Why Gradient Descent -- Not the Best Optimization Technique -- Works Best in Neural Networks: Qualitative Explanation

Jonatan Contreras  
*The University of Texas at El Paso, jmcontreras2@utep.edu*

Martine Ceberio  
*The University of Texas at El Paso, mceberio@utep.edu*

Olga Kosheleva  
*The University of Texas at El Paso, olgak@utep.edu*

Vladik Kreinovich  
*The University of Texas at El Paso, vladik@utep.edu*

Follow this and additional works at: [https://scholarworks.utep.edu/cs_techrep](https://scholarworks.utep.edu/cs_techrep)

Part of the Computer Sciences Commons, and the Mathematics Commons

Comments:

Technical Report: UTEP-CS-21-02


**Recommended Citation**

Contreras, Jonatan; Ceberio, Martine; Kosheleva, Olga; and Kreinovich, Vladik, "Why Gradient Descent -- Not the Best Optimization Technique -- Works Best in Neural Networks: Qualitative Explanation" (2021). *Departmental Technical Reports (CS)*. 1535.

[https://scholarworks.utep.edu/cs_techrep/1535](https://scholarworks.utep.edu/cs_techrep/1535)

This Article is brought to you for free and open access by the Computer Science at ScholarWorks@UTEP. It has been accepted for inclusion in Departmental Technical Reports (CS) by an authorized administrator of ScholarWorks@UTEP. For more information, please contact lweber@utep.edu.
Why Gradient Descent – Not the Best Optimization Technique – Works Best in Neural Networks: Qualitative Explanation

Jonatan Contreras, Martine Ceberio, Olga Kosheleva, and Vladik Kreinovich

Abstract In a usual Numerical Methods class, students learn that gradient descent is not an efficient optimization algorithm, and that more efficient algorithms exist, algorithms which are actually used in state-of-the-art numerical optimization packages. On the other hand, in solving optimization problems related to machine learning – and, in particular, in currently most efficient deep learning – gradient descent (in the form of backpropagation) is much more efficient than any of the alternatives that have been tried. How can we reconcile these two statements? In this paper, we explain that, in reality, there is no contradiction here. Namely, in usual applications of numerical optimization, we want to attain the smallest possible value of the objective function. Thus, after a few iterations, it is necessary to switch from gradient descent – which only works efficiently when we are sufficiently far away from the actual minimum – to more sophisticated techniques. On the other hand, in machine learning, as we show, attaining the actual minimum is not what we want – this would be over-fitting. We actually need to stop way before we reach the actual minimum. Thus, we do not need to get too close to the actual minimum – and so, there is no need to switch from gradient descent to any more sophisticated (and more time-consuming) optimization technique. This explains why – contrary to what students learn in Numerical Methods – gradient descent is the most efficient optimization technique in machine learning applications.

1 Formulation of the Problem

Optimization is ubiquitous. In many practical situations, we need to find the best alternative.
Each alternative can be described by the values of the corresponding quantities \( x_1, \ldots, x_n \). The quality of different alternatives can be usually described by assigning numerical values to each alternative, so that the smaller the value, the better the alternative. Crudely speaking, this value describes the drawbacks of this alternative.

If we denote the numerical value corresponding to the alternative \( (x_1, \ldots, x_n) \) by \( f(x_1, \ldots, x_n) \), then the task of finding the best alternative takes the following form: find the values \( x_1, \ldots, x_n \) for which the given function \( f(x_1, \ldots, x_n) \) attains the smallest possible value.

**Gradient descent: a brief reminder.** One of historically first optimization techniques is *gradient descent*, an iterative method in which:

- we start with some point \( x^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)}) \) and
- then, on each iteration, given the point \( x^{(k)} = (x_1^{(k)}, \ldots, x_n^{(k)}) \), we compute the next iteration point \( x^{(k+1)} = (x_1^{(k+1)}, \ldots, x_n^{(k+1)}) \) as

\[
    x_i^{(k+1)} = x_i^{(k)} - \alpha \cdot \frac{\partial f}{\partial x_i^{(k)}} ,
\]

for some appropriate value \( \alpha \).

