

Already have an account? Sign in

# One LEGO at a Time: Explaining the Math of how Neural Networks Learn with Implementation from Scratch



Omar U. Florez

Follow

Jun 1 · 9 min read \*

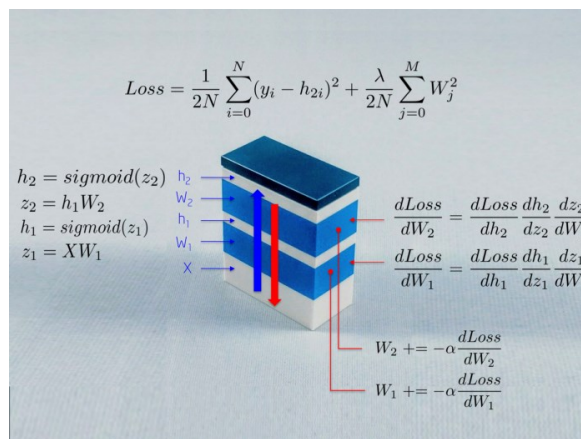
A **neural network** is a clever arrangement of linear and non-linear modules. When we choose and connect them wisely, we have a powerful tool to approximate any mathematical function. For example one that **separates classes with a non-linear decision boundary**.

Steps to run the code:

- git clone [https://github.com/omar-florez/scratch\\_mlp/](https://github.com/omar-florez/scratch_mlp/)
- python scratch\_mlp/scratch\_mlp.py

A topic that is not always explained in-depth, despite its intuitive and modular nature, is the **backpropagation technique** responsible for updating trainable parameters. Let's build a neural network from scratch to see the internal functioning of a neural network using **LEGO pieces as a modular analogy**, one brick at a time.

## Neural Networks as a Composition of Pieces



The above figure depicts some of the Math used for training a neural network. We will make sense of this during this article. The reader may find interesting that a neural network is a stack of modules with different purposes:

- **Input X** feeds a neural network with raw data, which is stored in a matrix in which observations are rows and dimensions are columns
- **Weights W1** maps input X to the first hidden layer h1. Weights W1 works then as a linear kernel
- A **Sigmoid function** prevents numbers in the hidden layer from falling out of range by scaling them to 0–1. The result is an **array of neural activations**  $h1 = \text{Sigmoid}(WX)$

At this point, these operations only compute a **general linear system**, which doesn't have the capacity to model non-linear interactions. This changes when we stack one more layer, adding depth to this modular structure. The deeper the network, the more subtle non-linear interactions we can learn and more complex problems we can solve, which may explain in part the rise of deep neural models.

## Why should I read this?

If you understand the internal parts of a neural network, you will quickly know **what to change first** when things don't work and define an strategy

to **test invariants** and **expected behaviors** that you know are part the algorithm. This will also be helpful when you want to **create new capabilities that are not currently implemented in the ML library** you are using.

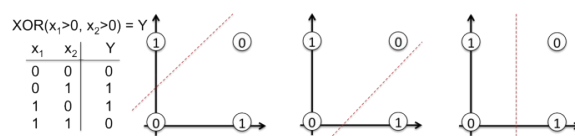
**Because debugging machine learning models is a complex task.** By experience, mathematical models don't work as expected the first try. They may give you low accuracy for new data, spend long training time or too much memory, return a large number of false negatives or NaN predictions, etc. Let me show some cases when knowing how the algorithm works can become handy:

- If it **takes so much time to train**, it is maybe a good idea to increase the size of a minibatch to reduce the variance in the observations and thus to help the algorithm to converge
- If you observe **NaN predictions**, the algorithm may have received large gradients producing memory overflow. Think of this as consecutive matrix multiplications that explode after many iterations. Decreasing the learning rate will have the effect of scaling down these values. Reducing the number of layers will decrease the number of multiplications. And clipping gradients will control this problem explicitly

## Concrete Example: Learning the XOR Function

Let's open the blackbox. We will build now a neural network from scratch that learns the **XOR function**. The choice of this **non-linear function** is by no means random chance. Without back-propagation it would be hard to learn to separate classes with a **straight line**.

