The Multilayer Perceptron

Mathematical Concepts, a Python3 Implementation, and Selected Applications

Jared Frazier*†‡

2022 January

*Department of Computer Science (2021-2022), Middle Tennessee State University.

†Department of Chemistry (2018-2021), Middle Tennessee State University.

‡Not endorsed by or affiliated with any of the authors or entities listed in the References section.

Contents

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. Following this implementation, I test the network I have built "from scratch" using `NumPy` and compare it with a reference implementation in `TensorFlow`. The concepts I will cover include 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 that is often "handwaved" to the extreme or explained in so much detail as to be utterly confusing–at least in my 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 I encourage you to visit 3Blue1Brown's Deep Learning YouTube Series [1] and peruse the first few chapters of texts such as *Deep Learning* (free, online) [2], *Neural Networks and Deep Learning* (free, online) [3], *Hands-on Machine Learning with Scikit-Learn, TensorFlow and Keras 2ed* (buy) [4], and/or *Deep Learning with Python 2ed* (buy) [5]. The present document is not intended to be a comprehensive overview of neural networks nor an extremely in-depth mathematical treatise but rather a document that highlights certain concepts that I found confusing or ambiguous when I was first learning about neural networks, particularly regarding the backpropagation algorithm.

Having taken my institution's CSCI 4350 (Intro to AI) and CSCI 4850 (Neural Nets) courses in addition to conducting independent research using variational autoencoders, recurrent neural nets, convolutional neural nets, and self-attention, I am ashamed to say my fundamental understanding of neural nets was far weaker than it should have been. While I am not a master of pedagogy, I hope this document will serve as a reminder of fundamental concepts for my future self and for others.

2 Introduction

The neural network (function approximator) is just a chain of geometric transformations (functions) each parametrized by a weight matrix $W \in \mathbb{R}^{n_h \times n_x}$ and a bias vector $b \in \mathbb{R}^{n_h}$ on the input vector $x \in \mathbb{R}^{n_x}$. The geometric transformations of the neural network are encapsulated by connecting layers (e.g., dense/fully connected layers) together. A neural network has L total layers and the current layer l receives the output from the previous layer $(l-1)$. Note that n_x is the number of features (or independent variables) in the input and n_h is the number of hidden units in the current layer. The following subsections will briefly elucidate both the claims and notation of the first sentence of this section.

2.1 Parametrized Functions

I assume you know what a function is; however, the term *parametrized* is one that appears often in deep learning literature and should be well-understood by the student. Consider a generic quadratic function [6] as

$$f(x) = ax^2 + bx + c \tag{1}$$

The *variable* x is an *argument* to the function f that has *parameters* a, b, and c. The parameters determine the behavior of the function (e.g., the steepness of slope, intercepts, shape, etc.) while the variable can take on some range of values. When a variable that takes on a particular value is passed as an argument to the function with defined parameters, the result is some other value y if $y = f(x)$. This explanation of a function should not be anything new; however, the *parameters* are quantities of particular interest for neural networks since the parameters are the quantities that are *learned* by the neural network over time. What it means to learn parameters will be explained later.

A neural network can be denoted as a function h with parameters W and b of a variable x. This statement can be compactly written as $h_{W,b}(x)$ as in [4] or $h(x; W, b)$ as in [2]. Incidentally, I encourage you to know both notation, but the former appears to be more common in general in addition to being quite common for specialized probabilistic models such as the variational autoencoder [7]. The subscript with W and b means that the weights W and biases b are *parameters* of the neural network h. The claim that a neural network is a chain of functions is useful later during the updating of the parameters of the network. To briefly illustrate the idea of chaining functions, the generic quadratic function, which can be defined as a composite function f, can be decomposed into simpler functions shown below.

$$g_a(x) = ax^2 \tag{2}$$

$$u_b(x) = bx \tag{3}$$

$$f_{a,b,c}(x) = g_a(x) + u_b(x) + c \tag{4}$$

Recognizing the decomposition of composite functions into their constituent functions is useful for applying rules of calculus–the basis of parameter learning via the backpropagation algorithm illustrated later.

2.2 Operand Types

The input x is not a single value as is conceived in the elementary formulations described above. Rather, the input x is a list of values referred to as a *column vector*. Each element of the vector is a value that a particular feature, or independent variable, could take on. The shape of the vector x is important to understand since the functions and operations performed by the neural network (dot product, Hadamard product, addition, etc.) restrict their vector/matrix operands to particular shapes. When using the term *vector*, I am

always referring to a *column vector* unless otherwise specified. Also, note that $x \in \mathbb{R}^{n_x}$ indicates that x is a vector with n_x elements and the j^{th} element is a real number. For example, the below vector x is shown and a common vector operation known as the transpose (converts a *column vector* to a *row vector* and is denoted with a superscript $\top$) is also shown.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{(n_x)} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{(n_x-1)} \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{(n_x-1)} \end{bmatrix}^\top \tag{5}$$

Many programming languages access the first element of a vector using 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, I will use the index 0 assumption since my implementation of the neural network uses 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. Consequently, with the index beginning at 0, the last index of a vector with n_x elements will be $(n_x - 1)$... and woe is the programmer who commits an off-by-one error.

