive that GD works for $n = 2, 3$ (e.g. a ball rolling down a hill) and while the algorithm is no different in higher dimensions, we should still be dubious that it is *effective* for $n > 3$. This is a point we will confirm, at least partially, in the section on convergence.

The algorithm has us start with an initial value $\mathbf{x}_0$, analogous to the ball's starting position in the hill example. It would be ideal to set this to an educated guess, but often we have no idea where the minimum will be before hand. As such, for practical usage, it is common to set it to some random vector near the origin, although a more appropriate choice would depend on the function f.

Once this initial value is set, we compute the gradient of f at $\mathbf{x}_0$, multiply it by some small value γ, then subtract it from $\mathbf{x}_0$ to produce a new, usually smaller, $\mathbf{x}_1$. Note that the usage of the gradient of f means that GD demands that f be differentiable, although with slight modification sub-differentiability may suffice.

The gradient here is analogous to the 'direction that slopes downward the most' in the hill example, while γ is analogous to how big the steps we take are in said direction. We often set γ to be some small value. If the steps are too big, we might overstep the minimum, and if they were *too* small we'd waste computing time with all the extra iterations:

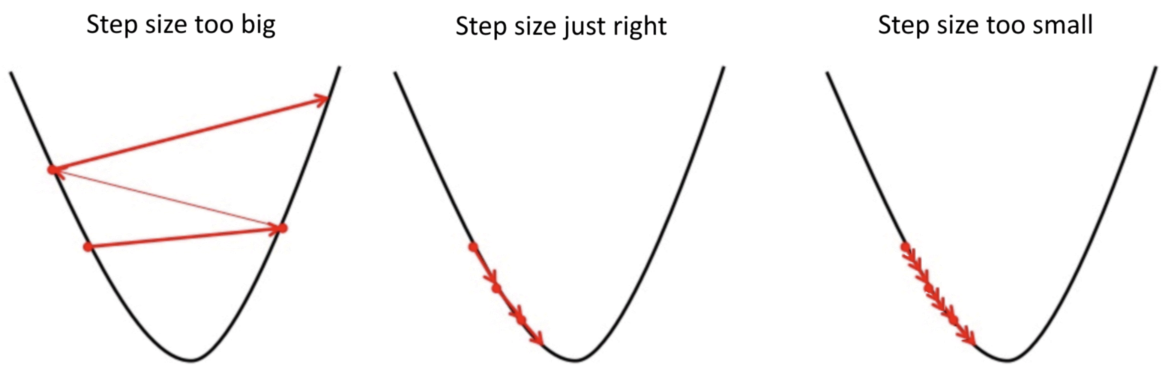


Figure 3: The problem with extreme step sizes.

But 'small' is relative, and so we must take care to scale γ appropriately to f or vice versa. There are also more structured ways of choosing values for γ such as scheduling, where the value of γ changes every iteration. An example of this is a linear schedule, where γ_i decreases by some fixed amount every iteration. A full discussion of these methods are beyond the scope of this paper however.

Getting back to the algorithm, after $\mathbf{x}_1$ is computed, the process is repeated to calculate $\mathbf{x}_2$ and so on *iter* times. It is at this point that our algorithm is finished and returns our result: $\mathbf{x}_{iter}$. How do we decide on *iter*? Again the most appropriate value depends on the problem. Often, instead of a set number of iterations, GD's stopping condition is whether $||\mathbf{x}_i - \mathbf{x}_{i-1}|| < \delta$ for some small, predetermined value of δ. This condition signals that $\mathbf{x}_i$ has converged and that more iterations are unnecessary. When this stopping condition is used, it is often coupled by having a variable *maxiter* represent the maximum number of iterations GD can take before giving up. We can reprhase GD, then, as [2]: picked

```
function GradientDescent($f, \gamma, \delta, maxiter$)
    $\mathbf{x}_0 \leftarrow$ initial value
    for $i \in \{1, 2, \cdots, maxiter\}$ do
        $\mathbf{x}_i \leftarrow \mathbf{x}_{i-1} - \gamma \nabla f(\mathbf{x}_{i-1})$
        if $||\mathbf{x}_i - \mathbf{x}_{i-1}|| < \delta$ then
            return $\mathbf{x}_i$
        end if
    end for
    return $\mathbf{x}_{maxiter}$
end function
```

Complexity

In the worst case scenario, our GD algorithm takes $k = maxiter$ iterations to return a result. As such, the worst case time complexity of GD is $O(k)$ right? Well, this ignores the time taken to compute the gradient of f with respect to some $\mathbf{x}_i$. Indeed, while we've written this as a single step in the previous section, it will take at least n steps, one for each component of the gradient. Since all n of these components need to be computed for each iteration, our final worst-case temporal complexity is:

$$O(k^2 n) \qquad \text{assuming the time to compute } \tfrac{\mathrm{d}f}{\mathrm{d}x_i} \text{ doesn't depend on } n.$$

In other words, the time taken has a quadratic relationship with number of iterations of GD, and a linear relationship with dimensionality of the function being minimized.