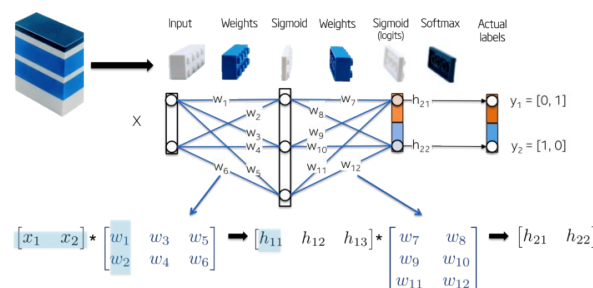
To illustrate this important concept, note below how a straight line cannot separate 0s and 1s, the outputs of the XOR function. **Real-life problems are also non-linearly separable**.



The topology of the network is simple:

- **Input X** is a two-dimensional vector
- **Weights W1** is a 2x3 matrix with randomly initialized values
- **The hidden layer h1** consists of three neurons. Each neuron receives as input a weighted sum of observations, this is the inner product highlighted in green in the below figure:  **$z1 = [x1, x2][w1, w2]$**
- **Weights W2** is a 3x2 matrix with randomly initialized values and
- **Output layer h2** consists of two neurons, since the XOR function returns either 0 ( $y1 = [0,1]$ ) or 1 ( $y2 = [1,0]$ )

More visually:

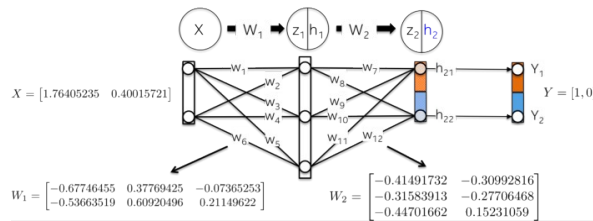


Let's now train the model. In our simple example the trainable parameters are weights, but be aware that current research is exploring more types of parameters to be optimized. For example shortcuts between layers, regularized distributions, topologies, residual, learning rates, etc.

**Backpropagation** is a method to update the weights towards the direction (**gradient**) that minimizes a predefined error metric known as **Loss function given** a batch of labeled observations. This algorithm has been repeatedly rediscovered and is a special case of a more general technique called automatic differentiation in reverse accumulation mode.

## Network Initialization

Let's **initialize the network weights** with random numbers.



## Forward Step:

This goal of this step is to **forward propagate** the input  $X$  to each layer of the network until computing a vector in the output layer  $h_2$ .

This is how it happens:

- Linearly map input data  $X$  using weights  $W_1$  as a kernel:

$$z_1 = XW_1$$

$$z_1 = [1.76405235 \quad 0.40015721] \begin{bmatrix} -0.67746455 & 0.37769425 & -0.07365253 \\ -0.53663519 & 0.60920496 & 0.21149622 \end{bmatrix}$$

$$z_1 = [-1.40982136 \quad 0.91005018 \quad -0.04529517]$$

- Scale this weighted sum  $z_1$  with a Sigmoid function to get values of the first hidden layer  $h_1$ . **Note that the original 2D vector is now mapped to a 3D space.**

$$h_1 = \text{sigmoid}(z_1)$$

$$h_1 = [0.19626223 \quad 0.71301043 \quad 0.48867814]$$

- A similar process takes place for the second layer  $h_2$ . Let's compute first the **weighted sum**  $z_2$  of the first hidden layer, which is now input data.

$$z_2 = h_1W_2$$

$$z_2 = [0.19626223 \quad 0.71301043 \quad 0.48867814] \begin{bmatrix} -0.41491732 & -0.30992816 \\ -0.31583913 & -0.27706468 \\ -0.44701662 & 0.15231059 \end{bmatrix}$$

$$z_2 = [-0.52507645 \quad -0.18394635]$$

- And then compute their Sigmoid activation function. This vector  $[0.37166596 \quad 0.45414264]$  represents the **log probability** or **predicted vector** computed by the network given input  $X$ .

$$h_2 = \text{sigmoid}(z_2)$$

$$h_2 = [0.37166596 \quad 0.45414264]$$

## Computing the Total Loss

Also known as "actual minus predicted", the goal of the loss function is to **quantify the distance between the predicted vector  $h_2$  and the actual label provided by humans  $y$ .**

Note that the Loss function contains a **regularization component** that penalizes large weight values as in a Ridge regression. In other words, large squared weights values will increase the Loss function, **an error metric we indeed want to minimize.**

$$Loss = \frac{1}{2N} \sum_{i=0}^N (y_i - h_{2i})^2 + \frac{\lambda}{2N} \sum_{j=0}^M W_j^2$$

## Backward step:

The goal of this step is to **update the weights of the neural network** in a direction that minimizes its Loss function. As we will see, this is a **recursive algorithm**, which can reuse gradients previously computed and heavily relies on **differentiable functions**. Since these updates reduce the loss

function, a network 'learns' to approximate the label of observations with known classes. A property called **generalization**.

