

A Structured Approach to Neural Networks

By Scott Raber

www.raberfamily.com

Introduction

A vector-based description of a Fully Connected Neural Network will be presented to provide a solid and easy to understand mathematical foundation. By using some simple Linear Algebra, complicated summation equations can be avoided, making the mathematics much cleaner and clearer. This is important because the mathematical underpinnings of a Fully Connected Neural Network form the basis for other kinds of connectionist networks such as Convolutional Neural Networks. Neural Networks have free parameters that must be solved for which is referred to as *training*. The kind of training that will be discussed is based on having a set of input data with associated expected results. This is known as Structured Learning. Network parameters are adjusted in an attempt to produce the expected results relative to associated inputs. To know how to make the adjustments, a function called a Loss function (a.k.a. Cost function or Objective function) is devised to measure the network output error and adjustments are made to the network parameters in an effort to minimize the error. This paper will focus on the Gradient Descent method to minimize the Loss function and a simple study of a Loss function Error Surface is conducted to give an appreciation for how the method works. To apply Gradient Descent the Error Surface gradient is needed, and to determine that value the Back-Propagation method will be used. The Back-Propagation method will be derived from the vector-based network equation.

There is a companion C++ library that mirrors closely the structure of the mathematics described in this paper. It is called the SimpleNet library as it is written to be easily readable and extensible. The paper examines the implementation accuracy of the library, and its structure is discussed, but the paper is not about the SimpleNet library. It is available to you and documentation for it is provide in an appendix if you are interested, but the paper stands on its own.

The target audience for this paper is the person interested in the underlying mathematics of Neural Networks. To fully benefit from the paper a reader should at least be familiar with basic linear algebra and differential calculus. Regarding the former, knowing how to multiply a matrix by a vector is enough. Regarding the later, if you know, or once knew what a function derivative is, that should be enough to follow along.

Mathematical Concepts, Nomenclature, and Conventions

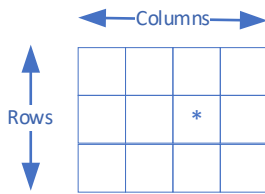
The mathematical concepts of Neural Networks are not complex but require strict and consistent application of basic mathematical principles. At times I will utilize nomenclature that is unique to this paper in an attempt to suppress ambiguity in the underlying principle. In this section I will describe the concepts needed to develop the mathematics of Neural Networks.

Linear Algebra Concepts:

This may seem rudimentary, but it is important to agree on every detail and convention to understand how to construct a Neural Network solver. In this section we will review some basic concepts of linear algebra to establish conventions that might be ambiguous to the reader if left unaddressed.

Matrix:

A matrix is a set of numbers arranged in a block. In this paper Rows indicate the number of horizontal rows and Columns indicate the number of (vertical) columns. The matrix below has 3 rows and 4 columns. We will always refer to rows first and columns second (rows x columns). The matrix below is 3 x 4, read "three by four". A **bold** font will be applied to a variable to indicate that it is a matrix.



A matrix element is identified by its' row position, then its' column position. In the figure above, the element with the star is element (2,3) using a unit offset convention. When coding these algorithms, a zero offset convention might be needed. The point here is that we refer to the element position by row and then column.

Column Vector:



A Column Vector has 1 column and R rows. It is just a matrix with one column. The Column Vector to the left is a 3 x 1 Column Vector.

A Column Vector Variable is denoted with a Right Arrow Accent

We will use the right Arrow accent over a variable to indicate that it is a Column Vector. If a variable is simply called a Vector, we will assume that it is a Column Vector. For example, the vector X is written \vec{X} .

Row Vector:



A Row Vector is a matrix with 1 row and C columns. The Row Vector to the left is 1 x 4. The transpose operator (T) turns a Column vector into a Row Vector. For example,

\vec{X}^T is a Row vector.

A Row Vector Variable is denoted with a Left Arrow Accent

Applying the transpose operator to a Column vector is a cumbersome way to indicate that a variable is a Row vector, and it can lead to inconsistencies when coupled with more complex mathematical operations. We need a way to just say: "this is a Row vector". In this paper we will use the left Arrow accent to indicate that a variable is a Row vector. For example, the Row vector Y is written \overleftarrow{Y} .


Basic Matrix Multiplication Rules

When performing matrix multiplication keep this rule in mind, the column number of the first operand must be equal to the row number of the second operand. When multiplied, the row number of the first operand is combined with the column number of the second operand to yield the shape of the result.

There are two matrix multiplication operations you must understand for the following mathematical development.

The first is a Row Vector times a Matrix.

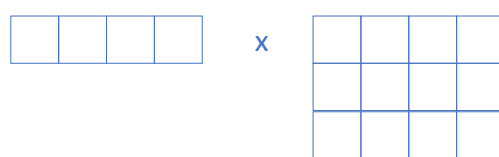
Example:

$$(1 \times 3) \quad \times \quad (3 \times 4) \quad = \quad (1 \times 4)$$


The diagram illustrates the multiplication of a 1x3 row vector (represented by three adjacent boxes) by a 3x4 matrix (represented by a 3x4 grid). The result is a 1x4 row vector (represented by four adjacent boxes).

Example of an invalid operation:

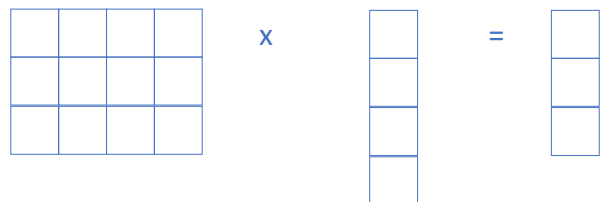
The following operation is not valid.

$$(1 \times 4) \quad \times \quad (3 \times 4)$$


The diagram shows a 1x4 row vector (four boxes) multiplied by a 3x4 matrix (3x4 grid). A red arc connects the '4' in the first dimension of the row vector to the '3' in the first dimension of the matrix. A red arrow points from this arc to the text: "These dimensions are not equal! The operation is invalid."

The other operation we will use is a Matrix times a Column Vector.

Example:

$$(3 \times 4) \quad \times \quad (4 \times 1) \quad = \quad (3 \times 1)$$


The diagram illustrates the multiplication of a 3x4 matrix (3x4 grid) by a 4x1 column vector (four vertically stacked boxes). The result is a 3x1 column vector (three vertically stacked boxes).

Function Conventions

Here we will establish the nomenclature for analytic functions used in this paper.

A scalar function f with vector input is designated $f(\vec{X})$. The function f takes a vector as input and returns a scalar value.

A vector function f with vector input is designated $\vec{f}(\vec{X})$. The function \vec{f} takes a vector as input and returns a vector value.