A matrix W represents the weight of edges between the k^{th} input neuron of n_x^{l-1} total input neurons (i.e., neurons in the previous layer $(l-1)$) and the j^{th} hidden neuron of n_h^l total hidden neurons in the current layer l. A weight matrix looks similar to the vector, except rather than having a single column, a matrix has a rows and columns–looking like a table. Vectors can be referred to as rank-1 tensors, matrices as rank-2 tensors, so-on and so-forth for multiple index "lists" in higher dimensions. A sample weight matrix $W \in \mathbb{R}^{n_h \times n_x}$ is shown below.

$$W = \begin{bmatrix} W_{00} & W_{01} & W_{02} & \cdots & W_{0(n_x-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ W_{(n_h-1)0} & W_{(n_h-1)1} & W_{(n_h-1)2} & \cdots & W_{(n_h-1)(n_x-1)} \end{bmatrix} \tag{6}$$

When implementing machine learning algorithms, pay close attention to the input and output shapes described in a dataset, journal article, tutorial, or the source code of others. If you do not pay careful attention to these shapes, your implementation may not run or, worse, it *will* run but it will not execute the operations you intended[1].

3 The Multilayer Perceptron

Here I define the operations that occur for the 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 The Dense/Fully Connected Layer

The affine transformation, which is the most fundamental transformation of the densely/fully connected layers that exist in the MLP, yields a weighted input vector z^l with elements $z_j^l = W_{jk}^l a_k^{l-1} + b_j^l$ for a layer l and neuron j. Here, the activation a_k^{l-1} denotes the activation of the k^{th} neuron of the previous layer $(l-1)$. Importantly, the input layer has no activation function ϕ associated with it, so the activation vector a^0 equals the input vector x. Moreover, the inner dimensions of the matrix product Wx match, that is the subscripts k are "adjacent" to one another. While you may observe that the activation vector $a^l \in \mathbb{R}^{n_a}$ is clearly not a matrix, numerical libraries will often treat a vector $v \in \mathbb{R}^{n_v}$ as equivalent to a matrix with a single column (i.e., $V \in \mathbb{R}^{n_v \times 1}$) for the purposes of performing fast matrix-matrix calculations.

Until now the discussion of the MLP has been entirely in abstract mathematical notation, so now a visual of a single layer (meaning single hidden layer) MLP is shown. The activation function ϕ is vectorized, meaning it applies to each element of a vector, matrix, or rank-n tensor. Vectorized activation functions are often denoted $\phi(\cdot)$ or $\sigma(\cdot)$, though I do not like the latter notation as σ typically references the sigmoid function. The number of neurons in a hidden layer (or output layer for that matter) n_h does not have to be constant, and this is shown in the Figure 1 below where the output layer has only a single neuron while the hidden layer has three neurons.

3.2 The Forward Pass and Cost Function

The MLP is a function approximator and the MLP learns the parameters of this function. Since the learning is just the determination of the values of the parameters of the MLP, then there must be some other function that determines how well the parameters of the MLP approximate the desired function. This other function is called the cost or loss function and is denoted C, though

[1] This is especially true for the matrix multiplication operation and the Hadamard (element-wise) product.

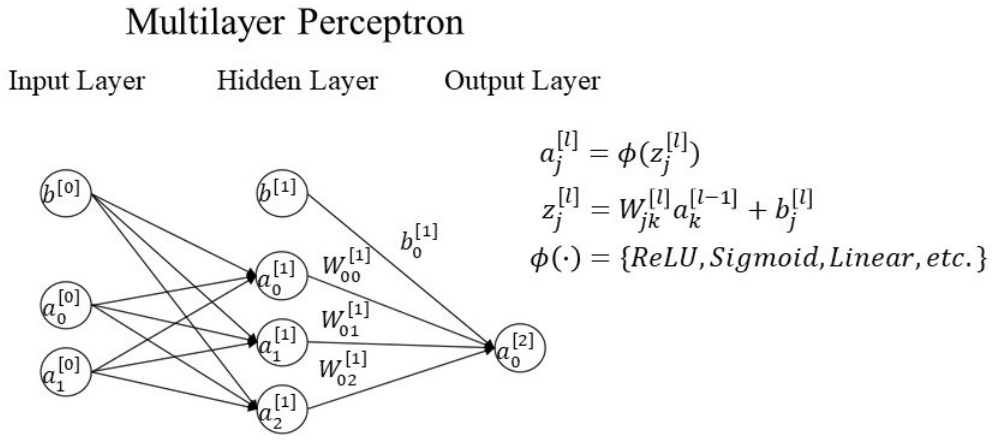

Figure 1: The multilayer perceptron with bias vectors b in layers $[l]$ and activations a for the j^{th} neuron in a layer. The weights W of the last layer are shown as edges connecting neurons in the hidden layer to the output layer. The operations are defined to the right of the figure along with a set of common vectorized activation functions $\phi(\cdot)$.

sometimes it is denoted $\mathcal{L}$ or J[2]. For regression problems, the most common cost function is the mean squared error. The cost function, unlike previous operations, returns a scalar and *not* a vector. The scalar value can be used to estimate performance during training; however, when implementing backpropagation, you will be more interested in the derivative of the cost with respect to certain variables and using gradient vectors (discussed later) rather than the scalar cost.

$$\begin{aligned} C &= \frac{1}{N} \sum_x C_x \\ &= \frac{1}{N} \sum_x ||(\hat{y} - y)||^2 \\ &= \frac{1}{N} \sum_x ||(h_{W,b}(x) - y)||^2 \\ &= \frac{1}{N} \sum_x ||(a^L - y)||^2 \end{aligned} \tag{7}$$

where x and y are in $\mathbb{R}^{n_x}$ and

[2]Despite these functions sharing the same notation as a matrix W, the result of the function is not a tensor but is rather a scalar.

$$||x - y||^n = \frac{1}{n_x} \sum_{i=0}^{n_x - 1} (x_i - y_i)^2 \tag{8}$$

The cost function for a single sample C_x is a multivariable function and it can also be written as a function of its parameters θ like $C_x(\theta)$ where $\forall$ layers l, $\theta = \{W^l, b^l\}$. Additionally, C_x can be written to emphasize a certain parameter such as the weight $C_x(W)$. More importantly, the derivative of the cost function C_x with respect the activation a_j^L for the j^{th} neuron of the last layer (i.e., output layer) L is defined as $\frac{\partial C_x}{\partial a_j^L}$. To avoid long superscripts, assume that the the last layer L actually indexes the location $(L-1)$. These partial derivatives are relevant to the backpropagation algorithm illustrated shortly.