**Known fact: gradient descent is not the best optimization technique.** While gradient descent is one of the simplest optimization techniques, it is, to put it mildly, not the best one. Textbooks on numerical methods usually start the description of optimization techniques with gradient descent, but then immediately move to more efficient methods that use second derivatives of the objective function \( f(x_1, \ldots, x_n) \) — methods that are actually used in numerical optimization packages; see, e.g., [3].

**Machine learning: one of the important applications of optimization.** One of the important applications of optimization is *machine learning*. Let us briefly recall what machine learning is about.

In many practical situations, we want to estimate or predict the value of a quantity \( v \) — e.g., tomorrow’s temperature in El Paso. We know that the value of the desired quantity is determined by the values of several other quantities \( u_1, \ldots, u_m \). For example, we know that to determine tomorrow’s temperature in El Paso, we need to know today’s values of temperature, wind speed, humidity, etc., at different locations in and near El Paso.

In some cases, we have an algorithm \( G \) that enables us to estimate \( v \) based on the known values of \( u_i \), as \( v = G(u_1, \ldots, u_m) \). However, in many other practical cases, we do not know such an algorithm. What we do know in such cases is several examples of situations \( s \) in which we know both the values of \( u_1^{(s)}, \ldots, u_m^{(s)} \) of the quantities \( u_i \) and the corresponding value \( v^{(s)} \) of the desired quantity \( v \). In view of the following analysis it is important to mention that these values come from measurements are, thus, only approximately known; see, e.g., [4].
Based on the available information – i.e., on the values $D_1^{(s)}, \ldots, D_m^{(s)}$ and $v^{(s)}$ – we need to find a function $G(u_1, \ldots, u_m)$ for which $v^{(s)} \approx G(u_1^{(s)}, \ldots, u_m^{(s)})$ for all $s$. Determining such a function is known as machine learning; see, e.g., [1]. Usually, we have a generic model

$$G(u_1, \ldots, u_m) = F(u_1, \ldots, u_m, x_1, \ldots, x_n)$$

with parameters $x_1, \ldots, x_n$, and we need to find the values of the parameters $x_1, \ldots, x_n$ for which the values $v^{(s)}$ are the closest to the values $F(u_1^{(s)}, \ldots, u_m^{(s)}, x_1, \ldots, x_n)$. Often, this closeness is described by the Euclidean distance between the points $(v^{(1)}, \ldots, v^{(S)})$ and

$$(F(u_1^{(1)}, \ldots, u_m^{(1)}, x_1, \ldots, x_n), \ldots, F(u_1^{(S)}, \ldots, u_m^{(S)}, x_1, \ldots, x_n)).$$

In this case, the task of finding the values $x_1, \ldots, x_n$ for which these two points are the closest means that we minimize the following objective function:

$$f(x_1, \ldots, x_n) \overset{\text{def}}{=} \sum_{s=1}^{S} \left( v^{(s)} - F(u_1^{(s)}, \ldots, u_m^{(s)}, x_1, \ldots, x_n) \right)^2. \quad (2)$$

Alternatively, other measures of closeness are used; see, e.g., [2].

At present, the most efficient machine learning technique is neural networks, especially deep neural networks [2].

**Surprisingly, gradient descent is very efficient in machine learning.** As we have just mentioned, the main task of machine learning is optimization.

- In general, as we have previously mentioned, gradient descent is not a very efficient optimization technique: when we use more sophisticated techniques, we get much better results.
- However, surprisingly, there is an application area of optimization where the situation is reverse: gradient descent – in the form of the so-called backpropagation – leads to a very efficient solution to the corresponding optimization problems, while numerous attempts to replace gradient descent with more complex numerical optimization techniques made machine learning much less efficient.