The Gradient of a Scalar Function

Recall from vector calculus that the gradient of a function such as $f(\vec{X})$ points in the direction where $f(\vec{X})$ is increasing most significantly. The gradient is computed by application of the Grad operator, commonly designated by a special character called Del and denoted by an upside-down delta (∇) (1). The Del operator in its most generic form is given by:

$$\nabla = \hat{e}_1 \frac{\partial}{\partial x_1} + \hat{e}_2 \frac{\partial}{\partial x_2} + \dots + \hat{e}_N \frac{\partial}{\partial x_N}$$

where N is the space of the independent variable (\mathbb{R}^N), and \hat{e}_i are the axes of \mathbb{R}^N . In \mathbb{R}^3 , \hat{e}_i denote the X, Y, and Z axes. The Del operator can be written more compactly by placing the axes of the space in the hyper-vector \vec{e} , and the partial derivative operators in the **Row vector** $\overleftarrow{\left(\frac{\partial}{\partial \vec{X}}\right)}$. A hyper-vector is a vector with elements that are themselves a vector or even a matrix and I make this distinction to account for the fact that \hat{e}_i is a unit vector (2). If the thought of a hyper-vector is causing you concern do not worry as it will not be around for long. Using it allows us to write:

$$\nabla = \overleftarrow{\left(\frac{\partial}{\partial \vec{X}}\right)} \vec{e}.$$

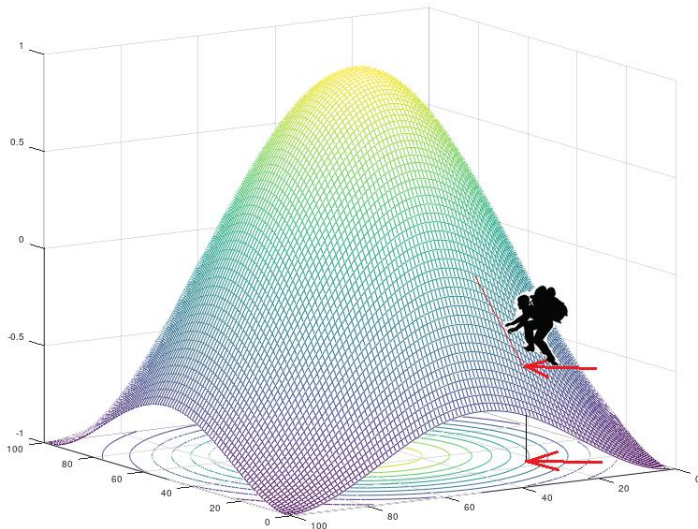
Using the compact form of Del, the gradient of $f(\vec{X})$ where $\vec{X} \in \mathbb{R}^N$, is given by:

$$\nabla f(\vec{X}) = \overleftarrow{\left(\frac{\partial f}{\partial \vec{X}}\right)} \vec{e} = \hat{e}_1 \frac{\partial f}{\partial x_1} + \hat{e}_2 \frac{\partial f}{\partial x_2} + \dots + \hat{e}_N \frac{\partial f}{\partial x_N}.$$

In this paper I will call a term such as $\overleftarrow{\left(\frac{\partial f}{\partial \vec{X}}\right)}$, the *Gradient Vector of f*, to distinguish it as a separate operation that is part of the gradient computation. There is some ambiguity in the literature about the resultant form of the *Gradient Vector*. Some authors show the result as a Column vector, others apply the transpose operator to a Column vector, and some authors specifically say that the result is a Row vector. As you will see, the choice of Row vector dovetails naturally with the computation of the Jacobian matrix which we will discuss in the next section.

In this paper the Del operator will sometimes be written with a vector variable subscript to designate which independent vector variable the Del operator pertains to. When applied to a scalar function with one vector input it is not needed, but in the mathematics that follow, we will be dealing with scalar functions with multiple vector inputs and in this case the Del subscript notation is helpful. For example, consider a function of the vectors \vec{X} and \vec{W} , given by $f(\vec{X}, \vec{W})$, where we are interested in the gradient with respect to \vec{W} . This is written as $\nabla_{\vec{W}} f(\vec{X}, \vec{W})$.

To help make it clear what the Del operator does, consider this example, if $f(\vec{X})$ defines the surface of a



mountain, and \vec{X}_0 denotes the place you are standing on the mountain, $\nabla f(\vec{X}_0)$ points in the direction of steepest ascent relative to your position. Note that in the mountain analogy, the gradient of $f(\vec{X})$ doesn't point up the mountain, it points into the mountain in the direction you should travel to go up the fastest, meaning gain the most elevation in the fewest steps. When I am hiking on a mountain this is the direction I usually avoid!

In the section above you may have noticed the word “vector” used quite a bit. The Del operator produces a vector when it is applied to a scalar function and each element of the resulting vector is accompanied by an axis direction \hat{e}_i . Yet input variables such as \vec{X} and \vec{W} are called vectors, or more precisely, Column vectors, without such axis designations. If we were to look in any good Linear Algebra textbook, we would see that Column vectors and Row vectors are concepts that are supported by coordinate systems. The elements of vectors found in linear algebra are implicitly associated with coordinate axes. The inconsistency comes from the collision of the subject of Vector Calculus with the subject of Linear Algebra. Fortunately, things can be clarified quite easily. Consider the Grad of $f(\vec{X})$

$$\nabla_{\vec{X}} f(\vec{X}) = \overleftarrow{\left(\frac{\partial f}{\partial \vec{X}}\right)} \vec{e} = \hat{e}_1 \frac{\partial f}{\partial x_1} + \hat{e}_2 \frac{\partial f}{\partial x_2} + \hat{e}_3 \frac{\partial f}{\partial x_3}$$

where $\vec{X} \in \mathbb{R}^3$.

We only need to recognize that $\overleftarrow{\left(\frac{\partial f}{\partial \vec{X}}\right)}$ is a Row vector in the Linear Algebra sense and once we start to work with that term, we can drop explicit use of the coordinate axis vector \vec{e} , it is implied. Continuing with the example above the Gradient Vector is given by:

$$\overleftarrow{\left(\frac{\partial f}{\partial \vec{X}}\right)} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{bmatrix}$$

The Jacobian Matrix

The Jacobian matrix can be thought of as repeated application of the Gradient partial derivative operator $\overleftarrow{\left(\frac{\partial}{\partial \vec{X}}\right)}$ to each element of the vector function. The resulting Row vectors, which are the Gradient Vectors of each axis, are stacked one atop the other to form a matrix. The resulting matrix may be square or rectangular. The Jacobian matrix is given by:

$$J(\vec{f}(\vec{X})) = \frac{\overleftarrow{\partial}}{\partial \vec{X}} \vec{f}(\vec{X}).$$

Below is an example with input vector $\vec{X} \in \mathbb{R}^3$ and vector function $\vec{f} \in \mathbb{R}^2$.

$$J(\vec{f}(\vec{X})) = \begin{array}{|c|c|c|} \hline \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \hline \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \hline \end{array}$$

The Neural Network Layer

A neural network is a composition of nested vector functions commonly referred to as layers (3) (4) (5) (6). One layer of a neural network is defined by the following equation:

$$\vec{Y} = \vec{f}(\mathbf{W}\vec{X} + \vec{B})$$

where:

\mathbf{W} is a matrix of dimension $a \times b$ and is referred to as the **Weight matrix**.