Before moving on to backpropagation, I briefly explain the meaning of the activations of the last layer. Put simply, The activation vector a^L for the last layer L is the prediction that the MLP makes. The activation vector a^L has a number of elements determined by the number of units n_h^L in the last layer, and to index the j^{th} activation (a scalar) of the activation vector one would write a_j^L. The activation for the j^{th} neuron is used to compute errors that are relevant for the backpropagation algorithm since how well a^L "lines up" with the known result y is determined by the choice of parameters W and b for all layers l. Here I loosely use the phrase "lines up" because $a^L = \hat{y}$ and for supervised learning problems, it is desirable for $\hat{y}$ to be very "close" to y. Of course, the metric for such "closeness" is the cost function!

3.3 Backpropagation

The key to learning for the MLP, and for artificial neural networks in general, is updating the weights W^l and biases b^l for each layer l such that the network performs better with respect to the loss function[3]. Since W and b are tensors, the gradients[4] of the cost function with respect to these parameters ($\frac{\partial C_x}{\partial W_{jk}^l}$ and $\frac{\partial C_x}{\partial b_j^l}$) are computed to determine how the elements of the parameter tensors should be modified to produce better predictions.

3.3.1 Parameters Revisited

At this point you might ask yourself again what the difference between variables and parameters is. In pure mathematics classes, you are often asked to minimize a function with respect to some variable(s), and rarely (at least in my experience) with respect to a parameter. Why minimize the cost function with respect to the parameters? The statement "minimize with respect to a parameter" seems to contradict the very definition of a parameter: "arguments that are... not explicitly varied in situations of interest are termed *parameters*"

[3]Machine learning APIs like TensorFlow tend to formulate cost functions such that they can be *minimized* (e.g., $minimize(NegativeLogLikelihood) \equiv maximize(LogLikelihood)$)

[4]Denoted using the gradients ∇ or sometimes the $\vec{\nabla}$ operator.

[8]. The simplest intuition behind the minimization of parameters instead of variables is that the parameters are constant for a single iteration of a machine learning experiment, and then updated such that the model performs better on future experiments. Conversely, variables (e.g., the input x) are independent of the model. An experiment encompasses the *fitting* of a model for a number of iterations equal to $NumEpochs \times \frac{TrainingDataSize}{BatchSize}$. A *batch* for learning is just a random subset of size m of the training data. Thus, one iteration of the machine learning experiment is performing the forward pass (i.e., making a prediction $a^{i,L} = y^i$ for each training example x^i in the batch), computing the gradients for the weight matrices and biases of each layer based on the model error, and then updating those parameters.

3.3.2 Typical Mathematical Notation for Gradient Descent

Knowing that the parameters need to be updated, most often the below equations will be written and the gradient computation process will be abstracted from the reader.

$$W^l = W^l - \eta \nabla_{W^l} C \tag{9}$$

$$b^l = b^l - \eta \nabla_{b^l} C \tag{10}$$

$$\theta^l = \theta^l - \eta \nabla_{\theta^l} C \tag{11}$$

The equations make quite a lot of sense if you think about exactly what they are saying. The first of the three equations says "update the weight matrix for a given layer by subtracting from the current weight matrix the gradient of the cost function with respect to that same weight matrix." By definition, the gradient points in the direction of local maxima, and the negative gradient points in the direction of local minima [9]. The size of the "step" in the direction of the local minima is proportional to the learning rate η, which is a hyperparameter (a parameter explicitly set by a user) of the model. Therefore, the semantic explanation of the equations is essentially to change the weights and biases (sometimes combined and denoted as a single parameter tensor θ) of the model by a small amount until convergence[5] is reached. In this way, a model will have parameters that can be used to *infer* predictions based on unseen data. Consequently, when a model is making predictions without updating the parameters, it is called *machine learning inference* or some variation on the word *inference*.

3.3.3 The Four Fundamental Equations of Backpropagation

While the previous equations are nice for rapidly intuiting the meaning of gradient descent, they provide no insight into the computation of *gradients* that are obviously needed for *gradient* descent.

[5] Ideally on a global minimum in the N-dimensional cost function space.

The fundamental equations of backpropagation, which are critical for *gradient* descent, are claimed without proof (see [3] for proofs). While the previous equations are the "pretty" form of gradient descent, for the accompanying implementation, one does not compute the gradient vectors of these parameters directly, rather the components of these gradient vectors $\nabla_W C$ or $\nabla_b C$ are computed using other outputs of the network such as a^l and z^l.

$$\delta^L = \nabla_{a^L} \mathrm{C} \circ \frac{\partial \phi}{\partial z}(z^L) \tag{12}$$

$$\delta^l = ((W^{l+1})^\top \delta^{l+1}) \circ \frac{\partial \phi}{\partial z}(z^l) \tag{13}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{14}$$

$$\frac{\partial C}{\partial W^l} = \delta^l (a^{l-1})^\top \qquad \delta^l \in \mathbb{R}^{j \times 1}; (a^{l-1})^\top \in \mathbb{R}^{1 \times k} \tag{15}$$

Before explaining the meaning of the above equations, consider first the gradient for a function f of two variables x and y:

$$f(x, y) = x^2 y \tag{16}$$

$$\nabla f(x, y) = \langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \rangle \tag{17}$$

$$\nabla f(x, y) = \langle 2xy, x^2 \rangle \tag{18}$$

You can see that the gradient is the vector where each element is the first-order partial derivative with respect to each variable in the function f.

