An Undergraduate's Explanation of the Multilayer Perceptron: Mathematical Concepts and a Python3 Implementation

Jared Frazier*
Department of Computer Science†, Middle Tennessee State University

2022
January

1 Preamble

The purpose of the present document is to explain and implement the major mathematical constructs/concepts behind feedforward neural networks, specifically the multilayer perceptron. This includes the layers that compose such networks, the cost (aka loss, error, or objective) function and activation functions, the forward pass through the network, the computation of gradients via backpropagation (a concept which is often "handwaved" to the extreme or explained in so much detail as to be confusing–at least in the author's experience), and the update of model parameters via mini-batch stochastic gradient descent. If the ideas such as *layer* and *backpropagation* are entirely unfamiliar to you, then the author encourages you to visit texts such as *Deep Learning* (free, online), *Neural Networks and Deep Learning* (free, online), *Hands-on Machine Learning with Scikit-Learn, TensorFlow and Keras 2ed* (buy), and/or *Deep Learning with Python 2ed* (buy). The present document is not intended to be a comprehensive overview of neural networks nor an extremely in-depth explanation but rather a document that highlights certain concepts that the author found confusing or ambiguous when he was learning about neural networks.

The explanations in the present document will essentially alternate between mathematics and concrete implementations using the Python3 programming language with the NumPy library. Note that the implementations here are not intended to be optimal. If you would like optimal implementations, the author encourages you to use a machine learning API such as TensorFlow or PyTorch, the tutorials for which will abstract and make easily usable many of the concepts elucidated in the present document.

*Not endorsed by or affiliated with any of the authors or entities associated with references
†Undergraduate 2021-2022

2 Introduction

The neural network (function approximator) is just a chain of geometric transformations (functions) each parametrized by a weight matrix $W \in \mathcal{R}^{n_x \times n_h}$ and a bias vector $b \in \mathcal{R}^{n_h}$ on the input vector $x \in \mathcal{R}^{n_x}$. With this single sentence the author makes several claims and uses notation that may or may not be familiar to the reader. Note that n_x is the input size (i.e., output size of the previous layer in a network with L total layers) and n_h is the number of hidden units in the current layer. The weights and biases are collectively known as parameters in the literature. Hence, the use of the phrase a *function* that is *parametrized* by *weights* and *biases.* Also, note that the input vector x is a *column vector*, which is important to understand since vector/matrix operations (dot product, Hadamard (element-wise) product, addition, etc.) restrict their operands to particular shapes. When using the term *vector*, the author is always referring to a *column vector* unless otherwise specified. Also, note that $x \in \mathcal{R}^{n_x}$ indicates that x is a vector with n_x elements and the j^{th} is a real number. For example, the below vector x is shown and a common vector operation known as transposition (converts a *column vector* to a *row vector* and is denoted with a superscript of $\top$) is also shown.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{(n_x)} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{(n_x-1)} \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{(n_x-1)} \end{bmatrix}^\top \tag{1}$$

Many programming languages assign the first element of a vector the index 0; this notation is shown above in addition to the more standard mathematical notation where the first element begins with the index 1. For the remainder of this document, the author will use the index 0 assumption since the author's implementation of the the neural network will use the Python programming language. If you wish to implement the same algorithms in a language such as R or Wolfram Mathematica, be wary of this index discrepancy.

3 Equations

Here the author defines the operations that occur for a multilayer perceptron (MLP). Note that the MLP can sometimes refer to any class of feedforward neural network, that is a network that applies affine transformations and activation functions to input from a previous layer in the network.

3.1 Single Hidden Layer Neural Network

$$\begin{aligned}
NeuralNet_{\theta}(X) &= \sigma(ReLU(XW^{[1]} + b^{[1]})W^{[2]} + b^{[2]}) \\
&= \sigma(g(ReLU(w(X)))) \\
A^{L} &= NeuralNet_{\theta}(X) && \text{Activation matrix } A \text{ for last layer } L
\end{aligned} \tag{2}$$

where σ, $ReLU$, g, and w are defined as follows:

$$\begin{aligned}
\sigma(t) &= \frac{1}{1+e^{-t}} \\
ReLU(t) &= max(0, t) \\
g_{\theta^{[2]}}(A) &= AW^{[2]} + b^{[2]} \\
w_{\theta^{[1]}}(A) &= AW^{[1]} + b^{[1]} \\
u_{\theta^{[l]}}(A) &= AW^{[l]} + b^{[l]} \quad \text{General form of } g \text{ and } w \text{ for } l^{th} \text{ layer}
\end{aligned} \tag{3}$$

3.2 Neural Network Prediction (Forward Pass)

$$\hat{y} \leftarrow NeuralNet_{\theta}(X) \tag{4}$$

3.3 Mean Squared Error Loss Function

$$\begin{aligned}
\mathcal{L}_{\theta}(X) &= \frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2 \\
&= \frac{1}{N}\sum_{i=1}^{N}(a^{(i)} + y^{(i)})^2 \quad a^{(i)} \text{ is the activation vector of the last layer } L \text{ for the } i^{th} \text{ input}
\end{aligned} \tag{5}$$

3.4 Gradient Update

$$\theta_i \leftarrow \theta_i - \eta(\nabla_{\theta}\mathcal{L}_{\theta}(\hat{y}, y)) \tag{6}$$