\vec{X} is a Column vector of dimension $b \times 1$ and is referred to as the **Input vector**.

\vec{B} is a Column vector of dimension $a \times 1$ and is referred to as the **Bias vector**.

The vector function f returns a vector of dimension $a \times 1$ and is referred to as the **Activation function**.

\vec{Y} is a Column vector of the same dimension as f and contains the result of the Activation function.

Let's examine the shapes of the variables in this equation. The format is a bit awkward, but it will do. The equation has been rewritten in a stretched-out format with the variable shapes written above each variable. Remember, the shape is described by row x column.

$$(a \times 1) = (a \times 1) [(a \times b) * (b \times 1) + (a \times 1)]$$

$$Y = f [\mathbf{W} * X + B]$$

The shape of the input vector X is influenced by the data that we are trying to model. Later we will talk about interior layers, a.k.a. hidden layers, for now all you need to know is that those layers take as input the Y output of the previous layer, and in that case the dimension of X is specified by the output vector Y of the previous layer. Either way you look at it, the data that you want to input into the network layer specifies the shape of X , i.e., the value of the "b" dimension. That means it determines the number of columns of matrix \mathbf{W} . The "a" dimension specifies the dimension of the network layer output vector.

By completing the multiplication of $\mathbf{W} * X$ the picture becomes clearer. Let $U = \mathbf{W} * X$. The dimension of U is $a \times 1$ and we can see that the equation reduces to Column vectors that are all of dimension $a \times 1$ with the activation function f returning a vector of dimension $a \times 1$. In practice, Activation functions are

usually based on one simple scalar function that is applied to each element of the input vector. Whatever the dimension of the input vector, that is the dimension of the output vector of the Activation function.

$$(a \times 1) = (a \times 1)[(a \times 1) + (a \times 1)]$$

$$Y = f [U + B]$$

Multiple Layers

Most neural networks are multiple layers and are sometimes referred to as Fully Connected Feed Forward Neural Networks. Before we go any further, we will need to introduce another index to the neural network layer equation to keep track of the layers. There are many ways to do this, but in this paper, I will introduce a pre-subscript. It looks a bit weird but the good part about that is that it is easy to remember what it designates. As you can see, it is just a way to keep track of all the parts of the neural network equation.

$${}_k\vec{Y} = {}_k\vec{f}({}_k\mathbf{W}{}_k\vec{X} + {}_k\vec{B})$$

Now that we have the k index to keep track of the layers, we have a way to write N-layer neural networks such as the two-layer network example below.

$${}_1\vec{Y} = {}_1\vec{f}({}_1\mathbf{W}{}_1\vec{X} + {}_1\vec{B})$$

$${}_2\vec{X} = {}_1\vec{Y}$$

$${}_2\vec{Y} = {}_2\vec{f}({}_2\mathbf{W}{}_2\vec{X} + {}_2\vec{B})$$

Layer two is called a hidden layer. Consider the dimensions of the variables of these layers. Layer one has dimensions ${}_1a$ and ${}_1b$, layer two has dimensions ${}_2a$ and ${}_2b$, where ${}_2b = {}_1a$. Note that dimension ${}_2a$ is determined by the required output of the network, but ${}_1a$ is a free parameter. Interior ${}_ka$ dimensions are free parameters; you decide what they should be when you create the network. In general, the larger the value of ${}_ka$ in the interior (hidden) layers, the more capacity the network has to fit the non-linearities of a data set.

The two-layer example above is enough to capture the repeating pattern of a Feed Forward Neural Network and it should be clear to the reader how one might write a neural network with three or more layers. The pattern is easily extended to more layers and with suitable values for ${}_k\mathbf{W}$ and ${}_k\mathbf{B}$, a multi-layer neural network can model extremely complex relationships in data sets. And therein lies the challenge of neural networks. How do we find proper values for ${}_k\mathbf{W}$ and ${}_k\mathbf{B}$?

Neural Network Training

In the language of the Neural Network community, solving for suitable ${}_k\mathbf{W}$ and ${}_k\mathbf{B}$, is said to “train the network”. Let’s look at the neural network layer equation again. We’ll omit the k subscripts and only use them when we need to.

$$\vec{Y} = \vec{f}(\mathbf{W}\vec{X} + \vec{B})$$

When a trained network is applied, the input variable is \vec{X} and the result is \vec{Y} , which implies that we know \mathbf{W} and \vec{B} . However, to train a network we use a known set of \vec{X} 's and corresponding \vec{Y} values, to solve for \mathbf{W} and \vec{B} .

The Loss Function

To proceed we need additional nomenclature. We need to be able to differentiate the actual network output \vec{Y} from the desired output specified by the training data. We'll represent the desired output with a prime accent over the return variable like so: \vec{Y}' . Now we need a measure of the error between the network output and the desired output. We can define a function that takes \vec{Y} and \vec{Y}' as input and returns a scalar value that is a measure of the error. This function is often termed the *Loss function* and is given by:

$$E = L(\vec{Y}, \vec{Y}').$$

For a given \vec{X} , E is a measure of the difference between what the network outputs, \vec{Y} , and what we want it to output, \vec{Y}' . Now let's discuss the nomenclature for the training set. First, we will define a training pair, which we said is an input value \vec{X} and a corresponding desired network output \vec{Y}' . But training data comes in sets of training pairs, so to account for that we'll add a left superscript and write $({}^i\vec{X}, {}^i\vec{Y}')$, where the left superscript i identifies a specific pair. Note the input value is ${}^i_1\vec{X}$, the leading subscript k is fixed at 1, indicating that it is input into the first or "top" layer. A training data **set** is given by $\{({}^1_1\vec{X}, {}^1_1\vec{Y}'), ({}^2_1\vec{X}, {}^2_1\vec{Y}'), \dots, ({}^N_1\vec{X}, {}^N_1\vec{Y}')$, where N is the number of training samples. Given this definition of the training set, the Loss function can be rewritten:

$${}^iE = L(\vec{Y}({}_k\mathbf{W}, {}_k\vec{B}, {}^i_1\vec{X}), {}^i\vec{Y}')$$

where ${}_k\mathbf{W}$ and ${}_k\vec{B}$ are the free variables. ${}_k\mathbf{W}$ and ${}_k\vec{B}$ are often referred to collectively as *Weight Space* and within that space resides the *Error Surface* iE . For a given training pair, if we were to assign a range of values to each of the elements of ${}_k\mathbf{W}$ and ${}_k\vec{B}$, and with those values compute the error iE for every combination, it would form a surface in *Weight Space* which we call the *Error Surface*. The *Error Surface* is a hyper-surface for virtually all neural networks, and we usually have no hope of being able to visualize it, but the concept is very important.

