Notes: Multi Layer Perceptron (MLP)

Notes on general structure of a basic neural network that uses a sigmoid neuron

Sarbajit Sarkar

December, 2025

Abstract

This technical note provides a foundational overview of the Multi-Layer Perceptron (MLP), the cornerstone of modern artificial neural networks. We derive the mathematical framework for forward propagation using sigmoid neurons and provide a detailed step-by-step derivation of the Backpropagation algorithm using the chain rule. Furthermore, we discuss the optimization of model parameters via Gradient Descent and the quantification of network performance through the Mean Squared Error cost function. This document serves as a self-contained reference for understanding the transition from single-neuron models to layered, trainable architectures.

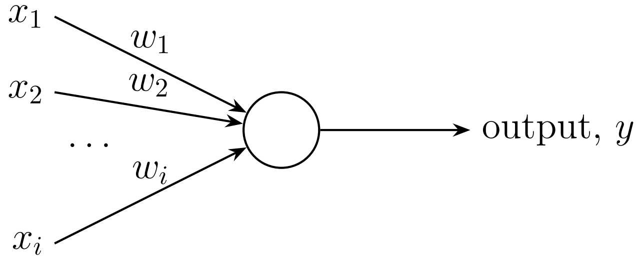
1 Introduction

One of the *Hello world* problem of the deep learning algorithm is handwritten digit recognition using a simple multilayer neural network. To understand the basic math of neural networks, we really do not need handwritten digit recognition specifically. Instead, we can generalize it as an algorithm that takes a multidimensional vector as input and returns a multidimensional vector (generally of a lower dimension). And most importantly, we would train the network using some labeled training data. Where a labeled training data means a large collection of input vector and desired output vector.

2 Architecture of the system

2.1 The sigmoid neuron

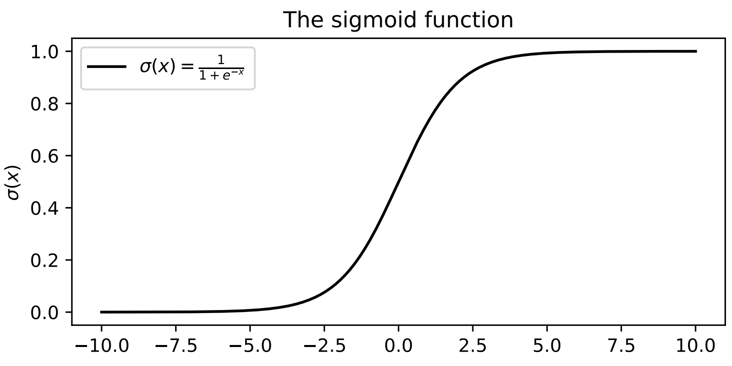
The **sigmoid neuron** is one of the first kind of artificial neuron conceptualized. A sigmoid neuron is an entity that takes several real valued *input* (x_i) each associated with it's *weight* (w_i) and returns a real value between 0 and 1. Each sigmoid neuron is associated with a parameter called *bias* (b) and the output value depends on the bias.



Perception formula,

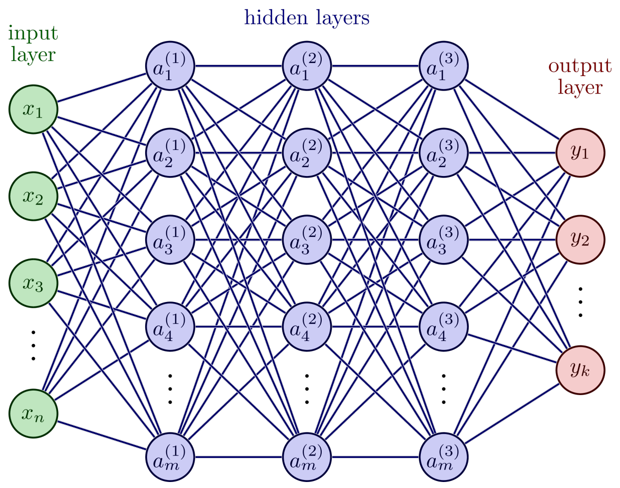
$$\begin{aligned} z &= x_i w_i + b \\ y &= \sigma(z) \\ \text{where, } \sigma(x) &= \frac{1}{1+e^{-x}} \end{aligned}$$

Note, *Einstein summation convention* is being used.



