Abdelwahid (Wahid) Benslimane
wahid.benslimane@gmail.com

The Non-Convexity of Loss Functions with Respect to Model Parameters in Neural Networks

Introduction

In this document, I aim to clarify a common misunderstanding among those new to machine learning or AI. It concerns gradient descent, the core optimization method used to train neural networks and find the best values for their many parameters. However, as I will explain, these values are not always truly optimal. Many assume that gradient descent always finds the global minimum, but in reality, it often only reaches a local minimum, and I will outline the reasons.

For those looking for a refresher, you can refer to the following documents I've written on gradient descent and gradient backpropagation (though a deep understanding of backpropagation isn't required to follow this document).

Gradient descent with optimal variable step size:
https://github.com/abenslimaneakawahid/iterative-methods/blob/main/Gradient_desc_opt_var_step_size.pdf

Conjugate gradient method:
https://github.com/abenslimaneakawahid/iterative-methods/blob/main/Conjugate_gradient.pdf

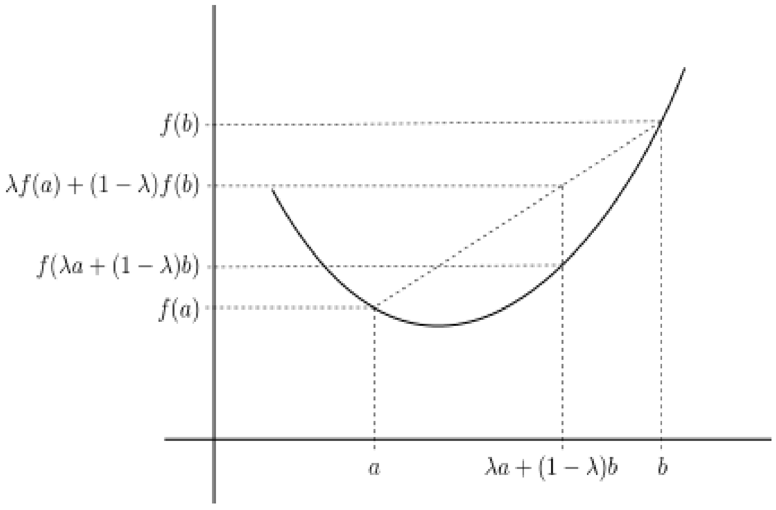
Gradient backpropagation: detailed explanation:
https://medium.com/@wahid.benslimane/gradient-backpropagation-detailed-explanation-c7db8d60aaa2

The loss function in a machine learning model quantifies the error between the model's predictions and the actual target values. This measure is typically represented using a convex function. However, when expressed in terms of the model's parameters, this function is often non-convex in complex systems, such as neural networks.

For the recall, the convexity of a function describes its shape. A function is convex if it "bends upwards," resembling a bowl. If you plot a curve, it is convex if the line connecting any two points on it always stays above or on the curve. A common example is the parabola $y=x^2$.

In high dimensions (more than three) , when it is not possible to visualize the function, a function is still convex if, for any two points a and b, the value at any point along the line connecting them is less than or equal to the average of their values. This can be expressed as:

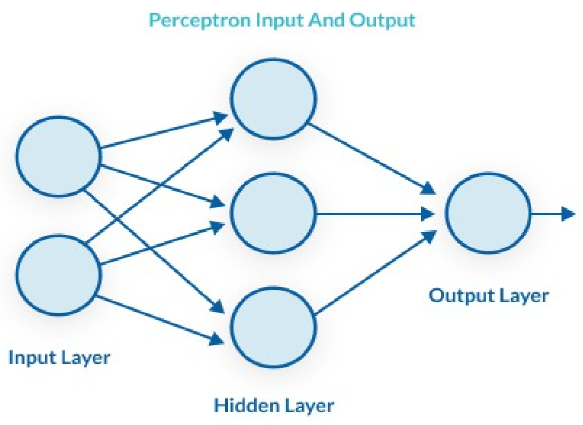
$$f(\lambda \cdot a+(1-\lambda)\cdot b)\leq \lambda \cdot f(a)+(1-\lambda)\cdot f(b), \lambda \in [0,1]$$



Non-linearity in the Parameters

In simple models like linear regression, the loss function, such as mean squared error (MSE), is convex because the relationship between the model parameters and the output is linear. A convex function has a single valley (global minimum), and gradient descent will ensure that this optimum is found. However, in more complex models, such as neural networks or deep learning models more broadly, the transformations applied to the data (like nonlinear activation functions) make the loss function highly nonlinear. This creates complex landscapes with multiple valleys, peaks, and plateaus, leading to numerous local minima and saddle points. In this kind of model, the parameters (weights and biases) are distributed across multiple layers. Each nonlinear layer transforms the input before passing it to the next layer. As a result, the relationship between the parameters of the different layers and the model's final output is nonlinear and complex.