The Loss Layer

The Neural Network training problem can be stated as follows:

Given a training set $\{({}^i_1\vec{X}, {}^i\vec{Y}')\}, i = 1 \text{ to } N$, and a network structure of k layers, find values for ${}_k\mathbf{W}, {}_k\vec{B}$ that minimize the *Average Error* defined by $E = \frac{1}{N} \sum_i {}^iE$.

The two-layer network example we saw above is modified for training by adding the Loss layer to the end of the computation. The complete set of equations are shown below.

$$\begin{aligned} {}_1\vec{Y} &= {}_1\vec{f}({}_1\mathbf{W} {}^i_1\vec{X} + {}_1\vec{B}) \\ {}_2\vec{X} &= {}_1\vec{Y} \end{aligned}$$

$$\vec{Y}_2 = \vec{f}_2(\mathbf{W}_2 \vec{X}_2 + \vec{B}_2)$$

$$E = L(\vec{Y}_2, \vec{Y}_2^{\text{target}})$$

Binary Classification Example

Before we discuss a technique for training a neural network let us look at an example simple enough that we can render the *Error Surface* and try to really understand what it represents (5). In this example a one-layer model will be used to solve a binary classification problem. The network is defined by:

$$\vec{Y}_1 = \vec{f}_1(\mathbf{W}_1 \vec{X}_1 + \vec{B}_1)$$

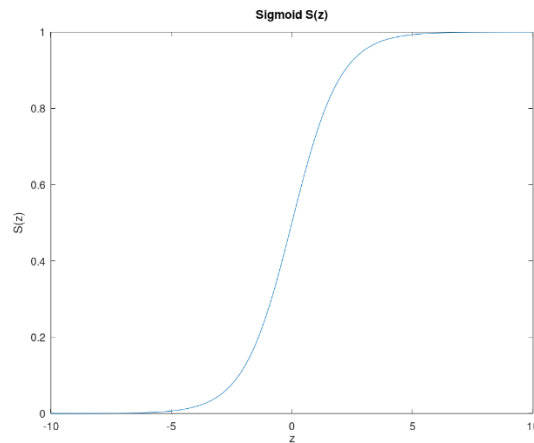
$$E = L(\vec{Y}_1, \vec{Y}_1^{\text{target}}),$$

where \vec{f}_1 is defined to be the *Sigmoid Activation* function and L is defined to be the *L2 Loss* function. I'll list some of the common Activation functions and Loss functions in an appendix, but here I will introduce the *Sigmoid Activation* function and the *L2 Loss* function.

The *Sigmoid Activation* function is given by:

$$\vec{S}(\vec{Z}) = \frac{1}{1+e^{-\vec{Z}}}, \text{ meaning that } S_i(\vec{Z}) = \frac{1}{1+e^{-Z_i}}.$$

The Sigmoid function approaches its minimum and maximum asymptotically, so this network will never output anything greater than one or less than zero.



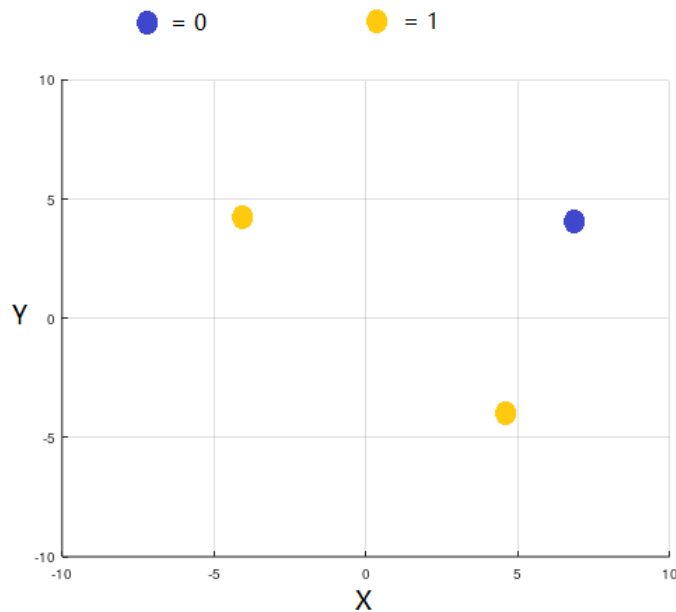
The *L2 Loss* function is given by:

$$L2(\vec{Y}_1, \vec{Y}_1^{\text{target}}) = \vec{Z}^T \vec{Z}, \text{ the Dot product of } \vec{Z} \text{ with itself, where } \vec{Z} = \mathbf{W}_1 \vec{X}_1 + \vec{B}_1.$$

Problem Definition:

Consider colored points placed on an X-Y Cartesian plane and restrict the color of a point to either yellow or blue. Codify the colors with the numeric values one and zero respectively. The choice of number assignment is not arbitrary, they are selected because they are the extremum of the *Sigmoid* function. The assigned values are commonly referred to as *labels*. The network input vector \vec{X}_1 will have two elements, one for the x-coordinate and one for the y-coordinate. The network output \vec{Y}_1 will contain a single value indicating the computed label. Given a set of points and labels, the network and

Loss layer defined above will try to define a surface, called the *Decision Surface*, which passes through each point label. Note that the Sigmoid function never attains a value of zero or one and only approaches those values asymptotically, so the *Decision Surface* can only try to get close to those values. In fact, our network is solving a logistic regression problem and we must use an interpretation rule outside of our model to make it a classifier. The rule could be something like: if ${}_1\vec{Y}$ is greater than 0.5 then the class label is 1, else the class label is 0. Don't be distracted by these details, the goal here is to use this simple example to introduce the concept of the *Decision Surface* and to explore the *Error Surface* of the network. To keep things tractable only three training points will be used and are shown graphically below.