2.2 The layered network

The entire concept of the neural network lies in the way it mimics the biological brain in a mathematical model. Like in a human brain, neurons are connected with each other in a neural network to form a layered structure.



Each neuron of a layer is connected to each neurons of the next or previous layer. Each connection is associated with a specific weight value w and each neuron is associated with a specific bias value b. These weights and biases are the parameters of the model. These are like the constant values of a function that is chosen carefully so that the function behaves in the desired way. In addition to that, each neuron corresponds to the activation value $a \in [0, 1]$. Note that, the activation value depends on the activation values of the neurons of the previous layer. Activation values are the variables that depend on the input.

Notations

We must label each parameter and variable uniquely.

Activation level of i th neuron of L th layer,	$a_i^{(L)}$
Bias value of the i th neuron of L th layer,	$b_i^{(L)}$
Weight of connection between k th and j th neuron of layer $L-1$ and L,	$w_{jk}^{(L)}$

From proposed notation,

$$z_j^{(L)} = w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

$$\therefore a_j^{(L)} = \sigma\left(z_j^{(L)}\right)$$

Matrix representation

Activation levels of level of layer L could be represented as,
(asssume L layer have n neurons)

$$\mathbf{a}^{(L)} = \begin{bmatrix} a_1^{(L)} \\ a_2^{(L)} \\ \vdots \\ a_n^{(L)} \end{bmatrix}_{n\times 1}$$

Similarly,

$$\mathbf{a}^{(L-1)} = \begin{bmatrix} a_1^{(L-1)} \\ a_2^{(L-1)} \\ \vdots \\ a_m^{(L-1)} \end{bmatrix}_{m\times 1}$$

So, our weight and bias matrix becomes

$$\mathbf{w}^{(L)} = \begin{bmatrix} w_{1,1}^{(L)} & w_{1,2}^{(L)} & \cdots & w_{1,m}^{(L)} \\ w_{2,1}^{(L)} & w_{2,2}^{(L)} & \cdots & w_{2,m}^{(L)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1}^{(L)} & w_{n,2}^{(L)} & \cdots & w_{n,m}^{(L)} \end{bmatrix}_{n\times m} \qquad \mathbf{b}^{(L)} = \begin{bmatrix} b_1^{(L)} \\ b_2^{(L)} \\ \vdots \\ b_n^{(L)} \end{bmatrix}_{n\times 1}$$

So, in matrix representation

$$\mathbf{z}^{(L)} = \mathbf{w}^{(L)}\mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}$$
$$\mathbf{a}^{(L)} = \sigma\left(\mathbf{z}^{(L)}\right)$$

2.3 Behavior of the network

We can consider the neural network algorithm as a *black box*. Here, we really don't know what is going on the hidden layers. But it returns a multidimensional vector output as we give it a vector input.

The output always depends on the input we provide. But for arbitrary values of parameters (weights and biases) we get gibberish output. We must finetune the weights and biases to get the desired output for inputs.

An approach might be to analyze the model and determine the best set of values for the parameters. However, neural networks are one of the most complicated algorithms to finetune analytically.

3 Finetune the network

3.1 The cost function

The output returned by the neural network might be gibberish, not our desired output. However, we have to quantify how bad (or good) the output is with respect to the desired output. The **cost function** is that metric used to give rating to the output.

$$C(w_i, b_i) = \frac{1}{2}\sum_i |y_i - a_i^{(N)}|^2 \qquad \mathbf{a}^{(N)} \text{ is output and } \mathbf{y} \text{ is the desired output}$$

Variables: parameters of the model $\{w_i, b_i\}$
Outputs: A real number

To get desired output for any input we need to choose model parameters such that cost tends to zero.

