How neural networks learn from errors.

An intuitive understanding of the backpropagation process.

Josue Valenzuela Perez
jos.val.pe07@gmail.com
GitHub: https://github.com/JosueVP17

March 2026

1 Introduction.

Artificial Neural Networks have become a fundamental tool in modern machine learning, enabling the modeling of complex, non-linear relationships across a wide range of applications. At the core of training these models lies the backpropagation algorithm, which provides an efficient way to compute gradients of a loss function with respect to the network parameters.

This document presents a detailed derivation of the backpropagation algorithm, starting from the forward pass and progressing through the application of the chain rule to obtain parameter updates. Special attention is given to the mathematical structure of each layer, including the role of weights, biases, and activation functions, as well as the propagation of error signals throughout the network.

In addition, the document explores the use of the softmax activation function in combination with the cross-entropy loss for multi-class classification tasks. This combination provides a probabilistic interpretation of the network outputs and leads to a significant simplification in the gradient computation of the output layer.

The objective of this work is to provide an intuitive understanding of backpropagation, bridging the gap between theoretical formulation and practical implementation. By the end of this document, the reader should be able to clearly understand how gradients are computed and how they drive the learning process in neural networks.

2 Background.

The goal is to compute the gradient of the loss function with respect to the weights and biases for a single input-output example, with the objective of minimize the loss. Recall that the gradient of a differentiable function F is the vector field ∇F whose value at a point $t = (t_1, \ldots, t_n)$ from its domain gives the direction of the fastest increase. The gradient is given by the vector whose components are the partial derivatives of F at t:

$$\nabla \mathcal{L}(t) = \left[\frac{\partial \mathcal{L}}{\partial t_1}(t), \frac{\partial \mathcal{L}}{\partial t_2}(t), \ldots, \frac{\partial \mathcal{L}}{\partial t_n}(t) \right]$$

If we use the individual components of the gradient vector of the function, each partial derivative will tell us how to update a certain weight or bias. Since the gradient gives the direction of the fastest increase, we need to set its negative value to go in the other direction, the fastest decrease. Therefore, the learning rules for the weights and biases of the network are the following:

$$W_{new} = W_{old} - \eta \cdot \frac{\partial \mathcal{L}}{\partial W}$$

$$b_{new} = b_{old} - \eta \cdot \frac{\partial \mathcal{L}}{\partial b}$$

Where:

- $\mathcal{L}$: is the loss function.
- η: is the learning rate.
- $\frac{\partial \mathcal{L}}{\partial w}$ and $\frac{\partial \mathcal{L}}{\partial b}$: are the partial derivatives of the loss with respect to the parameters.

The loss function we will use here is the mean squared error:

$$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$$

Where y is the ground truth value and $\hat{y}$ is the predicted value. The activation function will be the sigmoid function, denoted as:

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

3 Three-layer architecture.

Here is a simple architecture that can be used to classify a 2-dimensional input, for example (x_0, y_0), into a binary class.

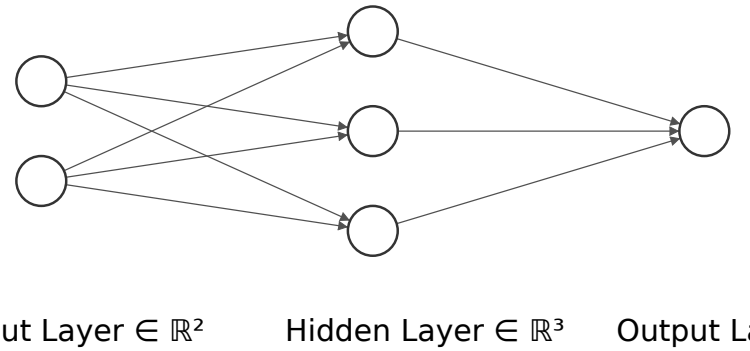

Figure 1: Two inputs, 3 neurons in hidden layer and one output.