Point	X	Y	Label
P1	6.5	4.0	0
P2	4.5	-4.0	1
P3	-4.0	4.5	1

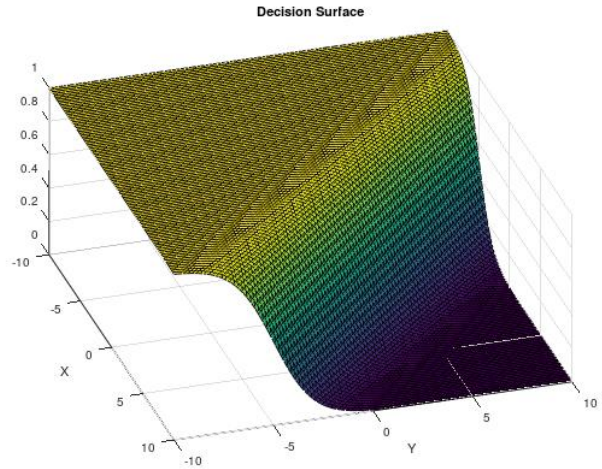
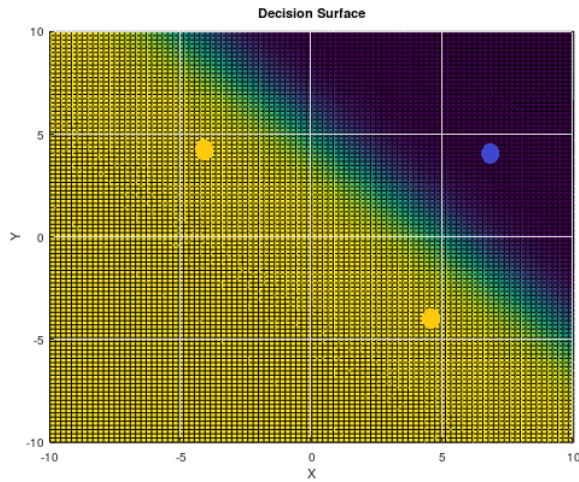
Let's review the details of the network layer used in this example. The problem definition states that the input vector (${}_1\vec{X}$) has two elements and the output vector (${}_1\vec{Y}$) has one element. That is everything we need to know to completely define the shape of ${}_1\mathbf{W}$ and ${}_1\vec{B}$. ${}_1\mathbf{W}$ must have two columns to accommodate the dimension of the input vector and must have one row to accommodate the dimension of the output vector. ${}_1\vec{B}$ must have one row to accommodate the size of the output vector. The *Weight Space* of this network is a 1 x 2 matrix $\begin{bmatrix} W_0 & W_1 \end{bmatrix}$ and a 1 x 1 Column Vector $\begin{bmatrix} B_0 \end{bmatrix}$.

The network was trained on the three points and resulted in the follow values:

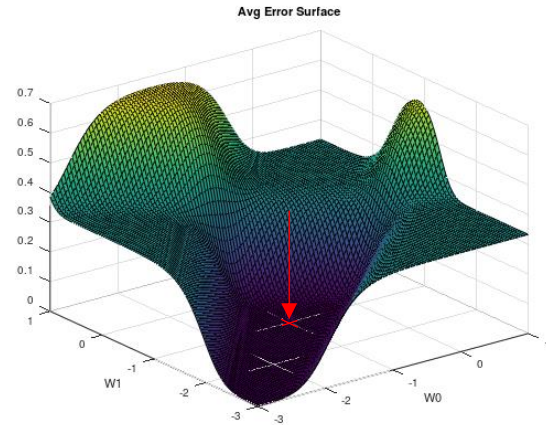
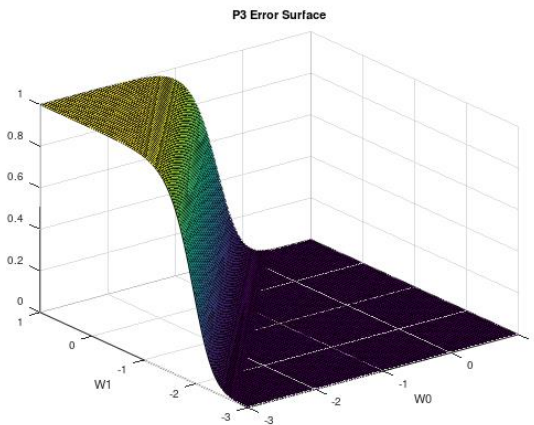
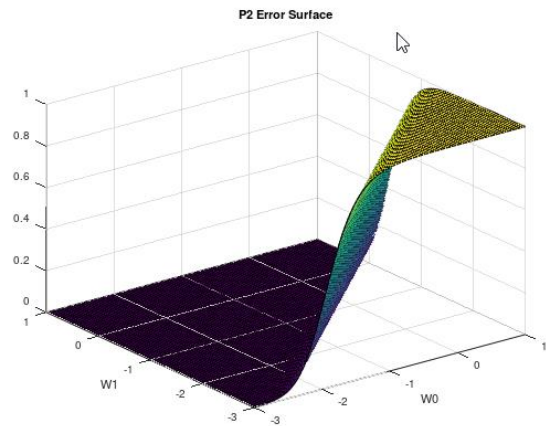
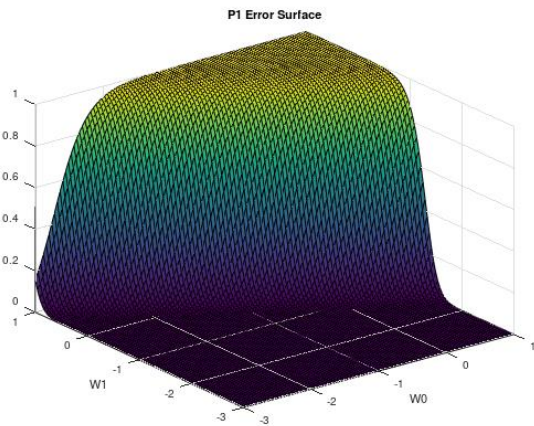
$${}_1\mathbf{W} = [-0.990791, -0.987997]$$

$${}_1\vec{B} = [4.97818]$$

We can render a *Decision Surface* for this network solution by defining a grid of points and evaluating each one with the solved network and plotting the result. The output of a 10 x 10 grid is shown below with the training points overlaid on the plot. As you can see, the resulting surface corresponds to the training point labels very well.



Now let's turn our attention to the Error Surface of each training pair. To make this tractable we will hold the bias vector constant and vary the values of the weight matrix. W_0 will range from -3 to 1 and W_1 will range from -3 to 1. The error value of the converged solution will be the point in the center of the graph because the W_0 and W_1 ranges were selected to put it in the center. The surface plots below show the Error Surface for each of the training points and the Average Error Surface. The Average Error Surface is the one where we want to find a minimum, preferably a global minimum.



The coordinate of the solution value is marked on the Average Error Surface plot with a red arrow. You can see that the solver correctly found a point in the region of the Average Error Surface that is low valued. The Average Error Surface possesses an offset rectangular region that is flat and low valued and any point in this region is an acceptable solution to this problem.

The fact that we could make the previous observation is not a luxury that we will have in any practical problem. That is the point of this exercise. The network is simple enough that we can plot the Error Surface and simply note that there is a deep rectangular “canyon” and see that an acceptable solution exists anywhere at the bottom of that canyon, because we can see that is where the surface attains a minimum. In even a slightly more complex network architecture we will have no sense of what the Error Surface looks like, if for no other reason than it exists in higher dimensions.