In terms of spatial complexity, i.e. how much storage is needed to run GD, we only need to store our current point (or the current point AND the last point if we use the convergence criterion), and whatever storage is needed to store/compute the gradient. Since there are n components to these vectors, our spatial complexity is simply:

$$O(n)$$

Convergence

While we cannot demonstrate all functions which converge under GD, we can highlight an important subclass of them. Let our function f be convex and differentiable, with ∇f Lipschitz continuous, meaning:

$$\exists L > 0, \forall x, y, \quad \|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L\|x - y\|$$

Then if we run GD for k iterations with a fixed step size $\gamma \leq 1/L$, then we have:

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{2k\gamma}$$

Where $\mathbf{x}^*$ achieves the minimum of f. This result means that, under the given conditions, GD is guaranteed to converge and with a rate at least proportional to $1/k$ [3].

Stochastic Gradient Descent

GD's Computational Burden

As mentioned previously, GD is often used in fitting machine learning models. In these cases, we are trying to find n parameters $\boldsymbol{\omega}$ that minimize the error our model $M_{\boldsymbol{\omega}} : X \to Y$ produces with some dataset $D = \{(\mathbf{x}_i, y_i) \mid i \in [1..S]\}$ with S samples. Using the mean square error (MSE), our function f would look something like this:

$$f(\boldsymbol{\omega}) = \frac{1}{S}\sum_{i=1}^{S}(M_{\boldsymbol{\omega}}(\mathbf{x}_i) - y_i)^2$$

Note that the closer the model's predictions of $\mathbf{x}_i$ are to the real answers y_i, the lower f is. Indeed if there are no errors we expect to see f reach 0. It seems, then, that minimizing f is a reasonable approach to achieving parameters $\boldsymbol{\omega}$ for a more accurate model $M_{\boldsymbol{\omega}}$, assuming doing better on this training set D produces a better model.

This problem forms the backbone of modern machine learning, which often boils down to finding optimal parameters of very complex models. However, there is a problem. In real world data sets, the size S is often enormous. Possibly on the order of millions. Computing ∇f for each iteration of gradient descent can be prohibitively time consuming. This is because the number of components n of $\boldsymbol{\omega}$ (i.e. the number of parameters in the model) can go into the hundreds.

Mini-batch Gradient Descent

Since computing the gradient for each of the potentially millions of training examples is burdensome to do for each iteration of GD, we could simply look at less of the data set at a time. One approach to leverage this intuition is the following:

1. Partition the data set D into p equally sized subsets $\{D_i\}_{i=1}^{p}$, each called a **mini-batch**.
2. For each mini-batch i:
   (a) Run GD on a modified function f_i defined as:

$$f_i(\boldsymbol{\omega}) = \frac{1}{|D_i|}\sum_{(\mathbf{x},y)\in D_i}(M_{\boldsymbol{\omega}}(\mathbf{x}) - y)^2$$

   (b) Update $\boldsymbol{\omega}$ to the result of this GD.

3. After all mini-batches, we arrive at our final $\boldsymbol{\omega}$, and can repeat step 2 more times if necessary.

This method, aptly dubbed **mini-batch gradient descent** [4], has the advantage of not having to compute ∇f for every single training example before some progress can be made in moving towards the minimum. More progress made down the gradient speeds up later iterations. The trade-off is that the calculation of the gradient at each mini-batch is no longer exact, but an approximation using only some of the data. In other words, our algorithm was made a bit more noisy. In order to minimize this trade-off, we must make sure the examples in our mini-batches are randomly distributed so as to be representative of the entire dataset.

All that said, this trade-off is usually well worth due to the time savings. Indeed, with said time savings, one could just run more iterations of this mini-batch GD, and/or even subdivide the dataset further...