Now, consider the error vector δ^L for the last layer L and note the gradient operator ∇_{a^L} is with respect to the activation vector a^L whose elements can be defined as $a^L = \{a^L_j\}_{j=0}^{n_h - 1}$. Thus, the error vector for the last layer is shown below for clarity. Assume that the cost function C is for a single example x and $\delta^L \in \mathbb{R}^{n_y}$ where n_y is the number of targets (or output classes).

$$\delta^L = \begin{bmatrix} \delta^L_0 \\ \delta^L_1 \\ \delta^L_2 \\ \vdots \\ \delta^L_{(n_y - 1)} \end{bmatrix} = \begin{bmatrix} \frac{\partial C}{\partial a^L_0}(a^L_0) \frac{\partial \phi}{\partial z}(z^L_0) \\ \frac{\partial C}{\partial a^L_1}(a^L_1) \frac{\partial \phi}{\partial z}(z^L_1) \\ \frac{\partial C}{\partial a^L_2}(a^L_2) \frac{\partial \phi}{\partial z}(z^L_2) \\ \vdots \\ \frac{\partial C}{\partial a^L_{(n_y - 1)}}(a^L_{(n_y - 1)}) \frac{\partial \phi}{\partial z}(z^L_{(n_y - 1)}) \end{bmatrix} \tag{19}$$

Showing the elements of the error vector δ^L above demonstrates what must be computed: $\frac{\partial C}{\partial a_j^L}$ and $\frac{\partial \phi}{\partial z}$. The derivative of the activation function $\frac{\partial \phi}{\partial z}$ is very easy to compute since activation functions with well-known derivatives are typically selected. For example, if the activation function is the sigmoid function, then the derivative is known and shown below.

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{20}$$

$$\frac{\partial \phi}{\partial z}(z) = \sigma(z)(1 - \sigma(z)) \tag{21}$$

The derivative of the cost function with respect to activations of different neurons $\frac{\partial C}{\partial a_j^L}$ is also easily computed. For the mean squared error function, you would derive the function with respect to only a single activation. That is, you are finding the rate of change of the cost function for an infinitesimally small change in the j^{th} activation of a layer.

$$C = \frac{1}{n_h} \sum_{j=0}^{(n_h - 1)} (a_j^L - y_j)^2 \tag{22}$$

$$\frac{\partial C}{\partial a_j^L} = \frac{2}{n_h}(a_j^L - y_j) \tag{23}$$

You might ask yourself, "how do I actually get $\frac{\partial C}{\partial a_j^L}$." Initially, I assumed the answer would be something quite confusing, but when computing such a derivative, you need only consider what a function is. If you think of $\frac{\partial C}{\partial a_j^L}$ as just another function, which it is, then of what variables is it a function? It is simple: a_j^L and y_j! We *know* the values of these variables since we know a_j^L is just the output of a single neuron in a layer of neurons. Moreover, y_j is not something we have to compute, but rather it is a known element of a vector of targets y, or desired outcomes.

Now, why does $\nabla_{a^L} C$ specify a^L? It is because C can be written as a function of many parameters and variables, and you only care about certain variables when performing backpropagation. A single layer neural network, that is a network with two weight matrices W^0 and W^1; two bias vectors b^0 and b^1; and three activations a^0, a^1, and a^2 can be written as $C(W^0, W^1, b^0, b^1, a^0, a^1, a^2)$. You might note that the number of activation vectors a exceeds the number of weight matrices and bias vectors. Recall that the input layer of the MLP has no preceding weight matrix, and a linear activation $\phi(x) = x$ is applied to the input vector, thus resulting in activations $a^0 = x$. Refer to Figure 1 to observe the lack of a weight matrix before the input layer. The gradient must be taken with respect to a certain parameter, otherwise the partial derivative with respect to each variable of a vector valued function would result. To visualize what the gradient only with respect to the activations of the last layer looks like, consider the below vector.

$$\nabla_{a^L} C = \begin{bmatrix} \frac{\partial C}{\partial a_0^L} \\ \frac{\partial C}{\partial a_1^L} \\ \frac{\partial C}{\partial a_2^L} \\ \vdots \\ \frac{\partial C}{\partial a_{(n_h-1)}^L} \end{bmatrix} \tag{24}$$

The above vector is in stark contrast to $\nabla_{\theta^L} C$, where θ^L are the relevant parameters and variables for the layer L. Such a matrix is called the Jacobian matrix and is shown below.

$$\nabla_{\theta^L} C = \begin{bmatrix} \frac{\partial C}{\partial a_0^L} & \frac{\partial C}{\partial W_{00}^L} & \frac{\partial C}{\partial W_{01}^L} & \frac{\partial C}{\partial W_{0(n_x-1)}^L} & \frac{\partial C}{\partial b_0^L} \\ \frac{\partial C}{\partial a_1^L} & \frac{\partial C}{\partial W_{10}^L} & \frac{\partial C}{\partial W_{11}^L} & \cdots & \frac{\partial C}{\partial b_1^L} \\ \frac{\partial C}{\partial a_2^L} & \frac{\partial C}{\partial W_{20}^L} & \frac{\partial C}{\partial W_{21}^L} & \cdots & \frac{\partial C}{\partial b_2^L} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial C}{\partial a_{(n_h-1)}^L} & \frac{\partial C}{\partial W_{(n_h-1)0}^L} & \frac{\partial C}{\partial W_{(n_h-1)1}^L} & \frac{\partial C}{\partial W_{(n_h-1)(n_x-1)}^L} & \frac{\partial C}{\partial b_{(n_h)-1}^L} \end{bmatrix} \tag{25}$$