This step goes in **backward order** than the forward step. It computes first the partial derivative of the loss function with respect to the weights of the output layer ( $dLoss/dW_2$ ) and then the hidden layer ( $dLoss/dW_1$ ). Let's explain in detail each one.

### **dLoss/dW2:**

The chain rule says that we can decompose the computation of gradients of a neural network into **differentiable pieces**:

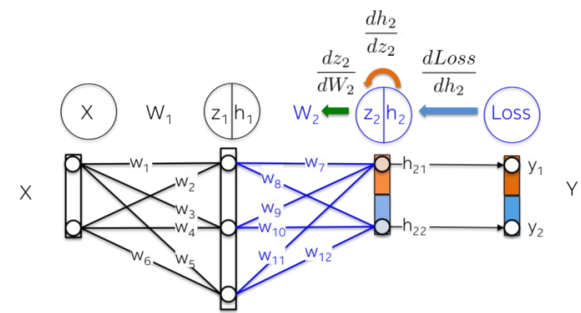
$$\frac{dLoss}{dW_2} = \frac{dLoss}{dh_2} \frac{dh_2}{dz_2} \frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_2} = -(y - h_2) \quad \frac{dh_2}{dz_2} = h_2(1 - h_2) \quad \frac{dz_2}{dW_2} = h_1$$

As a memory helper, these are the **function definitions** used above and their **first derivatives**:

Function	First derivative
Loss = $(y - h_2)^2$	$dLoss/dW_2 = -(y - h_2)$
$h_2 = \text{Sigmoid}(z_2)$	$dh_2/dz_2 = h_2(1 - h_2)$
$z_2 = h_1 W_2$	$dz_2/dW_2 = h_1$
$z_2 = h_1 W_2$	$dz_2/dh_1 = W_2$

More visually, we aim to update the weights  $W_2$  (in blue) in the below figure. In order to that, we need to compute three **partial derivatives along the chain**.



Plugging in values into these partial derivatives allow us to compute gradients with respect to weights  $W_2$  as follows.

$$\frac{dLoss}{dW_2} = \frac{dLoss}{dh_2} \frac{dh_2}{dz_2} \frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_2} = -(y - h_2) \quad \frac{dh_2}{dz_2} = h_2(1 - h_2) \quad \frac{dz_2}{dW_2} = h_1$$

$$\frac{dLoss}{dh_2} = -(y - h_2) = -\begin{bmatrix} 1 & 0 \end{bmatrix} - \begin{bmatrix} 0.37166596 & 0.45414264 \end{bmatrix} = \begin{bmatrix} -0.62833404 & 0.45414264 \end{bmatrix}$$

$$\frac{dh_2}{dz_2} = h_2(1 - h_2) = \begin{bmatrix} 0.37166596 & 0.45414264 \end{bmatrix} (1 - \begin{bmatrix} 0.37166596 & 0.45414264 \end{bmatrix}) = \begin{bmatrix} -0.23353037 & 0.2478971 \end{bmatrix}$$

$$\frac{dz_2}{dW_2} = h_1 = \begin{bmatrix} 0.19626223 & 0.71301043 & 0.48867814 \end{bmatrix}$$

The result is a 3x2 matrix  $dLoss/dW_2$ , which will update the original  $W_2$  values in a direction that minimizes the Loss function.

$$\frac{dLoss}{dW_2} = \frac{dLoss}{dh_2} \frac{dh_2}{dz_2} \frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_2} = -(y - h_2) \quad \frac{dh_2}{dz_2} = h_2(1 - h_2) \quad \frac{dz_2}{dW_2} = h_1$$

$$\frac{dLoss}{dW_2} = \begin{bmatrix} 0.19626223 & 0.71301043 & 0.48867814 \end{bmatrix}^T \begin{bmatrix} -0.23353037 & 0.2478971 \end{bmatrix} \begin{bmatrix} -0.62833404 & 0.45414264 \end{bmatrix}$$