Stochastic GD

Taking the mini-batch approach to the extreme, we could have mini-batches of size 1, effectively computing updating our parameters $\boldsymbol{\omega}$ for each datapoint $d \in D$. This is called **stochastic gradient descent** (SGD) [4]. Below we demonstrate the SGD algorithm:

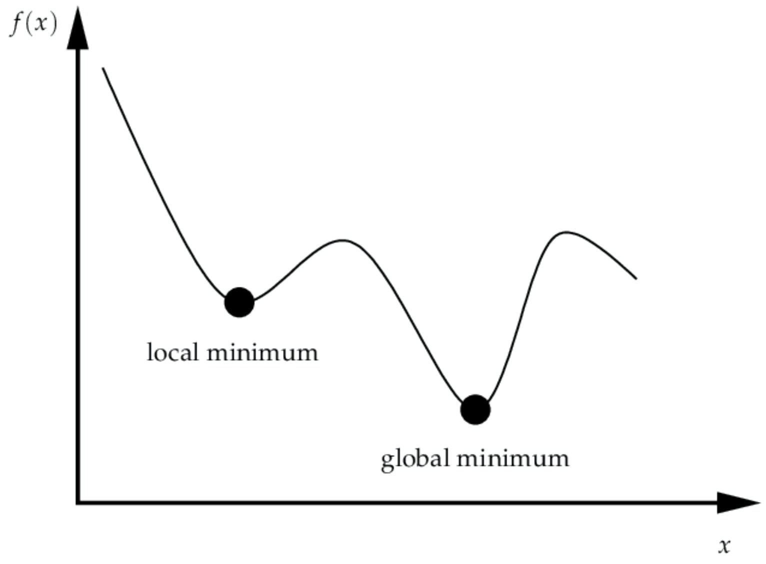
```
function StochasticGD($\gamma, \delta, maxiter$)
    $\boldsymbol{\omega}_0 \leftarrow$ initial value
    $i \leftarrow 1$
    repeat
        Pick random $(\mathbf{x}, y) \in D$
        $\boldsymbol{\omega}_i \leftarrow \boldsymbol{\omega}_{i-1} - \gamma\nabla(M_{\omega_{i-1}}(\mathbf{x}) - y)^2$ $i \leftarrow i + 1$
    until $\boldsymbol{\omega}_i - \boldsymbol{\omega}_{i-1} < \delta \vee i > maxiter$
    return $\boldsymbol{\omega}_i$
end function
```

Here we simply pick a random datapoint $d \in D$, slightly adjust our parameter values $\boldsymbol{\omega}$ based on the gradient we calculate with d, and repeat until we converge or just hit our iteration limit. While noisy, as it only calculates an approximate gradient, SGD has the benefit of immediate feeback for the model parameters, making learning quick. And as long as the datapoints uniformly random (and thus representative of the dataset) and enough iterations are run, the parmeters should converge just as well as they did with GD, just in a faster and slightly less precise manner.

Momentum based methods

While there many variants of GD, specifically of SGD, most of them make use of momentum [2]. To see what this means, consider the ball on a hill example we made use of previously. The example was a little flawed as the ball/point/parameters GD keeps track of doesn't have momentum. It just travels down the hill/gradient at a constant speed, determined by the step-size γ.

A ball in real life, however, would accrue some momentum from falling down a hill, meaning it would not follow the steepest path down a hill but some combination of that steepest path and whatever direction the ball has momentum in. Such a property would allow the ball to clear small hills that may be in the way to lower points on the hill:



Thus, keeping track of the ball's, or rather, the point's momentum (or some equivalent of it) might serve useful in creating a GD variant that is resistant to bumps in the high dimensional surface f defines. One way this is done is modifying the parmeter update formula:

$$\boldsymbol{\omega} \to \boldsymbol{\omega} - \gamma \nabla f(\boldsymbol{\omega}) + \alpha \Delta \boldsymbol{\omega}$$

Where $f(\boldsymbol{\omega})$ is the function we seek to minimize, γ is the step-size, $\Delta \boldsymbol{\omega}$ is the change made during last update, and α is how much weight we should put towards that last update. The greater α is, the more the momentum of the parameters comes into play. We can think of it as analogous to the strength of gravity on the ball. We can also apply a schedule to it, just as with γ, so that its influence diminishess as our paremeters become more and more fine-tuned.

References

[1] Claude Lemaréchal. Cauchy and the gradient method, 2012.

[2] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv e-prints*, page arXiv:1609.04747, September 2016.

[3] Ryan Tibshirani. 10-725: Optimization. `https://www.stat.cmu.edu/~ryantibs/convexopt-F13/scribes/lec6.pdf`, Sep 2013.

[4] Jiawei Zhang. Gradient Descent based Optimization Algorithms for Deep Learning Models Training. *arXiv e-prints*, page arXiv:1903.03614, March 2019.