Now that you understand the computation of δ^L, finding δ^l is trivial since the errors in the current layer plus one propagate backward through the network (i.e., you will use δ^L in the computation of δ^{L-1}). To validate the shapes of the operands relevant to the equation for δ^l, refer to the below statements.

$$W^{l+1} \in \mathbb{R}^{n_h^{l+1} \times n_h^l} \tag{26}$$

$$\delta^{l+1} \in \mathbb{R}^{n_h^{l+1}} \tag{27}$$

$$\frac{\partial \phi}{\partial z}(z^l) \in \mathbb{R}^{n_h^l} \tag{28}$$

With these comments about backpropagation, the practical implementation of the four equations can be discussed. Since the above equations focus on the computation of gradients, they must be used in the *gradient descent* equations below.

$$W^l = W^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^\top \tag{29}$$

$$b^l = b^l - \frac{\eta}{m} \sum_x \delta^{x,l} \tag{30}$$

The learning rate η is typically $\eta = \{1^i\}_{i=-2}^{-4}$, and m for mini-batch stochastic gradient descent is typically $m = \{2^i\}_{i=1}^{10}$.

4 Implementation

Here I define the Python classes that encapsulate the attributes (data members) and methods (functions) associated with the MLP. Since this approach is object oriented, it was helpful to define first the subunits of the MLP: operations (i.e., activation functions and cost functions) and layers.

4.1 Operation

The activation functions and cost functions are children of the Python abstract base class I define as `Operation`. The `Operation` has no arguments and initializes no attributes, but does have a `__call__` method, a `derivative` method, and a `gradient` method. All children of the `Operation` must implement these methods.

4.2 DenseLayer

The forward pass requires defining a class that encapsulates the fully/densely connected layer of the MLP. I define this as `DenseLayer` with attributes for the input dimensions n_x, the weight matrix $W \in \mathbb{R}^{n_h \times n_x}$, and the bias $b \in \mathbb{R}^{n_h}$. The `DenseLayer` is correspondingly passed arguments for n_x, n_h, and a `Callable` that will be used as the vectorized activation function $\phi(\cdot)$ for a given layer.

The `__init__` method saves the `Callable` and passes the n_x and n_h arguments to initialize the weight matrix and bias vector. The `glorot_uniform` method initializes the values of the weight matrix using Glorot Uniform initialization [10]. The bias vector is initialized with zeros.

The activations and weighted inputs of the `DenseLayer` are computed with the `__call__` method defined as

$$z = XW^\top + b \tag{31}$$

$$a = \phi(z) \tag{32}$$

where x is an argument to the `__call__` method and W, b, and ϕ are attributes of the `DenseLayer`. Note that x can be $x \in \mathbb{R}^{n_x}$, which is equivalent to $X \in \mathbb{R}^{1 \times n_x}$ for the purposes of matrix multiplication. However, the training dataset $\mathcal{D}$ is commonly $\mathcal{D} \in \mathbb{R}^{N \times n_x}$ where N is the total number of training examples. Therefore, a subset (aka batch) m of N means that $X \in \mathbb{R}^{m \times n_x}$.

Consider now the shapes of the affine transformation and the subsequent activation:

$$Z_{m \times n_h} = X_{m \times n_x} (W_{n_h \times n_x})^\top + b_{n_h} \tag{33}$$

For this weighted input matrix $Z_{m \times n_h}$, you may observe a problem with how it is computed: a *vector* is *added* to a *matrix*. This is an operation that is undefined. While this operation is undefined, numerical libraries treat this as a legitimate operation by *broadcasting* the vector b_{n_h} to $B_{m \times n_h}$ [2, 11].

The activation matrix A contains the activation vector as a row for each training example and is expected to be $A \in \mathbb{R}^{m \times n_h}$ since it will be passed onto to the next layer as input.

4.3 MLP

The `MLP` encapsulates the `DenseLayer` objects and implements a number of methods to approximate the function for a system.

The `__init__` method takes as arguments the dimension of the input n_x, the number of units n_h per hidden layer, the number of targets or classes n_y, the learning rate η, the total number of hidden layers l_h, the cost function C as a string, activation function ϕ for hidden layers as a string, and the activation function for the target layer φ as a string. This method also creates three `list` objects of particular important: `sequential`, `activations_cache`, and `weighted_inputs_cache` stored collectively as a `@property` called `cache`. These `list` objects are used to track the layers of the model, the activation vectors a for a forward pass, and the weighted input vectors z for a forward pass.

The `fit` method takes inputs X, targets y, the batch size m, and the number of epochs to train the model. Batching of the data occurs first. This can be easily accomplished using `numpy.random.choice` with the range of indices to sample from being the total number of training examples, passing `replace=False`, and setting `size=(num_batches, batch_size)`. The number of batches is $floor(\frac{\text{training examples}}{\text{batch size}})$.

The training loop is also quite simple: (1) make predictions, (2) compute the scalar loss[6], (3) compute the gradients, and (4) update the weights and biases layer by layer.

The `_forward_pass` first clears the cache of saved activations and weighted input vectors from the previous pass. Then each layer in the `sequential` list is called to compute and the cache weighted inputs and activations a^{l-1} as

[6] Only used to track model performance, not used to update gradients.

the input to the `__call__` method of each `DenseLayer`. Finally, the `__call__` of the last `DenseLayer` returns the last activation vector a^L representing the predictions the model made given the samples.