Let’s consider a simple neural network, specifically a multilayer perceptron with a single hidden layer and a standard loss function like MSE. I will illustrate why the loss function is not convex.



Let’s assume the model is caracterized by:

- An input $x \in \mathbb{R}^n$
- One hidden layer with a nonlinear activation functionn σ (for example, ReLU or Sigmoid)

- An output $\hat{y} \in R$

The model can be formulated as:

$$\hat{y} = W_2 \cdot \sigma(W_1 \cdot x + b_1) + b_2$$

Where:

- $W_1 \in R^{m \times n}$ represents the weights between the input and the hidden layer.
- $b_1 \in \mathbb{R}^m$ is the bias of the hidden layer.
- $W_2 \in \mathbb{R}^m$ represents the weights between the hidden layer and the output.
- $b_2 \in \mathbb{R}$ is the bias of the output.
- $\sigma(.)$ is a nonlinear activation function, such as the sigmoid function :

  $$\sigma(z) = \frac{1}{1+e^{-z}}$$

The loss function is generally defined as the measurement of the discrepancy between the model's prediction $\hat{y}$ and the true target value y. For a dataset with N examples, the MSE is given by:

$$L(W_1, W_2, b_1, b_2) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

Where $\hat{y}_i$ is the prediction for example i, which depends nonlinearly on the parameters W_1, W_2, b_1, and b_2 because of the activation function σ and matrix multiplication.

Now, let's consider the gradient of the loss function:

$$\frac{\partial L}{\partial W_1} = \frac{2}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i) \cdot \frac{\partial \hat{y}_i}{\partial W_1}$$

If we calculate $\frac{\partial \hat{y}_i}{\partial W_1}$, we get:

$$\frac{\partial \hat{y}_i}{\partial W_1} = W_2 \cdot \sigma'(W_1 \cdot x_i + b_1) \cdot x_i \quad \text{where} \quad \sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \quad \text{for the sigmoid function,}$$

This derivative is complex and nonlinear with respect to W_1, introducing non-convexity. This results in a loss landscape with local minima, saddle points, and plateaus, which complicates optimization.

Regularization

Regularization techniques, like L2 regularization or dropout, impose penalties on the model's parameters to mitigate overfitting, helping the model generalize better to unseen data. While these techniques improve generalization, they also complicate the structure of the loss function. L2 regularization adds a penalty proportional to the sum of the squared values of the parameters, discouraging the model from relying too heavily on any single weight. Dropout, on the other hand, randomly disables a fraction of neurons during training, preventing the model from becoming too specialized.

The introduction of regularization terms adds new components to the optimization problem, altering the loss surface by creating additional slopes, valleys, and peaks. These terms are not independent and vary with the model's parameters, meaning that the optimization process has to account for both the original loss and the regularization penalties. This increased complexity contributes to the non-convexity of the loss function, making it more challenging to find the global minimum during training. The optimization process must now navigate a more intricate landscape with potentially more local minima and saddle points.

Stochastic Optimization for Better Exploration

Stochastic optimization introduces randomness into the optimization process, which helps the algorithm avoid getting stuck in local minima and increases the chances of finding the global minimum of the loss function. A deterministic method like gradient descent may get trapped in a local minimum, but stochastic gradient descent (SGD), by using random subsets of data (mini-batches), introduces variability in the updates. This randomness enables the algorithm to "jump" out of local minima, explore more of the solution space, and ultimately move closer to the global minimum.

Key advantages of stochastic optimization include lower computational costs due to frequent updates with smaller data subsets, which help the algorithm escape local minima by introducing randomness that prevents it from getting trapped. This approach is particularly efficient for large datasets, where calculating a full gradient at each step is impractical. However, there are drawbacks: unstable convergence, as random fluctuations can reduce precision near the global minimum, necessitating adjustments like reducing the learning rate, and noise, where the inherent randomness can compromise the accuracy of gradient estimates.

This balance between randomness and efficiency makes stochastic optimization well-suited for large, complex models, increasing the likelihood of finding the global minimum while keeping computational costs manageable.