**What we do in this paper.** In this paper, we provide a (qualitative) answer to this question.
2 Analysis of the Problem and the Resulting Explanation

Main idea behind gradient descent: a brief reminder. In order to provide a convincing answer to the above why-question, let us recall where the gradient descent method comes from.

The main idea behind gradient descent is that every smooth function \( f(x_1, \ldots, x_n) \) can be approximated, in the vicinity of the point \( x^{(k)} \), by the first few terms in its Taylor expansion. In particular, in the first approximation, it can be approximated by the linear terms in its Taylor expansion:

\[
 f \left( x^{(k)} + \Delta \right) = f \left( x^{(k)} \right) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \cdot \Delta_i. \tag{3}
\]

We want to find the change \( \Delta = (\Delta_1, \ldots, \Delta_n) \) for which the resulting value of the objective function will be the smallest possible. Of course, the approximation (3) is only valid within some vicinity of the point \( x^{(k)} \), so we must restrict ourselves to changed \( \Delta \) for which the length of the change vector does not exceed a certain threshold value \( \varepsilon \), i.e., for which

\[
 \Delta_1^2 + \ldots + \Delta_n^2 \leq \varepsilon^2. \tag{4}
\]

The expression (3) is a linear function. The minimum of the linear function on a set is always attained at its boundary. So, it is sufficient to consider the points of the boundary of the closed ball (4), i.e., on the sphere

\[
 \Delta_1^2 + \ldots + \Delta_n^2 = \varepsilon^2. \tag{5}
\]

To find the minimum of the objective function (4) under the constraint (5), we can use the Lagrange multiplier method. According to this method, this constraint optimization problem can be reduced to the unconstraint minimization of the following auxiliary objective function

\[
 f \left( x^{(k)} \right) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \cdot \Delta_i + \lambda \cdot \left( \Delta_1^2 + \ldots + \Delta_n^2 - \varepsilon^2 \right). \tag{6}
\]

At the point where this function attains its minimum, all its partial derivatives are equal to 0. Differentiating the expression (6) with respect to \( \Delta_i \) and equating the derivative to 0, we conclude that

\[
 \frac{\partial f}{\partial x_i} + 2\lambda \cdot \Delta_i = 0,
\]

hence

\[
 \Delta_i = -\alpha \cdot \frac{\partial f}{\partial x_i},
\]
where we denoted $\alpha \overset{\text{def}}{=} \frac{1}{2\lambda}$. Thus, for $x_i^{(k+1)} = x_i^{(k)} + \Delta_i$, we have exactly the formula (1) describing the gradient descent.

**How accurate is this description.** In the above derivation of the gradient descent method, we started with the formula (3) – the formula that approximates the values of the objective function $f(x_1, \ldots, x_n)$ in the vicinity of the point $x^{(k)}$ by linear terms in its Taylor expansion. In this approximation, we ignore quadratic and higher order terms in the Taylor expansion.

Usually, linear terms in the Taylor expansion are larger than quadratic terms – since:

- linear terms are proportional to the first power of the small differences $\Delta_i$, while
- quadratic terms are proportional to squares and/or products of such small terms.

Similarly, quadratic terms are, in general, larger than cubic and higher order terms. Thus, to gauge how accurate is the linear approximation (3) – i.e., how large is the sum of quadratic and higher order terms – it is sufficient to consider the major of the remaining terms, i.e., the quadratic terms. If we take quadratic terms into account, then the corresponding expression takes the form

$$
f \left( x^{(k)} + \Delta \right) = f \left( x^{(k)} \right) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \cdot \Delta_i + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{\partial^2 f}{\partial x_i \partial x_j} \cdot \Delta_i \cdot \Delta_j. \quad (7)
$$

**So, at least in the beginning of the iterative process, gradient descent works well.** Since the differences $\Delta_i$ are small, we can conclude that when the first derivatives $\frac{\partial f}{\partial x_i}$ are sufficiently large, linear terms in the expression (7) are clearly larger than the quadratic terms. Since in this case, the objective function is accurately approximated by the linear expression (3), optimization (1) based on this approximation (i.e., gradient descent) should work well.