4.3.1 Backpropagation Pseudocode

The `_backward_pass` is the only pseudocode I will provide since it is really the only step that is confusing due to the use of local `list` objects and the creation of ragged tensors, which are just nested variable length lists. The ragged tensors are the natural result of different weight and bias tensor shapes of hidden layers and the output layer. I also specify the shapes of the tensors in the pseudocode for clarity. A *?* is used in describing the variable length dimension of a tensor. Also, n_l is the number of layers in the network (including the input layer) while n_x generically refers to the number of neurons in the previous layer.

Algorithm 1 Computation of gradients for weight and bias parameters via backpropagation

Initialize tensors
$A, Z \leftarrow$ `cache` ▷ $A \in \mathbb{R}^{n_l \times m \times ?}; Z \in \mathbb{R}^{n_l \times m \times ?}$
$Y \leftarrow \text{AssertAtLeast2D}(Y)$ ▷ Make sure at least $Y \in \mathbb{R}^{1 \times n_y}$

Compute errors for last layer
$A_{L,:,:} \leftarrow A_{-1,:,:}$ ▷ Get the last layer activation tensor
$Z_{L,:,:} \leftarrow Z_{-1,:,:}$ ▷ Get the last layer weighted input tensor
$\nabla_{b^L} C_{\text{samples}} \leftarrow \{\}$ ▷ Empty $\nabla_{b^L} C_{\text{samples}} \in \mathbb{R}^{m \times n_y}$
$\nabla_{b^L} C_{\text{samples}} \leftarrow \text{GradBiases}(A_L, Z_L, Y)$ ▷ Bprop eq. 1 $\forall b \in m$

$A_{L-1,:,:} \leftarrow A_{-2,:,:}$ ▷ Get the second to last layer activation tensor.
$\nabla_{W^L} C_{\text{samples}} \leftarrow \{\}$ ▷ Empty $\nabla_{W^L} C_{\text{samples}} \in \mathbb{R}^{m \times n_y \times n_h}$
$\nabla_{W^L} C_{\text{samples}} \leftarrow \text{GradWeights}(A_{L-1}, \nabla_{b^L} C_{\text{samples}})$ ▷ Bprop eq. 4 $\forall b \in m$

Backpropagate errors
$\nabla_b C_{\text{layers}} \leftarrow \{\}$ ▷ Empty $\nabla_b C_{\text{layers}} \in \mathbb{R}^{n_l \times m \times ?}$
$\nabla_W C_{\text{layers}} \leftarrow \{\}$ ▷ Empty $\nabla_W C_{\text{layers}} \in \mathbb{R}^{n_l \times m \times ? \times ?}$

$\nabla_b C_{\text{layers}}[-1, :, :] \leftarrow \nabla_{b^L} C_{\text{samples}}$ ▷ Update the last element of the bias cost tensor
$\nabla_W C_{\text{layers}}[-1, :, :, :] \leftarrow \nabla_{W^L} C_{\text{samples}}$ ▷ Update the last element of the weight cost tensor

for $l = (L-1), ..., 0$ **do** ▷ For all remaining layers, backpropagate error.
    $\nabla_{b^l} C_{\text{samples}} \leftarrow \{\}$ ▷ Empty $\nabla_{b^l} C_{\text{samples}} \in \mathbb{R}^{m \times n_h}$
    $\nabla_{W^l} C_{\text{samples}} \leftarrow \{\}$ ▷ Empty $\nabla_{W^L} C_{\text{samples}} \in \mathbb{R}^{m \times n_h \times n_x}$

    $W^{l+1} \leftarrow$ `sequential[l+1].W`
    $\frac{\partial \phi^l}{\partial z}(\cdot) \leftarrow$ `sequential[l].activation.derivative`

    for $b = 0, ..., m$ **do**
        $\delta^{l+1} \leftarrow \nabla_b C_{\text{layers}}[l+1, b, :]$
        $z^l \leftarrow Z[l, b, :]$
        $a^{l-1} \leftarrow A[l-1, b, :]$

        $\nabla_{b^l} C_{\text{sample}} \leftarrow \text{GradHiddenBiases}(W^{l+1}, \delta^{l+1}, z^l, \frac{\partial \phi^l}{\partial z}(\cdot))$
        $\nabla_{W^l} C_{\text{sample}} \leftarrow \text{GradWeights}(a^{l-1}, \nabla_{b^l} C_{\text{sample}})$

        $\nabla_{b^l} C_{\text{samples}}.\text{append}(\nabla_{b^l} C_{\text{sample}})$
        $\nabla_{W^l} C_{\text{samples}}.\text{append}(\nabla_{W^l} C_{\text{sample}})$
    end for

    $\nabla_b C_{\text{layers}}[l, :, :] \leftarrow \nabla_{b^l} C_{\text{samples}}$
    $\nabla_W C_{\text{layers}}[l, :, :, :] \leftarrow \nabla_{W^l} C_{\text{samples}}$
end for

return $\nabla_b C_{\text{layers}}, \nabla_W C_{\text{layers}}$

5 Applications

Equally important as theory, applications and experimentation demonstrate potential uses for the universal function approximator called the neural network. Here I describe a regression task and binary classification task, as well as the means by which implementations can be compared.

5.1 Regression Tasks

The derivative of the cost function and discussion in previous parts of this paper is focused on supervised learning problems in $\mathbb{R}$, so no extensive explanation of the regression task shall be broached. Suffice to say, the regression task is concerned with outputting a prediction tensor whose elements are in $\mathbb{R}$.

5.1.1 Diabetes Dataset

The diabetes dataset is available through `Scikit-Learn`, though the original data is available from Efron et al..

$$\mathbb{X} \in (-0.2, 0.2) \tag{34}$$
$$\mathbb{Y} \in [25..\ 346] \tag{35}$$
$$X \in \mathbb{X}^{442\times 10} \tag{36}$$
$$y \in \mathbb{Y}^{442\times 1} \tag{37}$$

Data was standard scaled using `Scikit-Learn`.