The takeaway is this, for every network an Error Surface does exist, and we are able to compute the gradient of the surface at any point. The most successful method of training a neural network to date is to begin by taking an educated guess at a reasonable starting point, which is not a guess at a solution, but merely an attempt to begin at a place on the error surface that is a reasonable value and reasonable slope (gradient). Reasonable being a value that is not so big or not so small as to be difficult to handle numerically. Then we compute the gradient at that point and go in the opposite direction, downhill, to the next point, then recompute the gradient and repeat the process until some stopping criteria is met. All without the benefit of knowing what the Error Surface looks like, or even being able to comprehend the dimensional space that it exists in, for that matter. Typical neural network weight space dimensions are on the order of 1000’s. The procedure for marching downhill is called gradient descent and the gradient is computed using a technique called Back-Propagation which will be covered next.

Gradient Descent

To date the most successfully applied method for training a neural network is the Gradient Descent method (4). The concept is simple. Initialize the parameters of the network with a guess, such as random values. This is a point in Weight Space. Then compute the gradient of the Error Surface at that point for each training pair. The average of the gradients of all training pairs in the training set yields the gradient of the Average Error Surface at the point in Weight Space. Knowing the gradient, use a Gradient Descent method to advance from the current point in Weight Space to one that is expected to yield a smaller Loss function value. But before a Gradient Descent method is applied the gradient value must be determined. The Back-Propagation method can be used to determine that value.

The Back-Propagation Method

The gradient computation of the Error Surface at a point can be broken down into a few formal steps that can be applied to a neural network of any depth (7). Breaking down these steps by working with the two-layer network example is enough to determine a pattern that can be generalized.

We begin with the two-layer network equations.

$${}_1\vec{Y} = {}_1\vec{f}({}_1\vec{Z}), \quad \text{where} \quad {}_1\vec{Z} = {}_1\mathbf{W} \cdot {}_1\vec{X} + {}_1\vec{B} \quad (1)$$

$${}_2\vec{X} = {}_1\vec{Y} \quad (2)$$

$${}_2\vec{Y} = {}_2\vec{f}({}_2\vec{Z}), \quad \text{where} \quad {}_2\vec{Z} = {}_2\mathbf{W} \cdot {}_2\vec{X} + {}_2\vec{B} \quad (3)$$

$${}^iE = L({}_2\vec{Y}, \dot{{}_1\vec{Y}}) \quad (4)$$

These equations really define just one single equation, and we can see that by substituting ${}_1\vec{Y}$ directly into Eq. (3) and then substituting that into Eq. (4) will yield:

$${}^iE = L\left(\left({}_2\vec{f}\left({}_2\mathbf{W}\left({}_1\vec{f}\left({}_1\mathbf{W}\left({}_1\vec{X} + {}_1\vec{B}\right)\right) + {}_2\vec{B}\right)\right), \dot{{}_1\vec{Y}}\right)$$

To facilitate understanding of the Back-Propagation method we will assign dimensions to the 2-layer network.

$$\text{Layer 1:} \quad a_1 = 2 \quad b_1 = 3$$

A 2-element Column vector is passed into the network (Layer 1) and a 3-element vector is returned from the first layer and passed to the second layer.

$$\text{Layer 2:} \quad a_2 = 3 \quad b_2 = 2$$

The second layer accepts a 3-element Column vector and returns a 2-element Column vector.

There is no need to give any meaning to the input and output of this network. Once the method development is complete it will be applicable to any dimension and network depth.

The Back-Propagation method begins with a forward pass through the network and the storage of supporting data. A forward pass-through Eq. (1) through Eq. (4) yields values for the intermediate variables: ${}_1\vec{Y}$, ${}_2\vec{Y}$, ${}_1\vec{Z}$, ${}_2\vec{Z}$, and ${}_2\vec{X}$.

The Weight Space of the example network is 17 dimensions. The gradient of E is also in \mathbb{R}^{17} and is given by:

$$\nabla E = \nabla_{{}_2\mathbf{W}} E + \nabla_{{}_2\vec{B}} E + \nabla_{{}_1\mathbf{W}} E + \nabla_{{}_1\vec{B}} E, \quad (5)$$

where each gradient on the right is a subspace of \mathbb{R}^{17} .

$$\nabla_{{}_1\mathbf{W}} E = \left(\overleftarrow{\frac{\partial L}{\partial {}_1\mathbf{W}}}\right) \vec{e} = \frac{\partial L}{\partial {}_1\mathbf{W}_1} \hat{e}_1 + \frac{\partial L}{\partial {}_1\mathbf{W}_2} \hat{e}_2 + \dots + \frac{\partial L}{\partial {}_1\mathbf{W}_6} \hat{e}_6 \quad (6)$$

$$\nabla_{{}_1\vec{B}} E = \left(\overleftarrow{\frac{\partial L}{\partial {}_1\vec{B}}}\right) \vec{e} = \frac{\partial L}{\partial {}_1\mathbf{B}_1} \hat{e}_7 + \frac{\partial L}{\partial {}_1\mathbf{B}_2} \hat{e}_8 + \frac{\partial L}{\partial {}_1\mathbf{B}_3} \hat{e}_9 \quad (7)$$

$$\nabla_{{}_2\mathbf{W}} E = \left(\overleftarrow{\frac{\partial L}{\partial {}_2\mathbf{W}}}\right) \vec{e} = \frac{\partial L}{\partial {}_2\mathbf{W}_1} \hat{e}_{10} + \frac{\partial L}{\partial {}_2\mathbf{W}_2} \hat{e}_{11} + \dots + \frac{\partial L}{\partial {}_2\mathbf{W}_6} \hat{e}_{15} \quad (8)$$

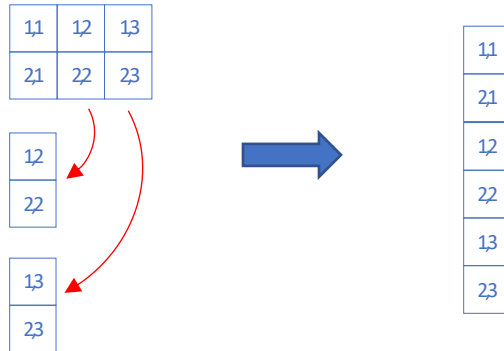
$$\nabla_{{}_2\vec{B}} E = \left(\overleftarrow{\frac{\partial L}{\partial {}_2\vec{B}}}\right) \vec{e} = \frac{\partial L}{\partial {}_2\mathbf{B}_1} \hat{e}_{16} + \frac{\partial L}{\partial {}_2\mathbf{B}_2} \hat{e}_{17} \quad (9)$$

Vectorizing a matrix

Take note of this notation: $\overleftarrow{{}_k\mathbf{W}}$, it is the matrix variable ${}_k\mathbf{W}$ with an arrow placed above it. The arrow notation over a matrix is used to indicate that the matrix is to be treated as a vector. I call this a vectorized matrix. A vectorized matrix means that each of its columns are placed one below the other to form a vector. We could just as well say that each row slides in front of the other, then transpose

that into a Column vector. In practice it is arbitrary, either way works as long as you remain consistent. Vectorizing by column was selected for this paper because the numerical implementation that accompanies it utilizes the Eigen matrix library, and the Eigen library vectorizes a matrix by columns.

