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

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.

# 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 that is often "handwaved" to the extreme or explained in so much detail as to be utterly confusing–at least 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 $W \in \mathcal{R}^{n_h \times n_x}$ and a bias vector $b \in \mathcal{R}^{n_h}$ on the input vector $x \in \mathcal{R}^{n_x}$. 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 $x \in \mathcal{R}^{n_x}$ 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 transposition (converts a *column vector* to a *row vector* and is denoted with a superscript of $\top$) is also shown.

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

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 $W \in \mathcal{R}^{n_h \times n_x}$ is shown below.

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

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 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 The Dense/Fully Connected Layer

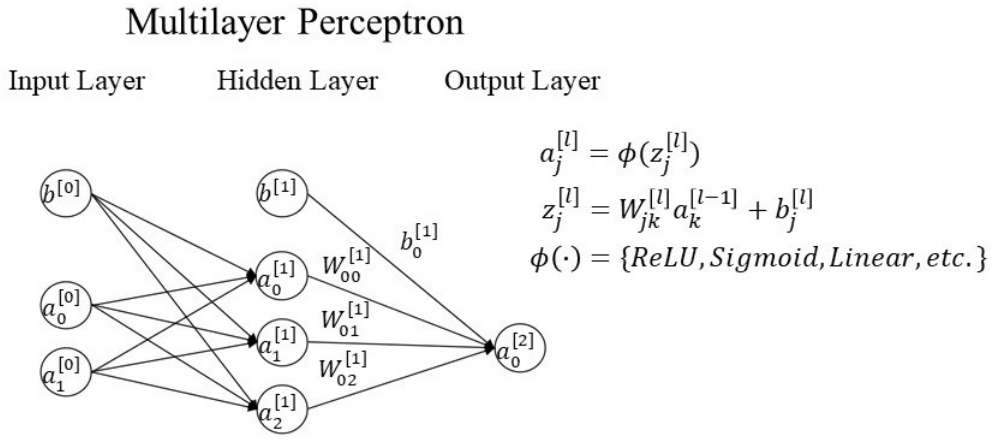
The affine transformation, which is the most fundamental transformation of the densely/fully connected layers that exist in the MLP, yields a weighted input vector $z^l$ with elements $z_j^l = W_{jk}^l a_k^{l-1} + b_j^l$ for a layer $l$ and neuron $j$. Here, the activation $a_k^{l-1}$ denotes the activation of the $k^{th}$ neuron of the previous layer $(l-1)$. Importantly, the input layer has no activation function $\phi$ 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 $a \in \mathcal{R}^{n_a}$ is clearly not a matrix, numerical libraries will often treat a vector $v \in \mathcal{R}^{n_v}$ as equivalent to a matrix with a single column (i.e., $V \in \mathcal{R}^{n_v \times 1}$) 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 $\phi$ 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 $\sigma$ 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 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 sometimes it is denoted $\mathcal{L}$ or $J$. 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. However, the cost for a single sample

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




$C_x$ is a vector. The scalar value can be used to estimate performance during training; however, for my implementation, the cost vector function $C_x$ is used for learning MLP parameters.

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

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 $\theta$ 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 $\frac{\partial C_x}{\partial a_j^L}$. 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 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[2]. Since $W$ and $b$ are tensors, the gradients[3] of the cost function with respect to these parameters ($\frac{\partial C_x}{\partial W^l_{jk}}$ and $\frac{\partial C_x}{\partial b^l_j}$)) 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*" [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.

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

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

$$W^l = W^l - \eta \nabla_{W^l} C \tag{8}$$

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

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

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 $\eta$, 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 of the model by a small amount until convergence[4] 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.

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 directly (i.e., $\nabla_W C$ or $\nabla_b C$). Rather, the components of the gradient are computed using the last two equations listed below.

$$\delta^L = \nabla_a \mathrm{C} \circ \frac{\partial \phi}{\partial z}(z^L) \tag{11}$$

$$\delta^l = ((W^{l+1})^\top \delta^{l+1}) \circ \frac{\partial \phi}{\partial z}(z^l) \tag{12}$$

$$\frac{\partial \mathrm{C}}{\partial b^l_j} = \delta^l_j \tag{13}$$

$$\frac{\partial \mathrm{C}}{\partial W^l_{jk}} = \delta^l_j (a^{l-1}_k)^\top \tag{14}$$

The error vector $\delta^L$ for the last layer $L$ uses the gradient operator $\nabla_a$ with respect to each activation $a_j$ for the $j^{th}$ neuron of the layer $L$. The gradient

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

for a function of two variables can be succinctly exemplified with the following equations:

$$f(x, y) = x^2 y \tag{15}$$

$$\nabla f(x, y) = \langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \rangle \tag{16}$$

$$\nabla f(x, y) = \langle 2xy, x^2 \rangle \tag{17}$$

That is, the gradient is the vector where each element is the first-order partial derivative with respect to each variable in the function.

