PORTADA/UAB.png

Universitat Autònoma de Barcelona

Degree Thesis

Alternatives to the Steepest Gradient Descend in Neural Network training

Author:
Gerard Lahuerta Martín

Supervisor:
Dr. Lluís Alsedà Soler

A thesis submitted in fulfilment of the requirements for the degree of Computational Mathematics and Data Analytics in the

SCIENCE FACULTY

June 2024

Declaration of Authorship

Abstract

Acknowledgements

I would like to express my gratitude to my family for their support throughout my thesis, degree, and my entire life.

I am especially grateful to my mother, Maria Montserrat, for her encouragement and guidance in facing the challenges of this thesis and my career.

I also appreciate the encouragement of my colleagues and the support of my girlfriend.

Finally, I want to give a special mention to Dr. Lluís Alsedà for guiding me through the thesis and being an inspiration.

Thank you all for everything.

Contents

Preface

Since in the early starts of my degree in *Matemàtiques Computacionals i Analítica de Dades*, when one of my professors explains the Gradient Descend method to optimise functions, in every subject about Artificial Intelligence or Neural Networks the method used to obtain the best parameters is always the Steepest Gradient Descend.

This made me think about the nonexistence of an alternative method to optimise Neural Network's parameters, and this thought made me uncomfortable, causing me to reflect on the subject.

No professors could give me an answer to why this is the best way and why it will give the best configuration of the Neural Network's weights, they just told me that this is the fastest way to arrive to the minimum of the *cost function*.

At first thought, the idea that a method whose only purpose is to minimise a value that represents how bad the model predicts seems a good proposition.

Nevertheless, there is no evidence that the results given by the method are the best ones.

I was feeling like the main idea was optimising a function without any information about it, with one algorithm that may or may not give the best result, only a relatively good one; and this way of thinking does not match with me.

This is why I am trying to change the ideology that the Steepest Gradient Descend is the best way to train Neural Network to obtain the best prediction; researching and testing different alternatives of optimization algorithms to train Neural Networks.

The final objective to this thesis is to obtain a better way to arrive to the best configuration; in other words, obtain an algorithm whose results have less error rate in Neural Network predictions (than the one obtained by Steepest Gradient Descend).

Introducción

1 Neural Networks

Before starting programming and testing algorithms in a Neural Network is necessary to understand how Neural Networks work.

A neural Network is made of individual and independent elements connected between them, passing and managing the information through the network formed.

In this thesis we will focus on one of the simplest networks, a multilayer perceptron, to test the different methods of optimization.

1.1 Multilayer Perceptron and Perceptron neuron

One of the simple Neural Networks to analyse and program is the Multilayer Perceptron.

It was first proposed by Frank Rosenblat[1] in 1958 (nevertheless its approach did not learn either produce accurate results).

This Neural Network is formed by elements (the artificial neurons) called *Perceptrons* (which gives its name to this network). This neuron is formed by input, weight and activation functions.

`Neural Network/perceptron_schema.png`

Figure 1: Schema of the Perceptron neuron

The number of inputs that the Perceptron receives is variable but at least needs one input and a biaxe (represented in the picture as the input with value 1).

All the inputs are escalated by a factor (the weights of the Perceptron) and then summed together

Finally, the result of this sum is introduced in the activation function which returns the output of the neuron.

[1]Frank Rosenblat, psychologist and father of deep learning, check its biografy.

The traditional activation function used in the Multilayer Perceptron is the Sigmoid:

$$f(x) = \frac{1}{1 + e^{-w \cdot x}}, \text{ where: } x, w \in \mathbb{R}^n$$

The objective of the activation function is to obtain how relevant the result of the weighted sum is.

The use of the sigmoid function as the activation function allows to represent the output of the perception as a probability (because the image of the sigmoid function is $[0, 1]$).

The Multilayer Perceptron is divided into layers with neurons, we can identify three types of layers:

`Neural Network/Multi-layer-perceptron-MLP-NN-basic-Architecture.png`

Figure 2: Schema of the Multilayer perceptron

- **Input Layer**: The initial set of neurons of the Multilayer Perceptron.
- **Output Layer**: The final set of neurons of the Multilayer Perceptron.
- **Hidden Layers**: The set of neurons (in layers) between the input and output layers.

The number of hidden layers depends on the problem and the number of neurons in each layer could not be the same.

Moreover, mention some vocabulary used in the Thesis in relation to the Neural Networks architectures:

- Fully connected: Architecture with all the neurons connected between layers.
- Deep: Architecture where the number of layers is huge to perform complex regression or classification tasks.