Example vectorizing a 2 x 3 matrix:



Derivative Chain Rule

The Back-Propagation method is made possible by the derivative chain rule. Recall that given a function of f , that is a function of g , that is a function of x , expressed as $f(g(x))$, the derivative of f with respect to x is given by:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$$

Evaluation of the Network Error Surface Gradient

We will use the derivative chain-rule to evaluate Eq. (6) – Eq. (9). In the sections that follow we examine each term in the equations and determine how to evaluate them. After we know how to evaluate all of the individual terms a pattern will emerge allowing us to easily evaluate the Error Surface Gradients for a Neural Network of any depth.

Evaluate Eq. (8):

Using the chain rule:

$$\overleftarrow{\left(\frac{\partial L}{\partial_2 \mathbf{W}}\right)} = \overleftarrow{\left(\frac{\partial L}{\partial_2 \bar{y}}\right)} \frac{\partial_2 \bar{f}}{\partial_2 \bar{z}} \frac{\partial_2 \bar{z}}{\partial_2 \mathbf{W}} \quad (10)$$

Now we consider how to evaluate the terms on the right one-by-one.

Gradient of the Loss Function

Determine the Gradient vector of the Loss function with respect to its input, $\overleftarrow{\left(\frac{\partial L}{\partial_2 \bar{y}}\right)}$. The value of the input vector to the Loss function is one of the intermediate values that is known because its value was saved during the forward pass. To proceed, the partial derivative of the Loss function with respect to its independent variable is required. To make this a concrete example let's suppose that the Loss function is the L2 function we saw earlier. Its derivative is given by:

$$\frac{\partial L2(\vec{N}\vec{Y}, \vec{t}\vec{Y})}{\partial \vec{N}\vec{Y}} = \frac{1}{2} \vec{Z}, \text{ where } \vec{Z} = \vec{N}\vec{Y} - \vec{t}\vec{Y},$$

where in this example $N = 2$. Remember, $\vec{t}\vec{Y}$ is known, it is part of a training pair.

You can think of this as a “drop-in” or “modular” computation. During Back-Propagation input the stored value of the Loss function input vector into the gradient vector derivative of the Loss function and get a Row vector back. The size (number of elements) in the returned Row vector is always the same as the size of the input vector. The output from the L2 loss function is $\frac{1}{2} \vec{Z}^T$.

Activation Function Jacobian

Now consider the second term on the right of Eq. (10), $\frac{\partial_2 \vec{f}}{\partial_2 \vec{Z}}$, this derivative is in the form of a Jacobian derivative. It is the Jacobian of the Activation function. The result of this operation is always a square matrix because the size of the vector output by the Activation function is always equal to the size of the input vector. The value of the input vector to the Activation function is known from the forward pass. To proceed, the Jacobian of the Activation function with respect to its independent variable is required. Let’s suppose that the Activation function is the Sigmoid function we saw earlier. The Jacobian is given by:

$$\begin{bmatrix} Q_1(\vec{Z}) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & Q_N(\vec{Z}) \end{bmatrix}$$

where

$$Q_i(\vec{Z}) = S_i(\vec{Z}) (1 - S_i(\vec{Z})).$$

Here the derivative of the Sigmoid is given without explanation. If you are interested in the development details, there are many available sources online.

In this example $\frac{\partial_2 \vec{f}}{\partial_2 \vec{Z}}$ is given by:

$$\begin{bmatrix} Q_1(2\vec{Z}) & 0 \\ 0 & Q_2(2\vec{Z}) \end{bmatrix}.$$

Note that the matrix above is a diagonal matrix. Most Activation functions yield this result with a few notable exceptions.

Like the Loss function, you can think of the Activation function Jacobian as a “drop-in” or “modular” computation. During Back-Propagation input the stored value of the Activation function input vector into the Activation function Jacobian computation and get a matrix back. The matrix will always be square as noted above. Most commonly, the return will be a diagonal matrix, but not always. If you implement your own neural network solver, a bit of clever code can exploit the times when the return is a diagonal matrix.

The Weight Matrix Derivative

Now we will address the final term in Eq. (10), $\frac{\partial_2 \vec{Z}}{\partial_2 \vec{W}}$. This is an important term to understand how to evaluate properly as it will come up again in the development of a Convolutional Neural Network. Before we begin, there is some bookkeeping to do with the terms that have already been evaluated. Recall that the Loss function gradient vector is a Row Vector, and that the Jacobian of the Activation function is a square matrix. The number of elements in the Row vector (its “b” dimension) will always be equal to the number of rows of the matrix (its “a” dimension), making them suitable for multiplication to form an intermediate Row Vector we’ll call Delta (δ) and given by:

$$\vec{\delta} = \overline{\left(\frac{\partial L}{\partial_2 \vec{y}}\right)} \frac{\partial_2 \vec{f}}{\partial_2 \vec{Z}}. \quad (11)$$

Now we are ready to evaluate the final term in Eq. (10). Note that the derivative takes the form of a Jacobian derivative. Expanding the term by substitution for \vec{Z} defined in Eq. (3) yields the following:

$$\frac{\partial_2 \vec{Z}}{\partial_2 \vec{W}} = \frac{\partial}{\partial_2 \vec{W}} (\vec{W} \vec{X} + \vec{B})$$

Clearly, the derivative of the Bias term is zero and we are left with

$$\frac{\partial_2 \vec{Z}}{\partial_2 \vec{W}} = \frac{\partial}{\partial_2 \vec{W}} (\vec{W} \vec{X}).$$

To show the next steps in the evaluation I’ll use the dimensions of the example network and evaluate the matrix multiplication on the right leading to:

$$\frac{\partial}{\partial_2 \vec{W}} (\vec{W} \vec{X}) = \frac{\partial}{\partial_2 \vec{W}} \overline{\begin{bmatrix} W_{1,1} X_1 + W_{1,2} X_2 + W_{1,3} X_3 \\ W_{2,1} X_1 + W_{2,2} X_2 + W_{2,3} X_3 \end{bmatrix}} \quad (12)$$

