How to Control Gradient Descent

Michael Zimmer

January 15, 2021

This article provides an introduction to the full paper on the Neograd algorithms [Zimmer, 2020a] (submitted for publication).

Introduction

There are many computations that are done in science and technology in which one has to minimize what is known as a cost function (CF). People use CFs because they're easy to construct, and because their minimum corresponds to what we seek to know. They have been used throughout machine learning (ML), physics, and many other fields. As it turns out, the gradient descent (GD) algorithm is especially convenient for finding minima of the CF. Recall that GD operates on a function $f(\theta)$ by updating its argument from θ_{old} to θ_{new} via

$$\theta_{new} = \theta_{old} - \alpha \nabla f \tag{1}$$

where α is a scalar known as the learning rate. This updating is repeated until a stopping condition is met. As shown, one only needs to compute the gradient (∇f), and to set α to a suitably small number.

However, finding the "best" value for α turns out to be an issue. Often, people will do several trial optimization runs using different α values to see which seems to lead to the best behavior, and then just select that value. However, even the mere idea of using a single value of α for an entire optimization rate is flawed. I argue that it's comparable to trying to drive between two cities at a single speed. For example, would the best speed to drive between Chicago and St Louis be 10mph, 40mph or 70mph? Certainly, for different parts of the trip, different speeds would be preferred. It's a similar situation with finding the best α for GD.

Now, there have been many papers on GD, so much so, that it's hard to convince anyone to look at another one. To convince you to keep reading, I'll tell you that Neograd allows you to *routinely* reach smaller values of the CF by a factor of 10^8 to 10^{12}. The introduction that's presented here will show an algorithm that far surpasses the results from all previous GD variants, such as Adam. In addition, it provides insight into why those other algorithms have limited performance.

The ρ Diagnostic

The first step is to introduce a measure that says when the learning rate is too large or small based on a pair of CF values. This diagnostic measure is motivated by Fig. 1, which displays the θ value before updating (θ_{old}) and after updating (θ_{new}), as well as

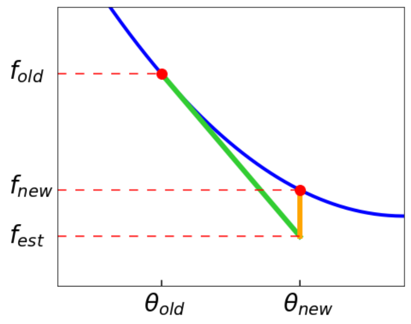

Figure 1: The blue curve is the cost function. The green line is tangent to the cost function at θ_{old}, and after extrapolating it to θ_{new}, equals f_{est}. The orange line is the difference between f_{new} and f_{est}.

the CF values at those points: $f_{old} = f(\theta_{old})$, $f_{new} = f(\theta_{new})$. In addition, we define f_{est}, which is the estimated CF value based on the GD algorithm:

$$f_{est} = f_{old} + \nabla f \cdot d\theta \tag{2}$$

where $d\theta = \theta_{new} - \theta_{old}$. Note that this is really just a linear equation, like the familiar "$y = mx + b$", except here $y = f_{est}$, $m = \nabla f$, $x = \theta_{new}$, and $b = f_{old} - \nabla f \cdot \theta_{old}$ The range of values of f_{est} is shown as a green line in Fig. 1 as $d\theta$ varies from 0 to $\theta_{new} - \theta_{old}$.

However, what is really of interest for us is the orange line in the figure. It represents the gap or "deviation" between the estimate f_{est}, and the actual value of the CF, f_{new}. In the figure, this deviation indicates whether α is too large or small. However, to make it really useful, it should be dimensionless. Here, that is achieved by dividing it by the vertical drop from f_{est}. This leads to the ρ diagnostic measure:

$$\rho = \left| \frac{f_{new} - f_{est}}{f_{old} - f_{est}} \right|. \tag{3}$$

Note this measure is invariant with respect to translating and scaling f (while keeping $d\theta$ constant), which is exactly what is needed. Thus, ρ serves as a sort of universal measure of the appropriateness of the size of α.

How to think about ρ

First note that for very small values of ρ, it is proportional to α. Next, one must understand what an acceptable value for ρ is. Certainly, values that are extremely small will certainly lead to inefficient optimization, while those that are too large will lead to relatively erratic and uncontrolled updates.

However, before giving a direct answer on a "best value" for ρ, first consider this analogy. Imagine someone with an old car who's trying to drive as fast as possible. He won't even be looking at the speedometer; he only pays attention to when the car begins physically shaking at high speed. This person reasons, "if the car is shaking only a little, then I can go faster, and if it's shaking a lot, I should slow down". Thus, the amount of shaking acts as a proxy for controlling the gas pedal, and indeed within the context of GD ρ can be employed in a similar manner with respect to α.