Let $x \in \mathbb{R}^{2\times 1}$ be the input vector of the network, the weight matrix $W^{(1)} \in \mathbb{R}^{3\times 2}$ and the bias vector $b^{(1)} \in \mathbb{R}^{3\times 1}$. The weighted sum for the first layer is:

$$\underbrace{z^{(1)}}_{3\times 1} = W^{(1)}x + b^{(1)}$$

Where the superscript represents the number layer (0 for the input layer). Its activation value in the sigmoid function is:

$$\underbrace{a^{(1)}}_{3\times 1} = \sigma(z^{(1)})$$

We can express the same terms for the output layer, with $W^{(2)} \in \mathbb{R}^{1\times 3}$ and $b^{(2)} \in \mathbb{R}$. Since the output layer is fed with the activation $a^{(1)}$ of the first layer, we have the following.

$$\underbrace{z^{(2)}}_{1\times 1} = W^{(2)}a^{(1)} + b^{(2)}$$

$$\underbrace{a^{(2)}}_{1\times 1} = \sigma(z^{(2)}) = \hat{y}$$

Let us compute the rate of change $\frac{\partial \mathcal{L}}{\partial W^{(2)}}$ for the learning rule for $W^{(2)}$. How can this be done? Well, using the chain rule from calculus! This is effective since $W^{(2)}$ affects the weighted sum, which changes the activation function, which has a certain loss. Therefore:

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

We can now individually compute each partial derivative:

$$\frac{\partial \mathcal{L}}{\partial a^{(2)}} = -(y - \hat{y})$$

$$\frac{\partial a^{(2)}}{\partial z^{(2)}} = \sigma'(z^{(2)}) = \sigma(z^{(2)})(1 - \sigma(z^{(2)})) = a^{(2)}(1 - a^{(2)}) = \hat{y}(1 - \hat{y})$$

$$\frac{\partial z^{(2)}}{\partial W^{(2)}} = (a^{(1)})^T$$

Notice how a transpose is set for the last partial derivative. You make your mind think that it comes from nowhere, but it establishes congruence when putting it all together. This manipulation is executed in the same fashion for the hidden layer. Hence, the gradient is as follows:

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \underbrace{-(y - \hat{y})\hat{y}(1 - \hat{y})}_{1\times 1} \underbrace{(a^{(1)})^T}_{1\times 3}$$

We can set $\delta^{(2)} \in \mathbb{R}$ as a local error for $W^{(2)}$, then:

$$\delta^{(2)} = \frac{\partial \mathcal{L}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} = \frac{\partial \mathcal{L}}{\partial z^{(2)}}$$

$$\underbrace{\frac{\partial \mathcal{L}}{\partial W^{(2)}}}_{1\times 3} = \underbrace{\delta^{(2)}}_{1\times 1} \cdot \underbrace{(a^{(1)})^T}_{1\times 3}$$

And we can compute the learning updates for $W^{(2)}$. It is pretty similar for the bias as well, since only the last term changes in the chain rule:

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \frac{\partial \mathcal{L}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}}$$

$$\frac{\partial z^{(2)}}{\partial b^{(2)}} = 1$$

Therefore, the local gradient for for $b^{(2)}$ can be simply stated as:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial b^{(2)}}}_{1\times 1} = \underbrace{\delta^{(2)}}_{1\times 1}$$

We can follow the same logic for the hidden layer local gradient, although it will have more terms since it causes an additional weighted sum and activation, in addition with the previous terms discussed. Note that $\frac{\partial z^{(2)}}{\partial W^{(2)}}$ is not used here, as we cannot complete the chain rule based on $W^{(1)}$.

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial W^{(1)}}$$

$$\frac{\partial z^{(2)}}{\partial a^{(1)}} = (W^{(2)})^T$$

$$\frac{\partial a^{(1)}}{\partial z^{(1)}} = \sigma(z^{(1)})(1-\sigma(z^{(1)})) = a^{(1)}(1-a^{(1)})$$

$$\frac{\partial z^{(1)}}{\partial W^{(1)}} = x^T$$

In this layer, we have a weight matrix of 3×2 dimension, then, the chain rule to be correct, it is necessary that each neuron weight is updated according to its own error. Consequently, it is mandatory to use the Hadamard product $\odot$ (multiplication element by element), instead of a normal matrix multiplication. Note that this is not necessary in the last layer, since we worked with vectors and normal multiplication already provides the entire gradient. It is only used for number of layers greater than 1. Consequently, the local gradient is:

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \underbrace{\delta^{(2)}}_{1\times 1} \cdot \underbrace{(W^{(2)})^T}_{3\times 1} \odot \underbrace{a^{(1)}}_{3\times 1} \odot \underbrace{(1-a^{(1)})}_{3\times 1} \cdot \underbrace{x^T}_{1\times 2}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \underbrace{\left(\delta^{(2)} \cdot (W^{(2)})^T \odot a^{(1)} \odot (1-a^{(1)})\right)}_{3\times 1} \cdot \underbrace{x^T}_{1\times 2}$$

If we again set the entire term as a local error, which in this case is $\delta^{(1)} \in \mathbb{R}^{3\times 1}$, then the local gradient simplifies as:

$$\delta^{(1)} = \frac{\partial \mathcal{L}}{\partial z^{(1)}}$$

$$\underbrace{\frac{\partial \mathcal{L}}{\partial W^{(1)}}}_{3\times 2} = \underbrace{\delta^{(1)}}_{3\times 1} \underbrace{x^T}_{1\times 2}$$

Regarding $b^{(1)}$, the last term from its local gradient is again the unit $\dfrac{\partial z^{(1)}}{\partial b^{(1)}} = 1$, whereby it again follows the structure:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial b^{(1)}}}_{3\times 1} = \underbrace{\delta^{(1)}}_{3\times 1}$$

At this moment, we can notice that terms follow a pattern.

4 L-layer architecture for binary classification.

First, let us define the terms for each layer l. For a neural network with L layers, we define the parameters of the layer l as:

- n_l: Number of neurons in the layer l. For $n_L = 1$.
- $W^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$: Weight matrix.
- $b^{(l)} \in \mathbb{R}^{n_l \times 1}$: Bias vector.
- $a^{(l)} \in \mathbb{R}^{n_l \times 1}$: Activation vector. For layer 0 is $a^{(0)} = x$ and for layer L is $a^{(L)} = \hat{y}$.
- $z^{(l)} \in \mathbb{R}^{n_l \times 1}$: Weighted sum vector. For layer l is $z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$.

The backpropagation process begins in the output layer L. For this layer, the local error $\delta^{(L)}$ for any loss function $\mathcal{L}$ and for any activation function $f^{(L)}$ is

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$$

$$= \frac{\partial \mathcal{L}}{\partial a^{(L)}} \odot f'^{(L)}(z^{(L)}) = \frac{\partial \mathcal{L}}{\partial \hat{y}} \odot f'^{(L)}(z^{(L)})$$

For any other layer l, using any activation function $f^{(l)}$, its local error is defined as

$$\delta^{(l)} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}}$$

$$= \delta^{(l+1)} \odot (W^{(l+1)})^T \odot f'^{(l)}(z^{(l)})$$

Depending on the election of the activation function $f^{(l)}(z)$, its derivative $f'^{(l)}(z)$ may take various forms:

- **Sigmoid:** $f'^{(l)}(z^{(l)}) = \sigma(z^{(l)})(1 - \sigma(z^{(l)})) = a^{(l)} \odot (1 - a^{(l)})$
- **Tanh:** $f'^{(l)}(z^{(l)}) = 1 - \tanh^2(z^{(l)}) = 1 - (a^{(l)})^2$
- **ReLU:** $f'^{(l)}(z^{(l)}) = \begin{cases} 1 & \text{if } z^{(l)} > 0 \\ 0 & \text{if } z^{(l)} \leq 0 \end{cases}$

Generalizing, the local gradients with respect to the parameters of any layer l are:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$

$$\frac{\partial \mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}$$

5 L-layer architecture for multi-classification.

We can use the same terms for each layer l as before to describe this architecture. Imagine that the network does not have 1 weighted sum output, but C weighted sums outputs, (one per class):

$$z^{(L)} = [z_1, z_2, \ldots, z_C]$$

The output layer typically employs the softmax activation function, which transforms this raw output into a probability distribution over the C classes, which at the same time is the number of neurons n_L in the output layer. The probability of the i-th neuron from the output vector $\hat{y}$ is assigned by:

$$\hat{y_i} = \frac{e^{z_i}}{\sum_{k=1}^{n_L} e^{z_k}}$$

This ensures that each output lies in the interval $[0, 1]$ and that all outputs sum to one, making them interpretable as probabilities. To match this definition, it is necessary that the expected output y of the class is in the form of a vector encoded with One-Hot. Let $c \in 1, \ldots, C$ be the real class of a sample. The One-Hot vector, $y \in \{0, 1\}^C$, is defined from component to component as:

$$y_i = \delta_{ic} = \begin{cases} 1 & \text{if } i = c \\ 0 & \text{if } i \neq c \end{cases}$$

Where δ_{ic} is the Kronecker Delta function.