Note, this is an example of cost function called *Mean Squared Error*. We may choose a different form of cost function too. But for this specific case, $\nabla_a C = (\mathbf{a}^{(N)} - \mathbf{y})$

3.2 Gradient descent

Gradient descent is a numerical approach to find local minima of a function.

Assume, $f(\mathbf{x}) = y$, $\mathbf{x}$ be a vector and y be a scalar.
Choose a starting point $\mathbf{x}_0$ and step α.
The choice of starting point and step is entirely a matter of trial and error. But this choice affects the algorithm a lot.

The core idea of gradient descent is to take small steps (α) iteratively down the steepest gradient and eventually reach the valley (minima).

Iteration rule

$$\mathbf{x}_n = \mathbf{x}_{n-1} - \alpha\nabla f(\mathbf{x}) \quad \text{where,} \nabla = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \cdots, \frac{\partial}{\partial x_n}\right)^T$$

For a good choice of $\mathbf{x}_0$ and α,

$$|f(\mathbf{x}_n) - f(\mathbf{x}_{n-1})| \to 0 \text{ as, } n \to \infty$$

We stop the iteration when $|f(\mathbf{x}_n) - f(\mathbf{x}_{n-1})| < \epsilon$

Example Python Implementation of Gradient descent

```
import numpy as np

# 1. Define Function and Gradient
def f(v):
    x, y = v
    return x**2 + y**2

def gradient(v):
    x, y = v
    return np.array([2 * x, 2 * y])

# 2. Setup Parameters
# Start at a random point far from center, e.g., (-4, 3)

current_pos = np.array([-4.0, 3.0])
alpha = 0.1
tolerance = 0.01

# 3. Gradient Descent Loop

while ( f(current_pos) > tolerance ):
    grad = gradient(current_pos)
    current_pos = current_pos - alpha * grad

# 4. Result
minima_position = current_pos
```

This is just an example code. Although it is mathematically correct but not standard for practical implementation.

3.3 The Backpropagation Method

The *backpropagation method* is an algorithm based on the chain rule of derivatives. It is used to find the gradient of the cost function.

Objective

The cost function depends on the model parameters,

$$C \equiv C\left(w_{jk}^{(L)}, b_j^{(L)}\right)$$

Define,

$$\mathbf{v} := \begin{bmatrix} \mathbf{w}^{(1)} \\ \vdots \\ \mathbf{w}^{(N)} \\ \mathbf{b}^{(1)} \\ \vdots \\ \mathbf{b}^{(N)} \end{bmatrix}_{k\times 1} = \begin{bmatrix} w_{1,1}^{(L)} \\ w_{1,2}^{(L)} \\ \vdots \\ w_{n,m}^{(L)} \\ \\ b_1^{(L)} \\ b_2^{(L)} \\ \vdots \\ b_n^{(L)} \end{bmatrix}_{k\times 1} \quad \text{Here, } \mathbf{b}^{(L)} = \begin{bmatrix} b_1^{(L)} \\ b_2^{(L)} \\ \vdots \\ b_n^{(L)} \end{bmatrix}_{n\times 1} \quad \text{and, } \mathbf{w}^{(L)} = \begin{bmatrix} w_{1,1}^{(L)} \\ w_{1,2}^{(L)} \\ \vdots \\ w_{i,j}^{(L)} \\ \vdots \\ w_{n,m}^{(L)} \end{bmatrix}_{(n\times m)\times 1}$$

Note that n is the number of neurons in layer L and m is the number of neurons in the previous layer. Also, $\mathbf{w}^{(L)}$ here is vectorized/flattened version of the original *weight matrix.*
The aim of the backpropagation method is to find the gradient,

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \\ \frac{\partial C}{\partial v_3} \\ \vdots \\ \frac{\partial C}{\partial v_k} \end{bmatrix}_{k\times 1}$$

v_i is the ith element of the $\mathbf{v}$ vector and k is the total number of model parameters.