The approach that will be adopted will be to attempt to adjust α so that ρ is kept at some target value. This is called the *constant ρ ansatz*, and an α that achieves that is called the *ideal* learning rate. (In the case where ρ can only be (mostly) kept to only a target interval, it will be referred to as a *near-ideal* learning rate.) The effectiveness of this approach will be demonstrated next on a simple cost function. In the paper, a typical target value was $\rho = 0.1$ and a typical target interval was $(0.015, 0.15)$.

An Example

A good example for demonstrating the usefulness of ρ is the 1-dimensional quartic CF: $f = \theta^4$. In this case

$$d\theta = -\alpha\nabla f = -4\alpha\theta^3 \tag{4}$$

$$f_{old} = \theta^4 \tag{5}$$

$$f_{new} = (\theta + d\theta)^4 \tag{6}$$

$$f_{est} = f_{old} + \nabla f \cdot d\theta \tag{7}$$

A straightforward calculation [Zimmer, 2021] reveals

$$\rho = 6q - 16q^2 + 16q^3\,, \tag{8}$$

where $q = \alpha\theta^2$. The most important feature to take note of is that ρ now depends on θ. This means that a single α will have a different effect depending on the nearness to the minima. Even more interesting is that it can be easily inverted (approximately) in favor of α.

$$\alpha \approx \rho/(6\theta^2) \tag{9}$$

Although quite simple, this is actually quite remarkable. We now have a formula by which to set α given a desired value of ρ. With each iteration of the GD algorithm, as θ changes, this formula guarantees that a target value of ρ is maintained; this is called an "Ideal GD" algorithm. This is illustrated in Fig. 2, which shows $\log f$ and $\log \rho$ vs iteration for Adam and the Ideal GD. The initial value was $\theta = -3$ for both; for Adam: $\alpha = 0.15$, $\beta = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. This graph shows that the value reached with Ideal GD is more than a factor of 10^{15} times smaller than that from Adam after only 150 iterations. Obviously, the longer the run, the larger this factor becomes. In addition, it shows that with Adam, the stage at which it begins to plateau coincides with its ρ value dropping. In the figure, this occurs near the 50th iteration. This feature was seen repeatedly with Adam. Since ρ is not controlled with Adam,

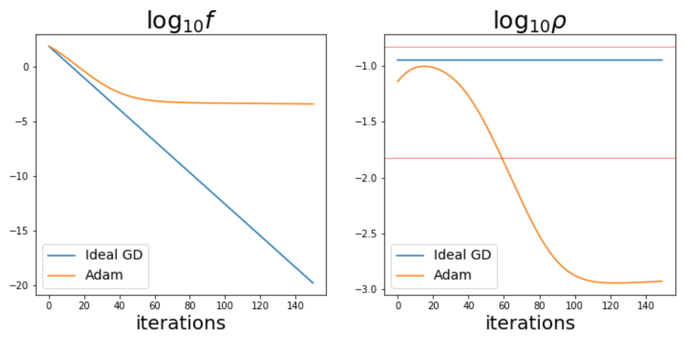

Figure 2: Comparison between Neograd and Adam on the quartic CF. Note the linear descent in the $\log f$ for Neograd, while there is a plateau for Adam.

the algorithm naturally evolves such that ρ takes on very small values, causing it to become very inefficient. Essentially, *Adam has a built-in regularization, due to its lack of effectiveness in reaching a minimum.* It's akin to early stopping. In addition, running Adam was a challenge because a number of trial optimization runs had to be done even to find an α that led to this sub-optimal result. With respect to the Ideal GD algorithm, the main issue is that it makes the CF so small that one has to worry about machine precision errors, since the numbers get that small.

Efficiently computing ρ

In the previous section, we gave an example of a simple cost function and determined an exact relation between ρ and α. Using that, it was possible to set α as a function of a target value for ρ. In a real application, the CF becomes extremely complicated, and the computation of such a relationship becomes out of the question. Furthermore, in a real application, a measurement of ρ will be used in setting α, and as it stands now, the formula for ρ makes it appear two computations of the CF will be needed, per iteration. However, it's possible to easily measure ρ with only one computation of the CF per iteration. Let us first begin with the vanilla version of GD, as shown in Algo. 1

Algorithm 1 : Sequence of operations for a basic GD algorithm

```
Init: $\boldsymbol{\theta}$
for i = 1 to num+1 do
    $f = f(\boldsymbol{\theta})$
    $\boldsymbol{g} = \boldsymbol{\nabla} f(\boldsymbol{\theta})$
    $d\boldsymbol{\theta} = -\alpha \boldsymbol{g}$
    $\boldsymbol{\theta} = \boldsymbol{\theta} + d\boldsymbol{\theta}$
end for
Return: $\boldsymbol{\theta}$
```

The most important thing about this version is the order of the following operations within the FOR-loop: (1) evaluate CF; (2) evaluate gradient; (3) update θ. This "CF-gradient-update" sequence is now modified so there's an initial "CF" before the loop, and inside the loop the order is "gradient-update-CF". This rewriting is equivalent, in the ways that matter. This allows it to have access to f_{new} and f_{old}, and hence ρ can be implemented with almost no additional overhead. This rewriting leads to Algo. 2.