**The problem with gradient descent.** The problem with the gradient descent is related to the fact that in the desired minimum point, according to calculus, all the partial derivatives are equal to 0: $\frac{\partial f}{\partial x_i} = 0$. Thus, as we approach the desired minimum point, the absolute values of these partial derivatives will become smaller and smaller.

In this case, as the linear terms in the expression (7) become smaller and smaller, the quadratic terms become relatively larger and larger in comparison, and so, it is no longer possible to ignore these quadratic terms – they become the main terms. In this case, the linear expression (3) is no longer a good approximation to the objective function. Thus, the optimization step based on this approximation is no longer efficient – we need to take quadratic terms into account (and thus use more complex optimization techniques).

Specifically, when the gradient becomes close to 0, the changes in $x$ – as described by the formula (1) – become smaller and smaller, so the whole process stalls.
It is important to also take into account computational time needed for each iteration. The fact that the quadratic formula (7) provides a better approximation of the objective functions, does not mean that we should always use methods based on the quadratic approximation. Indeed:

- the gradient descent requires computing only the first derivatives, while
- optimizing the quadratic expression (7) requires that we compute the values of the second derivatives as well.

Thus, each iteration of more complex method requires much more computation time than an iteration of gradient descent.

Still, when we get close to the actual minimum, gradient descent, in effect, stalls. Hence, we need to use more complex techniques – in spite of the fact that they require more computation time.

On the other hand, when we are still reasonably far away from the optimum, it makes sense to use gradient descent step: maybe more complex method will lead to slightly larger decrease in the value of the objective function, but since they require much more computation time, their use will drastically slow down the process. This is well known in optimization community, and it is taken into account in the actual algorithm design.

So, we arrive at the following clarification.

Clarification: which algorithms are actually used for optimization. In the currently used numerical optimization algorithms:

- at first, we, in effect, follow the steps of gradient descent;
- only later, when the derivatives become small, we start using more sophisticated techniques.

So gradient descent is still used. In the beginning, it is always beneficial to use gradient descent.

Usually, in numerical optimization problems, we want to find a very accurate location of the minimum. In this case, once we get close to the minimum, we need to switch from gradient descent to more complex optimization techniques.

On the other hand, if we are satisfied with an approximate solution to the optimization problem, there may be no need to switch, and we may just use gradient descent all the way.

What about machine learning? At first glance, it may seem like in machine learning, we do need to find the absolute minimum of the expression (3). However, let us analyze this situation more deeply.

Let us first consider what is the actual minimum of the expression (3). In general, when we have a system of equations, and the number of unknowns is larger than or equal to the number of equations, then this system has a solution. This is a known fact for generic linear systems, and since every non-linear system can be locally approximated by a linear one, this is true for non-linear systems as well. Thus, if the number $n$ of unknowns is larger than or equal to the number $S$ of experiments,
we can, in general, find the values $x_1, \ldots, x_n$ of these parameters for which, for all $s = 1, \ldots, S$, we have

$$F(u_1^{(s)}, \ldots, u_m^{(s)}, x_1, \ldots, x_n) = v^{(s)}. \quad (8)$$

For these values $x_1, \ldots, x_n$, the objective function (2) is equal to 0. Since the expression (2) is always non-negative – as the sum of the squares – this means that the minimum value of this objective function is 0, and this minimum is attained when all the differences between the left-hand and right-hand sides of the formula (8) are equal to 0.

But is this what we want? Not really. Indeed, the value $v^{(s)}$ comes from measurements, and measurements are never absolutely accurate, there is always a non-zero measurement error – the difference between the measurement result $v^{(s)}$ and the (unknown) actual value of the corresponding quantity; see, e.g., [4]. If we ignore measurement errors corresponding to measuring $u_i$ and assume that the dependence $v = G(u_1, \ldots, u_m)$ is exact, then the actual value of the quantity $v$ in the $s$-th experiment is equal to $G(u_1^{(s)}, \ldots, u_m^{(s)})$. In reality, there are measurement errors in measuring $u_i$, and the models are usually approximate.