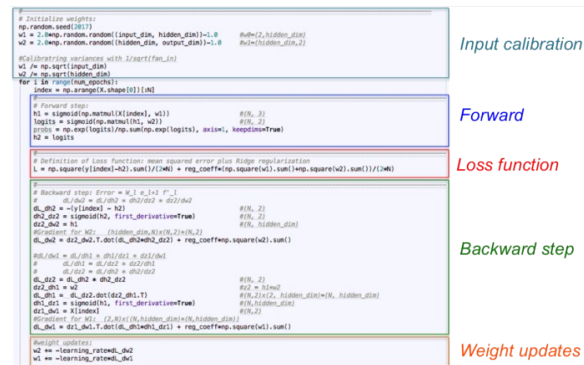
$$= \begin{bmatrix} -0.02879856 & 0.02209533 \\ -0.10462365 & 0.08027117 \end{bmatrix}$$

$$W^2 = W^{2*}$$

## Implementation

Let's translate the above mathematical equations to code only using [Numpy](#) as our **linear algebra engine**. Neural networks are trained in a loop in which each iteration present already **calibrated input data** to the network. In this small example, let's just consider the entire dataset in each iteration. The computations of **Forward step**, **Loss**, and **Backward step** lead to good generalization since we update the **trainable parameters** (matrices  $w_1$  and  $w_2$  in the code) with their corresponding **gradients** (matrices  $dL_{dw1}$  and  $dL_{dw2}$ ) in every cycle. Code is stored in this repository:

[https://github.com/omar-florez/scratch\\_mlp](https://github.com/omar-florez/scratch_mlp)

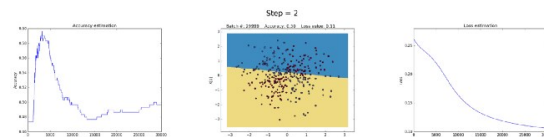


## Let's Run This!

See below **some neural networks** trained to approximate the **XOR function over** many iterations.

**Left plot:** Accuracy. **Central plot:** Learned decision boundary. **Right plot:** Loss function.

First, let's see how a neural network with **3 neurons** in the hidden layer has a small capacity. This model learns to separate 2 classes with a **simple decision boundary** that starts being a straight line but then shows a non-linear behavior. The loss function in the right plot nicely gets low as training continues.



Having **50 neurons** in the hidden layer notably increases the model's power to learn more **complex decision boundaries**. This could not only produce more accurate results but also **explode gradients**, a notable problem when training neural networks. This happens when very large gradients, multiply weights during backpropagation and thus generate large updated weights. This is the reason why the **Loss value suddenly increases** during the last steps of the training (step > 90). The **regularization component** of the Loss function computes the **squared values** of weights that are already very large ( $\sum(W^2)/2N$ ).

This problem can be avoided by **reducing the learning rate** as you can see below. Or by implementing a policy that reduces the learning rate over time. Or by enforcing a stronger regularization, maybe L1 instead of L2. **Exploding** and **vanishing gradients** are interesting phenomenons and we will devote an entire analysis later.


Machine Learning

Python




Deep Learning

Neural Networks

Featured




1.7K claps



ooo

WRITTEN BY



**Omar U. Florez**

Follow

Senior Research Manager in AI at Capital One - Conversational AI Research team. Teaching computers to see, read, and



## Towards AI

Follow

Towards AI, is the world's fastest-growing AI community for learning, programming, building and implementing AI.

[See responses \(5\)](#)

### More From Medium

More from Towards AI

#### Keras Callbacks Explained In Three Minutes



Andre Duong in Towards AI

Jul 21 · 3 min read ★



673



More from Towards AI

#### Tutorial on Data Visualization: Weather Data



Benjamin Obi Tayo Ph.D. in Towards AI

Jul 20 · 2 min read ★



497



More from Towards AI

#### MINE: Mutual Information Neural Estimation



Sherwin Chen in Towards AI

Jul 23 · 7 min read ★



380