Algorithm 2 : Amended basic GD algorithm, now including computation of ρ

```
1: Init: $\boldsymbol{\theta}_{old}$
2: $f_{old} = f(\boldsymbol{\theta}_{old})$
3: **for** i = 1 to num **do**
4:     $\boldsymbol{g} = \boldsymbol{\nabla} f(\boldsymbol{\theta}_{old})$
5:     $d\boldsymbol{\theta} = -\alpha \boldsymbol{g}$
6:     $\boldsymbol{\theta}_{new} = \boldsymbol{\theta}_{old} + d\boldsymbol{\theta}$
7:     $f_{new} = f(\boldsymbol{\theta}_{new})$
8:     $f_{est} = f_{old} + \boldsymbol{g} \cdot d\boldsymbol{\theta}$
9:     $\rho =$ get_rho$(f_{old}, f_{new}, f_{est})$
10:    $f_{old} = f_{new}$
11:    $\boldsymbol{\theta}_{old} = \boldsymbol{\theta}_{new}$
12: **end for**
13: Return: $\boldsymbol{\theta}_{new}$
```

Finally, if one would prefer not to do this rewriting, it is also possible to include an IF-conditional inside the original algorithm and achieve the same result. The version used here was chosen since it's preferred by the author.

Approximating α

With that out of the way, the next step is understanding how to use the original ρ formula to set α. It turns out it can be used in an approximate manner, and with no *significant* computational overhead. With the definition of ρ in mind (Eqn. 3), define the quantities A and B via

$$A = (f_{new} - f_{est})/\alpha^2$$
$$B = (f_{est} - f_{old})/\alpha\,.$$

This is done to pull out the leading order dependence on α and make it explicit. As a result, A and B are constants, to leading-order in α. However, note that in general A retains α-dependence, while B has no α-dependence. Using these three equations, an expression for α easily follows

$$\alpha = \left|\frac{B}{A}\right| \rho\,. \tag{10}$$

Note that after an update, A, B, ρ can be computed, and they produce an α via this equation. The simplest way this equation can be used is to compute α between lines 9 and 10 in Algo. 2. This approach works very well, assuming there are no extreme features in the CF. Also, implicit in this simple approach is the assumption that the ρ computed in line 9 doesn't differ significantly from its target value. If it does, an adjustment can be made, as shown in the paper [Zimmer, 2020a]. Another improvement that can be made is to introduce momentum, which is also shown the paper. When both of these enhancements are used, the version of the algorithm is referred to as NeogradM. It is this version that works well against Adam in real applications; the reader is advised against skipping these enhancements.

Here is an example of the application of NeogradM to the problem of digit recognition (with data from Scikit-learn [2020]). The CF was a cross-entropy penalty constructed between the output of a neural net and the training label. Details of the model are in the paper. As can be seen in Fig. 3, the values of the CF reached using NeogradM

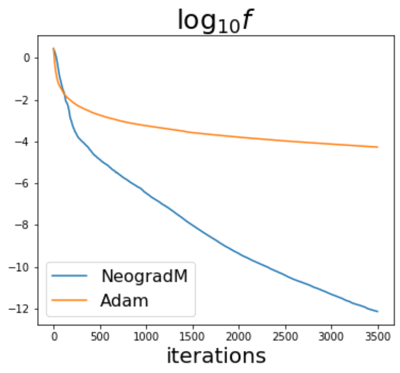

Figure 3: A comparison plot between NeogradM and Adam on the CF from a neural net model for digit recognition. Note the 8 order of magnitude difference between values for $\log f$.

are about 10^8 times lower than with Adam. In addition, as shown in the paper in Fig. 13, the value of ρ mainly stays in the target interval; the ρ values for Adam drop below the target interval by a factor of about 100. Once again, the picture with Adam is that ρ becomes very small, and the CF profile has a plateau.

Final comments

The purpose of this article has been to introduce you to the main concepts from the full paper on the Neograd family of algorithms. Perhaps the most important takeaway is that the ρ metric is easy to implement and is a reliable indicator of the learning rate. If you go on to implement it in your own gradient descent programs, that is half the battle. Then, you might see that your current program could be run faster. At that point, perhaps you'll be willing to take the next step and implement NeogradM; sample code is on Github [Zimmer, 2020b].

In addition to what has been reviewed here, the paper [Zimmer, 2020a] includes a number of other useful results, such as other metrics, another derivation of GD, and other results.

Hope you enjoyed the article!

References

M.F. Zimmer. Neograd: Gradient descent with a near-ideal learning rate. *arXiv preprint*, arXiv:2010.07873, 2020a.

M.F. Zimmer. Computing rho for three examples, 2021. URL `http://www.neomath.com/gallery/rho_algebra.pdf`.

Scikit-learn. Digits dataset, 2020. URL `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html`.

M.F. Zimmer. Github repository, 2020b. URL `www.github.com/mfzimmer`.