In general, the difference $v^{(s)} - G(u_1^{(s)}, \ldots, u_m^{(s)})$ is a non-zero random variable. Let $M$ denote the expected value of its square. Usually, measurement errors corresponding to different measurements $s$ are independent. So, due to the large numbers theorem (see, e.g., [5]), for large $S$, the average value of the square of the difference is approximately equal to $M$:

$$\frac{1}{S} \sum_{s=1}^{S} \left( v^{(s)} - G(u_1^{(s)}, \ldots, u_m^{(s)}) \right)^2 \approx M,$$

hence

$$\sum_{s=1}^{S} \left( v^{(s)} - G(u_1^{(s)}, \ldots, u_m^{(s)}) \right)^2 \approx S \cdot M. \quad (9)$$

In particular, for the function $G(u_1, \ldots, u_m) = F(u_1, \ldots, u_m, x_1, \ldots, x_m)$, we want to have

$$\sum_{s=1}^{S} \left( v^{(s)} - F(u_1^{(s)}, \ldots, u_m^{(s)}, x_1, \ldots, x_n) \right)^2 \approx S \cdot M. \quad (10)$$

But the left-hand side of this equality is exactly the objective function $f(x_1, \ldots, x_n)$.

Thus, what we really want is:

• not the absolute minimum (equal to 0) of the objective function (2),
• but rather a point $x = (x_1, \ldots, x_n)$ for which $f(x_1, \ldots, x_n) \approx S \cdot M$.

Comment. The need to avoid the absolute minimum is, by the way, well understood in machine learning – maybe not on the exact mathematical level, but still understood: the solutions for which the objective function is close to 0 are known as over-fitting.
Resulting explanation. How close is the desired point – for which $f(x_1, \ldots, x_n) \approx S \cdot M$ – to the actual minimum for which $f(x_1, \ldots, x_n) = 0$? We do not know a priori how close they are, but we can gauge their closeness by the difference $S \cdot M$ between the values of the objective function in these two points. Usually, we have many examples – i.e., the number $S$ of training examples is large, thus the difference $S \cdot M$ is reasonably large. Hence, we never get close to the absolute minimum of the objective function.

Thus, in line of the above description of when we need to switch from gradient descent to a more complex techniques – when we are close to the actual minimum – for machine learning applications, we do not need to switch at all.

This explains why in machine learning applications, gradient descent is the empirically the most efficient optimization technique.

3 Our Explanation: Summary

In this paper, we try to explain the following seeming contradiction:

• On the one hand, in numerical optimization, it is a well-known fact that gradient descent – a “grandfather” of many optimization techniques – is, by itself, not the most efficient method. Efficient state-of-the-art optimization packages use more complex, more sophisticated optimization algorithms.

• On the other hand, in machine learning applications, the most efficient optimization technique is backpropagation, which is just an algorithmic implementation of gradient descent. Several attempts have been made to replace backpropagation with more complex (and supposedly more efficient) optimization techniques, but in all these attempts, simple gradient descent was shown to perform much better.

How can we explain this paradox? We explain it by showing that both above statements are not exactly true:

• First, in numerical optimization, actually the most efficient algorithms first use fast-and-simple gradient descent steps and only then, when we are very close to the actual minimum, switch to more complex optimization techniques.

• On the other hand, the problems of machine learning are only in the first crude approximation described as optimization problems. In reality, the desired solution should not be too close to the actual minimum of the objective function – the desired value of the objective function should be proportional to the number of training samples and to the accuracy of the corresponding measurements.

Since in machine learning, we do not need to get too close to the actual minimum, we thus do not need to switch from fast gradient descent to any other technique – and this explains why gradient descent is indeed very efficient in machine learning applications.
Acknowledgments

This work was supported in part by the National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science), and HRD-1834620 and HRD-2034030 (CAHSI Includes).

References