An important property of this vector is that the sum of its elements is always 1.

$$\sum_{i=1}^{C} y_i = 1$$

To measure the discrepancy between the predicted distribution $\hat{y}$ and the true distribution y, the cross-entropy loss is used:

$$\mathcal{L} = -\sum_{k=1}^{n_L} y_k \log(\hat{y}_k)$$

This loss function strongly penalizes confident but incorrect predictions, while assigning a low penalty when the predicted probability for the correct class is high.

Good prediction case:

- **Ground truth:** $y = [1, 0, 0]$
- **Prediction:** $\hat{y} = [0.90, 0.05, 0.05]$
- **Loss:** $\mathcal{L} \approx 0.105$

Bad prediction case:

- **Ground truth:** $y = [1, 0, 0]$
- **Prediction:** $\hat{y} = [0.10, 0.80, 0.10]$
- **Loss:** $\mathcal{L} \approx 2.302$

All gradients for layers $\in [1, \ldots, L-1]$ remain the same as **Section 4**, but the goal here is to compute $\frac{\partial \mathcal{L}}{\partial W^{(L)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(L)}}$for the last layer. Let us begin with the weights, with chain rule again:

$$\frac{\partial \mathcal{L}}{\partial W^{(L)}} = \frac{\partial \mathcal{L}}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial W^{(L)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial W^{(L)}}$$

Since we have a weighted sum vector and softmax activation produces a vector, then $\frac{\partial \hat{y}}{\partial z^{(L)}}$ is not a vector, but a matrix. This is the Jacobian matrix, which maps the interdependence between outputs. The Jacobian $\mathbf{J} \in \mathbb{R}^{C \times C}$ has the partial derivatives of each output with respect to each input:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \hat{y}_1}{\partial z_1} & \cdots & \frac{\partial \hat{y}_1}{\partial z_C} \\ \vdots & \ddots & \vdots \\ \frac{\partial \hat{y}_C}{\partial z_1} & \cdots & \frac{\partial \hat{y}_C}{\partial z_C} \end{bmatrix}$$

Which is basically a vector of gradients:

$$\mathbf{J} = [\nabla \hat{y}_1, \nabla \hat{y}_2, \ldots, \nabla \hat{y}_C]$$

Let us derive $\frac{\partial \hat{y}}{\partial z^{(L)}}$. From a fixed activation $\hat{y}_i$ in softmax, we can define an auxiliar variable S:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{k=1}^{n_L} e^{z_k}} = \frac{e^{z_i}}{S}$$

Deriving for the k activation using the division rule:

$$\frac{\partial}{\partial z_k} \left(\frac{e^{z_i}}{S} \right) = \frac{\frac{\partial}{\partial z_k}(e^{z_i}) \cdot S - e^{z_i} \cdot \frac{\partial S}{\partial z_k}(S)}{S^2}$$

For $\frac{\partial}{\partial z_k}(e^{z_i})$ depends whether $i = k$ or not:

If $i = k$:

$$\frac{\partial}{\partial z_k}(e^{z_i}) = e^{z_i}$$

If $i \neq k$:

$$\frac{\partial}{\partial z_k}(e^{z_i}) = 0$$

This can be written with Kronecker delta:

$$\frac{\partial}{\partial z_k}(e^{z_i}) = e^{z_i} \cdot \delta_{ik}$$

For $\frac{\partial}{\partial z_k}(S)$ only one term depends on z_k:

$$\frac{\partial}{\partial z_k}(S) = e^{z_k}$$

Substituting back:

$$\frac{\partial}{\partial z_k}\left(\frac{e^{z_i}}{S}\right) = \frac{e^{z_i} \cdot \delta_{ik} \cdot S - e^{z_i} \cdot e^{z_k}}{S^2}$$
$$= \frac{e^{z_i}(\delta_{ik} \cdot S - e^{z_k})}{S^2}$$