Therefore, we can interpret a Multilayer Perceptron as a weighted sum of activation functions.

But, why a weighted sum of functions can perform complex tasks like classification or regression?

1.2 Why Neural Networks works

This topic is quite extensive, so, to simplify it, we will only consider two tasks: *Classification* and *Regression*.

All other capabilities (such as *Natural Language Processing* or *Computer Vision*) of Neural Networks can be classified in one way or another within this spectrum.

Taking this into consideration, we will now explain a simple interpretation of how Neural Networks work in these tasks.

1.2.1 Neural Networks in Classification Tascks

Let's imagine that we want to classify two hypothetical objects (blue and red ones) based on two relevant attributes: A and B. If we plot these objects on a coordinate axis with attributes A (x-axis) and B (y-axis), one possible resulting graph could be the following one.

Neural Network/red_vs_blue.png

Figure 3: AB attributes graph of red and blue hypothetical objects

In this scenario, one solution to classify the two objects is to divide the AB attribute space (the coordinate axis) in half using the function $r(x) : y = x - 1$, or in other words, by observing the sign of the expression $x - y - 1$.

In this case, the activation function provides information about the probability of an object belonging to one of these two categories.

If we want to classify more objects, we only need to add more 'decision functions,' as they are known. Alternatively, if the distribution of the categories is not linear, it is possible to approximate it with enough linear functions, weighted by the activation function.

One simple example of the last situation could be if the distribution of the red and blue objects were as shown in Figure **??**.

Neural Network/red_vs_blue_case_2.png

Figure 4: Example of non-linear distribution

In this case, we can consider dividing the space between two linear functions: $r(x) : -0.62x + 0.61y = -0.38$ if $x < 2$ and $g(x) : -2.14x + 0.39y = -3.72$ otherwise.

Despite the differences between scenarios, if we want to classify a new object with this space division, the procedures are very similar to the previous one, with just a few extra steps needed.

1. Calculate the value of teach expression: $-0.62x + 0.61y + 0.38$ and $-2.14x + 0.39y + 3.72$.

2. Evaluate each value obtained using the sigmoid activation function and take the average as the probability of belonging to one group.

The final function that calculates the classification (probability to belong to one group) is:

$$P(x, y) = \frac{1}{2}\left(\frac{1}{1 + e^{0.38-0.62x+0.61y}} + \frac{1}{1 + e^{3.72-2.14x+0.39y}}\right)$$

This idea of the Multilayer Perceptron Neural Network have the following architecture:

`Neural Network/diagram_example_mlp.png`

Figure 5: Architecture diagram for the multilayer perception of figure **??**

In this architecture, node $N1$ could have the weights for parameters A (x-axis) and B (y-axis) as $\{-0.62, 0.61\}$ and a bias of 0.38. Node $N2$ would have weights $\{-2.14, 0.39\}$ and a bias of 3.72.

Moreover, node $N3$ will not have a bias (its value is 0), and the weights will be $0.5, 0.5$. In this case, the activation function will be the identity function.

The concept remains the same when dealing with cases involving more attributes or complex decision functions; add more layers and weights to the neural network.

1.2.2 Neural Networks in Regression Tascks

Let's imagine that we want to perform regression on a set of points that exhibit a certain trend, such as a polynomial trend.

In this scenario, it's important to note that any function can be approximated by an infinite sum of sigmoids:

$$f(x) \approx \sum_{i=0}^{N} a_i \sigma(w_i^T \cdot x + b_i) \tag{1}$$

In this context, the function $f : \mathbb{R}^n \to \mathbb{R}^m$ is a continuous function to be approximated, $\sigma : \mathbb{R} \to \mathbb{R}$ is the sigmoid function, w_i represents the weights of the variable $x \in \mathbb{R}^n$, b_i represents the bias, and the parameter a_i is a vector of size $(m, 1)$ containing the weights of the sigmoid function.

This approximation is known as the **Universal Approximation Theorem**, and it enables the Multilayer Perceptron to perform both regression and classification tasks.

1.3 Cost Function

The most relevant application of the **Universal Approximation Theorem** in the field of Neural Networks is that it allows us to approximate cost functions, which are functions that evaluate the errors made by the model.
This approximation enables the model to learn by minimizing the error.

The most used cost functions are:

1. **Mean Squared Error**:
   The Mean Squared Error (*MSE*) is used in regression tasks. It estimates the error of the model. Its expression is:

$$MSE = \frac{1}{2} \sum_{i=1}^{n} (\hat{y_i} - y_i)^2 \tag{2}$$

   Where $\hat{y_i} := f(x_i)$ represents the predicted value of the dependent variable y_i to be estimated, and n is the number of samples in the dataset, where $n \in \mathbb{N}$.