Calculation

Note that,

$$z_j^{(L)} = w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

$$\therefore \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} = a_k^{(L-1)} \qquad \text{and, } \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = 1$$

Let us define a quantity error, $\delta_j^{(L)}$ that represents how much the total cost changes for a tiny change in the input *weighted sum* of the jth neuron of layer L,

$$\delta_j^{(L)} = \frac{\partial C}{\partial z_j^{(L)}}$$

Apply chain rule to get relation,

$$\frac{\partial C}{\partial z_j^{(L)}} = \sum_{k=1}^{n} \frac{\partial C}{\partial z_k^{(L+1)}} \frac{\partial z_k^{(L+1)}}{\partial z_j^{(L)}} = \frac{\partial C}{\partial z_k^{(L+1)}} \frac{\partial z_k^{(L+1)}}{\partial z_j^{(L)}} = \left(\frac{\partial C}{\partial z_k^{(L+1)}} \frac{\partial z_k^{(L+1)}}{\partial a_j^{(L)}} \right) \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \tag{1}$$

We know,

$$z_j^{(L)} = w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

$$\implies z_k^{(L+1)} = w_{kj}^{(L+1)} a_j^{(L)} + b_k^{(L+1)}$$

$$\implies \frac{\partial z_k^{(L+1)}}{\partial a_j^{(L)}} = w_{kj}^{(L+1)}$$

and,

$$a_j^{(L)} = \sigma\left(z_j^{(L)}\right)$$

$$\implies \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \sigma'\left(z_j^{(L)}\right)$$

From eq.(1) we have,

$$\delta_j^{(L)} = \left(\delta_k^{(L+1)} w_{kj}^{(L+1)}\right) \sigma'\left(z_j^{(L)}\right) \tag{2}$$

$$\text{Matrix form, } \boldsymbol{\delta}^{(L)} = \left(\left(\mathbf{w}^{(L+1)}\right)^T \boldsymbol{\delta}^{(L+1)}\right) \odot \sigma'\left(\mathbf{z}^{(L)}\right)$$

The eq.(2) carries the meaning why this specific algorithm is called backpropagation. We used the error of the next layer to calculate the error of the previous one.

Calculate elements of gradient,

$$\frac{\partial C}{\partial b_j^{(L)}} = \frac{\partial C}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}$$

$$\implies \frac{\partial C}{\partial b_j^{(L)}} = \delta_j^{(L)}$$

$$\frac{\partial C}{\partial w_{jk}^{(L)}} = \frac{\partial C}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}}$$

$$\implies \frac{\partial C}{\partial w_{jk}^{(L)}} = \delta_j^{(L)} a_k^{(L-1)}$$

So, now we just need to evaluate error for the last layer and we can find rest of it by propagating backwards. For the output layer,

$$\delta_j^{(N)} = \frac{\partial C}{\partial z_j^{(N)}} = \frac{\partial C}{\partial a_j^{(N)}} \frac{\partial a_j^{(N)}}{\partial z_j^{(N)}}$$

$$\implies \delta_j^{(N)} = \left(a_j^{(N)} - y_j\right) \sigma'\left(z_j^{(N)}\right)$$

Matrix form,

$$\boldsymbol{\delta}^{(N)} = \nabla_a C \odot \sigma'\left(\mathbf{z}^{(N)}\right) = \left(\boldsymbol{a}^{(N)} - \mathbf{y}\right) \odot \sigma'\left(\mathbf{z}^{(N)}\right)$$

Note, specifically for a *sigmoid neuron*, $\sigma'(z) = \sigma(z)(1 - \sigma(z)) = a(1 - a)$ as, $\sigma(x) = \frac{1}{1+e^{-x}}$