5.1.2 Regressor Architecture

The architecture of the network is as follows: a single hidden layer with ReLU activation function and 32 neurons; an output layer with linear activation function and 1 neuron; the MSE cost function; batch size of 32; trained for 5 epochs; update parameters with stochastic gradient descent; and learning rate of 0.01.

5.2 Binary Classification Tasks

A binary classification task takes some input x and produces a prediction $\hat{y}$ that is the probability (or logit, aka unscaled log probability) that x belongs to the class $y \in \{0, 1\}$.

The binary cross entropy (BCE) cost function for a problem such as this over m examples in a batch is shown below. Note that i denotes the i^{th}prediction of batch *not* the i^{th} neuron of a layer since there are not multiple neurons in the last layer of the network.

$$
\begin{aligned}
C &= \frac{-1}{m} \sum_{i=0}^{m-1} C_i \\
&= \frac{-1}{m} \sum_{i=0}^{m-1} y_i \ln\left(a_i^L\right) + (1 - y_i) \ln\left(1 - a_i^L\right)
\end{aligned}
\tag{38}
$$

The partial derivative for a single sample $\frac{\partial C_i}{\partial a_0^L}$ is needed, and no gradient is used since the gradient is, by definition, at least a rank-1 tensor. Why is no gradient computed? It is because there is only a single neuron in the output layer of a binary classification problem. Gradients were needed in previous problems to compute the initial errors of the parameters in the last layer. This is because the output layer of previous problems is a vector of n_y neurons where $n_y \geq 1$ and therefore the gradient was $\nabla_{a^L} C = \langle \frac{\partial C}{\partial a_0^L}, \frac{\partial C}{\partial a_1^L}, \cdots, \frac{\partial C}{\partial a_{(n_y-1)}^L} \rangle$. The pseudocode still works for when $n_y = 1$; however, the algorithm must generalize to multiple dependent variables in a regression problem. Comparatively, the output of the forward pass for a single sample x for a binary classification problem is always a single value for the probability of the sample x belonging to the $class = 1$. Thus, the partial derivative of BCE for a single sample is shown below:

$$
\frac{\partial C_i}{\partial a_0^L} = -1\left(\frac{y}{a_0^L} - \frac{1 - y}{1 - a_0^L}\right) \tag{39}
$$

5.2.1 Breast Cancer Dataset

The breast cancer dataset is available through `Scikit-Learn`, and the original data is available from the UCI Machine Learning repository.

$$
X \in \mathbb{R}^{569 \times 30} \tag{40}
$$

$$
\mathbb{Y} \in \{0, 1\} \tag{41}
$$

$$
y \in \mathbb{Y}^{569} \tag{42}
$$

The input data X was standard scaled using `Scikit-Learn`.

5.2.2 Classifier Architecture

The architecture of the network is as follows: a single hidden layer with ReLU activation function and 32 neurons; an output layer with sigmoid activation function and 1 neuron; the BCE cost function; batch size of 32; trained for 5 epochs; update parameters with stochastic gradient descent; and learning rate of 0.01.

Note that my implementation is not the optimized implementation used by TensorFlow, see [12] and the binary cross entropy TensorFlow backend source code.

5.3 Model Comparison

Cross validation using K-fold cross validation, which is detailed on page 118 of [2], allows one to compute a mean of losses across K folds rather than just a single sample. The randomization of training and testing data allows for a better estimate of the generalization error of the model, that is the capability of the model to perform well on data it has never seen before. Likewise, confidence intervals indicate how likely the population mean of a particular metric falls within a given range. Cross validation and the subsequent construction of confidence intervals allows for robust model comparison.

6 Results and Discussion

The performance of my implementation of the multilayer perceptron on the regression and classification tasks is shown below.

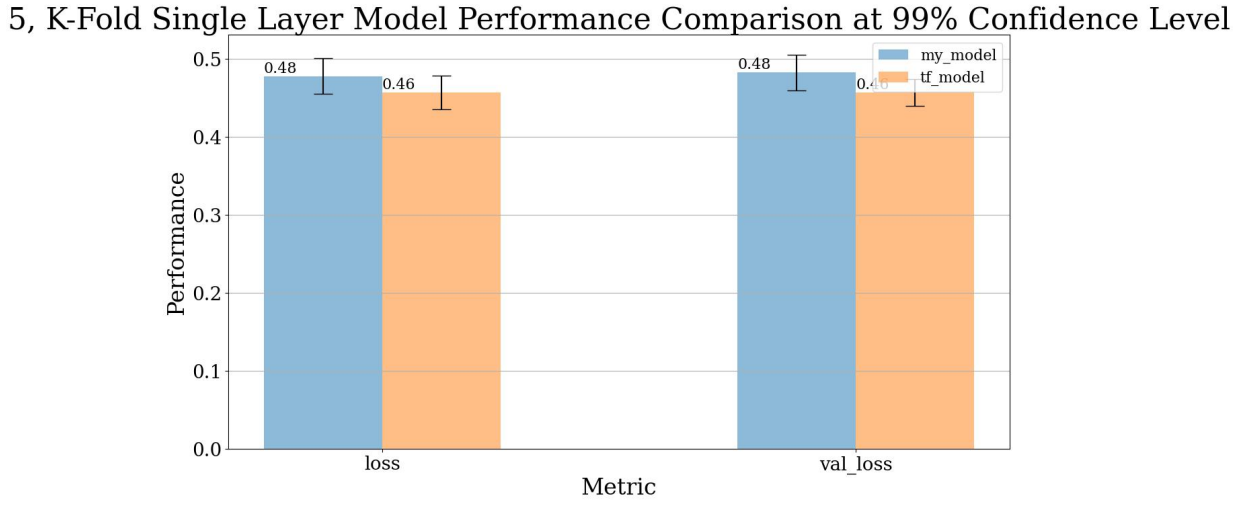

Figure 2: Performance of model on diabetes regression task.