Rewriting in terms of softmax:

$$\hat{y}_i = \frac{e^{z_i}}{S} \longrightarrow e^{z_i} = \hat{y}_i \cdot S$$
$$\hat{y}_k = \frac{e^{z_k}}{S} \longrightarrow e^{z_k} = \hat{y}_k \cdot S$$

$$\frac{\partial}{\partial z_k}\left(\frac{e^{z_i}}{S}\right) = \frac{\hat{y}_i \cdot S(\delta_{ik} \cdot S - \hat{y}_k \cdot S)}{S^2} = \hat{y}_i(\delta_{ik} - \hat{y}_k)$$

Therefore, the value of the Jacobian matrix for any position (i, k) is:

$$\mathbf{J}_{ik} = \hat{y}_i(\delta_{ik} - \hat{y}_k)$$

Which has two cases:

If $i = k$ (diagonal):

$$\mathbf{J}_{ii} = \hat{y}_i(1 - \hat{y}_i)$$

If $i \neq k$ (out of diagonal):

$$\mathbf{J}_{ik} = -\hat{y}_i\hat{y}_k$$

We can now compute $\frac{\partial \mathcal{L}}{\partial W^{(L)}}$. Let us begin with the analysis for the loss with respect to the weighted sum; then we can analyze for the weights per se. For the weighted sum z_i:

$$\frac{\partial \mathcal{L}}{\partial z_i^{(L)}} = \sum_{k=1}^{n_L} \frac{\partial \mathcal{L}}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_i^{(L)}}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_k} = \frac{\partial}{\partial \hat{y}_k}\left(-\sum_{k=1}^{n_L} y_k \log(\hat{y}_k)\right) = -\frac{y_k}{\hat{y}_k}$$

Substituting back:

$$\frac{\partial \mathcal{L}}{\partial z_i^{(L)}} = \sum_{k=1}^{n_L} \left(-\frac{y_k}{\hat{y}_k}\right) \frac{\partial \hat{y}_k}{\partial z_i^{(L)}}$$

Let us separate the summation into two parts to simplify $\frac{\partial \hat{y}_k}{\partial z_i^{(L)}}$:

When $i = k$:

$$\left(-\frac{y_i}{\hat{y}_i}\right)\hat{y}_i(1-\hat{y}_i) = -y_i(1-\hat{y}_i)$$

When $i \neq k$:

$$\sum_{k=1;i\neq k}^{n_L}\left(-\frac{y_k}{\hat{y}_k}\right)\cdot(-\hat{y}_i\hat{y}_k) = \sum_{k=1;i\neq k}^{n_L} y_k\hat{y}_i = \hat{y}_i\sum_{k=1;i\neq k}^{n_L} y_k$$

Using the property for a One-Hot vector:

$$\sum_{k=1}^{n_L} y_k = 1 \longrightarrow \sum_{k=1;i\neq k}^{n_L} y_k = 1 - y_i$$

Therefore:

$$\hat{y}_i\sum_{k=1;i\neq k}^{n_L} y_k = \hat{y}_i(1-y_i)$$

Now, summing both cases:

$$\frac{\partial\mathcal{L}}{\partial z_i^{(L)}} = -y_i(1-\hat{y}_i) + \hat{y}_i(1-y_i)$$

$$= -y_i + y_i\hat{y}_i + \hat{y}_i - y_i\hat{y}_i = \hat{y}_i - y_i$$

This result avoids computing the full Jacobian of the softmax function. Since this is pretty simplified, we can start using vectorized notation as in previous sections. As stated in **Section 4**, we can set a local error $\delta^{(L)}$ specifically for this output layer, following exactly the same structure:

$$\delta^{(L)} = \frac{\partial\mathcal{L}}{\partial\hat{y}}\frac{\partial\hat{y}}{\partial z^{(L)}} = \hat{y} - y$$

Hence, generalizing for any layer, the local gradients for the parameters follow the same structure as before.

$$\frac{\partial\mathcal{L}}{\partial W^{(l)}} = \delta^{(l)}a^{(l-1)}$$

$$\frac{\partial\mathcal{L}}{\partial b^{(l)}} = \delta^{(l)}$$