Now, $\nabla_a C$ specifies $a$ because $C$ can be written as a function of many parameters and variables. For example, with a single layer neural network, that is a network with two weight matrices $W^0$ and $W^1$, two bias vectors $b^0$ and $b^1$ can be written as $C(W^0, W^1, b^0, b^1, a^0, a^1)$. Consider the activation vector for a layer $l$ over $j$ neurons as $a = \{a_j^l\}_{j=0}^{(n_h-1)}$. This is written as

$$a = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{(n_h)-1} \end{bmatrix} \tag{18}$$

So if you reduce the problem to the element-by-element case, i.e.

$$\delta^L = \nabla_a \mathrm{C} \circ \frac{\partial \phi}{\partial z}(z^L) \tag{19}$$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial \phi}{\partial z}(z_j^L) \tag{20}$$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}(a_j^L) \frac{\partial \phi}{\partial z}(z_j^L) \tag{21}$$

it becomes clear that you must compute $\frac{\partial C}{\partial a_j^L}$.

Assuming $C$ is for a single training vector $x$, then

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

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

and the partial derivative returns a scalar, but no summation is taken since a summation over a vector would return a scalar, and the desired result of the gradient is a vector.

The gradient must therefore be taken with respect to a certain parameter, otherwise known as the Jacobian matrix, the partial derivative with respect to each variable of a vector valued function, would result. To visualize what the gradient vector with respect to the activation only 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 $\theta^L$ are the relevant parameters and variables for the layer layer $L$. Such a matrix would be as follows:

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

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 the $\frac{\partial C}{\partial a_j^L}$ as just another function, which it is in fact, 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 the $a_j^L$ 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, on which the error of the network is based.

The practical implementation of the four equations will be explained shortly after showing the *gradient descent* equations below.

$$W^l = W^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^\top \tag{26}$$

$$b^l = b^l - \frac{\eta}{m} \sum_x \delta^{x,l} \tag{27}$$

The learning rate $\eta$ 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, somewhat out of order, the Python classes that will 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.

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 $W \in \mathcal{R}^{n_h \times n_x}$, and the bias $b \in \mathcal{R}^{n_h}$. The `DenseLayer` is correspondingly passed arguments for $n_x$, $n_h$, and an `Optional[Callable]` that will be used as the vectorized activation function $\phi(\cdot)$ for a given layer. The `__init__` method saves the `Optional[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

$$WeightedInputZ = Wx^\top + b \tag{28}$$

$$ActivationA = \phi(WeightedInputZ) \tag{29}$$

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

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

$$Z_{n_h \times m} = W_{n_h \times n_x}(X_{m \times n_x})^\top + b_{n_h} \tag{30}$$

For this weighted input matrix $Z_{n_h \times m}$, you may observe a problem with how it is computed: a *vector* is *added* to a *matrix*. This is an operation that is undefined. The matrix-vector addition occurs because $W_{n_h \times n_x}(X_{m \times n_x})^\top$ returns a matrix $C_{n_h \times n_x}$ and the vector $b_{n_h}$ is then added to $C_{n_h \times n_x}$ such that $W_{n_h \times n_x}(X_{m \times n_x})^\top + b_{n_h} = C_{n_h \times n_x} + b_{n_h}$. While this operation is undefined, numerical libraries treat this as a legitimate operation by *broadcasting* the vector $b_{n_h}$ to $B_{n_h \times m}$ [2, 11]. One can imagine that the column vector $b_{n_h}$ is copied and into adjacent columns until there are $m$ copies of the column vector in what is now a $n_x \times m$ matrix $B$.

The activation matrix (i.e., activation vector for each training example) $A$ will also be $A \in \mathcal{R}^{n_h \times m}$. Since the activations are passed onto the next layer as the new input $X$ to the `__call__` method, one must reshape the matrix to be the expected input.

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, `derivative` method. All children of the `Operation` must implement these methods. However, if the child is a *cost function*, then this child class must also implement a `gradient` method ($\nabla_a \mathrm{C}$).

# 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.