2. **Cross-Entropy Loss**:
   The Cross-Entropy Loss (*CEL*) is used in multi-class classification tasks. It estimates the accuracy of the model. Its expression is:

$$CEL = -\frac{1}{n} \sum_{i=1}^{n} (y_i \cdot log(\hat{y_i}))\,, \text{ where } \hat{y_i} := f(x_i), n \in \mathbb{N} \tag{3}$$

   Where $\hat{y_i}$ is the predicted probability of the *i-th* data classified as the class y_i and $n \in \mathbb{N}$ the number of samples.

Despite the existence of these two cost functions, there are more complex functions that are widely used, such as the **Mean Absolute Error**.

This minimization process (despite the cost function) is used iteratively to determine the values of the parameters in Neural Networks, regardless of the type of task or network.

The use of iterative optimizers to obtain the best parameters for a Neural Network arises due to the complexity of finding the coefficients that minimize the cost function.
Moreover, in some cases, it's possible that such coefficients may not exist or cannot be obtained analytically.

This is why it's important to use differentiable cost functions in training neural networks, as they allow us to leverage the gradient of the cost function to obtain the optimal set of parameters. However, we will discuss this topic in more detail shortly.

1.4 BackPropagation

The importance of using differentiable cost functions lies in their compatibility with the BackPropagation training algorithm. This algorithm relies on the derivatives of the cost function to compute the gradients for each neuron, which are then used in gradient descent optimization.

One widely used example, often employed for educational purposes to practice and understand the algorithm, is the following network:

Neural Network/back-graph.png

Figure 6: Example of BackPropagation algorithm in simple graph

In the example shown in Figure **??**, if we want to modify the value of the parameter a based on the output d, we can apply the chain rule of derivatives, as demonstrated in this case:

$$\frac{\partial D}{\partial a} = \frac{\partial d}{\partial a} + \frac{\partial d}{\partial c} \cdot \frac{\partial c}{\partial a} = c + a = 2a + b$$

This algorithm solves one of the biggest problems in early neural network studies related to their development and training. In practice, when a neural network is used, regardless of the task, the only information available is the input and the output.

This aspect renders the usage of neural networks with more than 2 layers (with more than 1 hidden layer) unviable.

The solution to this problem was the **Back-Propagation** algorithm, which "propagates the effects backwards", through each neuron based on its relevance to the obtained output.

This solution was first introduced by Frank Rosenblatt in 1962 for training an *MLP* (Multi-Layer Perceptron). It was subsequently adopted and used in the training process of various types of neural networks due to its effectiveness, allowing the network to optimize the parameters of each neuron to minimize the error, as represented by the cost function, through **gradient descent**.

2 Optimizers

2.1 Types of Optimizers

2.2 Deterministic Optimizers

2.3 Heuristic Optimizers

2.4 Exploid and Explore philosofy

References

[1] Wikipedia, *Multilayer perceptron*,
https://en.wikipedia.org/wiki/Multilayer_perceptron

[2] Wikipedia, *Perceptrón*,
https://es.wikipedia.org/wiki/Perceptr%C3%B3n

[3] Wikipedia, *Activation function*,
https://en.wikipedia.org/wiki/Activation_function

[4] Wikipedia, *Frank Rosenblatt*,
https://es.wikipedia.org/wiki/Frank_Rosenblatt

[5] Wikipedia, *Activating Function*,
https://en.wikipedia.org/wiki/Activating_function

[6] Wikipedia, *Artificial neural network*,
https://en.wikipedia.org/wiki/Artificial_neural_network

[7] Wikipedia, *Neural network*,
https://en.wikipedia.org/wiki/Neural_network

[8] Wikipedia, *Universal Approximation Theorem*,
https://en.wikipedia.org/wiki/Universal_approximation_theorem

[9] Wikipedia, *Mean Squared Error*,
https://en.wikipedia.org/wiki/Mean_squared_error

[10] Wikipedia, *Mean Absolute Error*,
https://en.wikipedia.org/wiki/Mean_absolute_error

[11] Wikipedia, *Cross-Entropy*,
https://en.wikipedia.org/wiki/Cross-entropy

[12] Wikipedia, *BackPropagation*,
https://en.wikipedia.org/wiki/Backpropagation

[13] Dantzig, G.B. y P. Wolfe, «Decomposition principle for linear programs», *Operations Research*, **8**, págs. 101–111, 1960.