Summary

$$
\begin{aligned}
\delta_j^{(N)} &= \frac{\partial C}{\partial a_j^{(N)}}\,\sigma'\left(z_j^{(N)}\right) = \left(a_j^{(N)} - y_j\right)\sigma'\left(z_j^{(N)}\right) \\
\delta_j^{(L)} &= \left(\delta_k^{(L+1)} w_{kj}^{(L+1)}\right)\sigma'\left(z_j^{(L)}\right) \\
\frac{\partial C}{\partial w_{jk}^{(L)}} &= \delta_j^{(L)} a_k^{(L-1)} \\
\frac{\partial C}{\partial b_j^{(L)}} &= \delta_j^{(L)}
\end{aligned}
$$

Matrix form,

$$
\begin{aligned}
\boldsymbol{\delta}^{(N)} &= \nabla_a C \odot \sigma'\left(\mathbf{z}^{(N)}\right) \\
\boldsymbol{\delta}^{(L)} &= \left(\left(\mathbf{w}^{(L+1)}\right)^T \boldsymbol{\delta}^{(L+1)}\right) \odot \sigma'\left(\mathbf{z}^{(L)}\right) \\
\frac{\partial C}{\partial \mathbf{w}^{(L)}} &= \boldsymbol{\delta}^{(L)}(\mathbf{a}^{(L-1)})^T \\
\frac{\partial C}{\partial \mathbf{b}^{(L)}} &= \boldsymbol{\delta}^{(L)}
\end{aligned}
$$

So finally, we can assemble the global gradient vector ∇C by stacking the flattened gradients of each layer.

$$
\nabla C = \begin{bmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \\ \frac{\partial C}{\partial v_3} \\ \vdots \\ \frac{\partial C}{\partial v_k} \end{bmatrix}_{k\times 1} = \begin{bmatrix} \delta_1^{(L)} a_1^{(L-1)} \\ \delta_1^{(L)} a_2^{(L-1)} \\ \vdots \\ \delta_n^{(L)} a_m^{(L-1)} \\ \\ \delta_1^{(L)} \\ \delta_2^{(L)} \\ \vdots \\ \delta_n^{(L)} \end{bmatrix}_{k\times 1}
$$

If you are confused, note that the order of elements is the same as the order of elements in the definition of $\mathbf{v}$.

3.4 The Training Process

For training, we need a large collection of training data. Training data simply means the input vector and the desired output vector for the input. The training is nothing but finding the minima of cost function using *gradient descent* loop. A training loop consists of four steps.

1. **Forward Pass**
   Input the vector $\mathbf{a}^{(0)}$ in the neural network and compute $\mathbf{a}^{(L)}$ for each layer until the output $\mathbf{a}^{(N)}$ is reached.

2. **Backward pass**
   Calculate the errors $\boldsymbol{\delta}^{(L)}$ using the backpropagation method.

3. **Gradient calculation**
   Calculate the gradient ∇C using $\boldsymbol{\delta}^{(L)}$ and $\mathbf{a}^{(L)}$

4. **Parameter update**
   Use the *gradient descent* formula to update model parameters,

$$\mathbf{v}' = \mathbf{v} - \alpha\ \nabla C$$

Points to note

If we have some collection consisting t number of training data, first we calculate and average the cost and the gradient and then update the parameters.

$$C = \frac{1}{t}\sum_{i=1}^{t} C_i$$

$$\nabla C = \frac{1}{t}\sum_{i=1}^{t} \nabla C_i$$

Note, C_i refers to the cost associated with the ith training datapoint.

But, practically this method is very slow and resource intensive. So, mini-batch processing and other things are done. Also, instead of gradient descent, we use other fast algorithms such as *stochastic gradient descent*. However, the core concept remains the same.

4 Conclusion

The Multi-Layer Perceptron demonstrates how simple computational units, when layered and interconnected, can approximate complex non-linear functions. Through the back-propagation algorithm, we can efficiently distribute the "blame" for an incorrect output back through the network, allowing each weight and bias to be adjusted in the direction that minimizes total cost. While modern architectures have introduced more sophisticated activation functions (like ReLU) and optimizers (like Adam), the fundamental mechanics of the forward and backward passes described here remain the universal engine of deep learning. Mastery of these derivations provides the necessary intuition to debug, design, and optimize more advanced neural architectures.