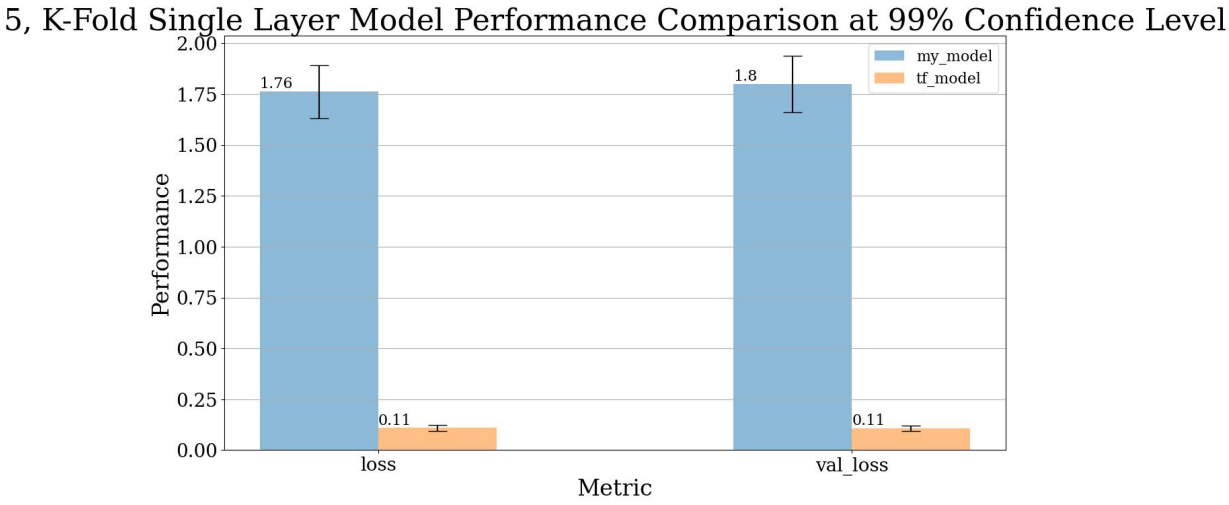

Figure 3: Performance of model on breast cancer classification task.

My implementation with respect to the regression task is within the bounds of the TensorFlow reference implementation, which indicates that my implementation sufficiently approximates that of a known, optimized implementation.

My implementation with respect to the classification task is well outside the bounds of the TensorFlow reference implementation, but this is likely due to optimization and smoothing considerations that I did not implement for the sake of simplicity. Nevertheless, my model's performance is *reasonable*, if not ideal, and sufficiently demonstrates the concepts such as the derivative of a given cost function with respect to its weights, activations, biases, etc.

7 Conclusion

In this report, I elucidate the data types and language of neural networks, as well as employ fairly standard notation that adheres to that suggested by *notation.pdf* taken from Stanford's Deep Learning course cs230. The elementary operations of the neural network such as affine transformations, activation functions, and cost functions are also discussed. The relevance of these functions with respect to neural network learning via backpropagation and gradient descent is also illustrated with an admittedly long-winded explanation of the fundamental equations of backpropagation and the relevant shapes of the operands. The neural network theory is followed by an overview of the Python implementation, as well as the pseudocode supplied for an unoptimized backpropagation algorithm. The applications of the multilayer perceptron to simple regression and classification tasks, as well as a brief discussion on comparing models is demonstrated. This work served as introspective evaluation of my understanding of neural network operations, and its writing helped clarify many previously abstracted concepts. Future work could entail optimization of the algorithm by calling C programming language code from Python, as well as the addition of multi-class or multi-class multi-label classification problems.

References

[1] 3Blue1Brown: Grant Sanderson. *But What is a Neural Network — Chapter 1, Deep Learning*. YouTube. 2017. URL: https://www.youtube.com/watch?v=aircAruvnKk.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[3] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/index.html.

[4] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2020.

[5] Francois Chollet. *Deep Learning with Python, Second Edition*. Manning Publications, 2021.

[6] nmasanta (https://math.stackexchange.com/users/623924/nmasanta). *What is the difference between variable, argument and parameter?* Mathematics Stack Exchange. URL: https://math.stackexchange.com/q/3562026.

[7] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2014. arXiv: 1312.6114 [stat.ML].

[8] Eric Weisstein. *Parameter*. MathWorld–A Wofram Web Resource. URL: https://mathworld.wolfram.com/Parameter.html.

[9] *The Gradient*. 2021. URL: https://math.libretexts.org/Bookshelves/Calculus/Supplemental_Modules_(Calculus)/Vector_Calculus/1%3A_Vector_Basics/2%3A_The_Gradient.

[10] Xavier Glorot and Yoshua Bengio. "Understanding the Difficulty of Training Deep Feedforward Neural Networks". In: ed. by Yee Whye Teh and Mike Titterington. Vol. 9. PMLR, Dec. 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[11] Charles R. Harris et al. *Broadcasting*. Broadcasting - NumPy v1.22 Manual. 2021. URL: https://numpy.org/doc/stable/user/basics.broadcasting.html.

[12] Rafay Khan. *How do tensorflow and Keras implement binary classification and the binary cross-entropy function?* Dec. 2020. URL: https://rafayak.medium.com/how-do-tensorflow-and-keras-implement-binary-classification-and-the-binary-cross-entropy-function-e9413826da7.