We know that a gradient derivative operator such as $\frac{\partial}{\partial_2 \vec{W}}$ results in a Row vector, and recalling the form of a vectorized matrix, this gradient derivative is applied to each equation in the Column vector on the right in Eq. (12). To be perfectly clear about this, let’s expand the gradient derivative operator.

$$\frac{\partial}{\partial_2 \vec{W}} = \left\{ \frac{\partial}{\partial_2 W_{1,1}}, \frac{\partial}{\partial_2 W_{2,1}}, \frac{\partial}{\partial_2 W_{1,2}}, \frac{\partial}{\partial_2 W_{2,2}}, \frac{\partial}{\partial_2 W_{1,3}}, \frac{\partial}{\partial_2 W_{2,3}} \right\} \quad (13)$$

The resulting Jacobian matrix will be 2 x 6. A good way to visualize this computation is to think of each of the partial derivatives above as a column heading, while the values in the column are due to the application of the partial derivative in the heading. Evaluation of Eq. ?? results in the following matrix:

$$\begin{bmatrix} X_1 & 0 & X_2 & 0 & X_3 & 0 \\ 0 & X_1 & 0 & X_2 & 0 & X_3 \end{bmatrix}$$

Inserting this matrix and the Delta Row vector into Eq. (10) we can complete the evaluation of the term.

$$\begin{aligned} \overline{\left(\frac{\partial L}{\partial_2 \vec{W}}\right)} &= [\delta_1 \quad \delta_2] \begin{bmatrix} X_1 & 0 & X_2 & 0 & X_3 & 0 \\ 0 & X_1 & 0 & X_2 & 0 & X_3 \end{bmatrix} \\ &= [\delta_1 X_1 \quad \delta_2 X_1 \quad \delta_1 X_2 \quad \delta_2 X_2 \quad \delta_1 X_3 \quad \delta_2 X_3] \end{aligned} \quad (14)$$

At this point let us remind ourselves of why we made this computation. Each element in the result of Eq. (14) is the slope of the Error Surface with respect to a corresponding element of the ${}_2\mathbf{W}$ matrix. We will use this slope to modify the value of the corresponding element in an effort to find a minimum on the Error Surface. It makes no conceptual difference how we access the value of the slope, but in practice it is much more efficient to have all the slopes of Eq. (14) in the same form as ${}_2\mathbf{W}$. To this end, the result of Eq. (14) is reformed into a matrix. We use the “column headings” of Eq. (13) to know where to place the elements. The rearrangement leads to the following result:

$$\Delta_2\mathbf{W} = \begin{bmatrix} \delta_1 \ 2X_1 & \delta_1 \ 2X_2 & \delta_1 \ 2X_3 \\ \delta_2 \ 2X_1 & \delta_2 \ 2X_2 & \delta_2 \ 2X_3 \end{bmatrix} \quad (15)$$

There is a simple way to compute $\Delta_2\mathbf{W}$ directly, it is given by:

$$\Delta_2\mathbf{W} = ({}_2\vec{X} \ \vec{\delta})^T. \quad (16)$$

Note that there is no mathematical rhyme or reason to Eq. (16), we get there through inference by examining the hard-earned result of Eq. (15).

Evaluate Eq. (9):

$$\overleftarrow{\left(\frac{\partial L}{\partial_2\mathbf{B}}\right)} = \overleftarrow{\left(\frac{\partial L}{\partial_2\vec{Y}}\right)} \frac{\partial_2\vec{f}}{\partial_2\vec{Z}} \frac{\partial_2\vec{Z}}{\partial_2\mathbf{B}} \quad (17)$$

The first two terms on the right-hand side we have already seen, and we know how to evaluate them.

The B Derivative

$$\frac{\partial_2\vec{Z}}{\partial_2\mathbf{B}} = \frac{\partial}{\partial_2\mathbf{B}} ({}_2\mathbf{W} \ 2\vec{X} + {}_2\vec{B}) = \frac{\partial}{\partial_2\mathbf{B}} ({}_2\vec{B}) \quad (18)$$

The evaluation of this derivative is less cumbersome, but it is still a Jacobian derivative.

$$\frac{\partial}{\partial_2\mathbf{B}} = \left\{ \frac{\partial}{\partial_2B_1}, \frac{\partial}{\partial_2B_2} \right\}$$

The resulting Jacobian matrix will be 2 x 2. Evaluation of Eq. (18) results in the following matrix:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Inserting this matrix and the Delta Row vector into Eq. (9) we can complete the evaluation of the term.

$$\begin{aligned} \overleftarrow{\left(\frac{\partial L}{\partial_2\mathbf{B}}\right)} &= [\delta_1 \ \delta_2] \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ &= [\delta_1 \ \delta_2] \end{aligned}$$

Finally, the result can be reformed into a Column vector given by:

$$\Delta_2\mathbf{B} = \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} \quad (19)$$

Evaluate Eq. (6):

Now we will begin to evaluate the Error Surface Gradients due to Layer-1 and you will see why we started with the Layer-2 gradients.

$$\left(\frac{\partial L}{\partial \vec{\mathbf{W}}_1}\right) = \left(\frac{\partial L}{\partial \vec{\mathbf{Y}}}\right) \frac{\partial_2 \vec{f}}{\partial_2 \vec{Z}} \frac{\partial_2 \vec{Z}}{\partial_2 \vec{X}} \frac{\partial_1 \vec{f}}{\partial_1 \vec{Z}} \frac{\partial_1 \vec{Z}}{\partial_1 \vec{\mathbf{W}}_1} \quad (20)$$

On the right-hand side of Eq. (20) the third term has a form that we haven't seen before.

$$\frac{\partial_2 \vec{Z}}{\partial_2 \vec{X}} = \frac{\partial}{\partial_2 \vec{X}} (\vec{\mathbf{W}}_2 \vec{X} + \vec{\mathbf{B}}_2) = \frac{\partial}{\partial_2 \vec{X}} (\vec{\mathbf{W}}_2 \vec{X})$$

Expanding the matrix multiplication as we did in Eq. (12) yields:

$$\frac{\partial}{\partial_2 \vec{X}} (\vec{\mathbf{W}}_2 \vec{X}) = \frac{\partial}{\partial_2 \vec{X}} \begin{bmatrix} \vec{\mathbf{W}}_{2,1} X_1 + \vec{\mathbf{W}}_{2,2} X_2 + \vec{\mathbf{W}}_{2,3} X_3 \\ \vec{\mathbf{W}}_{2,1} X_1 + \vec{\mathbf{W}}_{2,2} X_2 + \vec{\mathbf{W}}_{2,3} X_3 \end{bmatrix} \quad (21)$$

The gradient derivative operator,

$$\frac{\partial}{\partial_2 \vec{X}} = \left\{ \frac{\partial}{\partial_2 X_1}, \frac{\partial}{\partial_2 X_2}, \frac{\partial}{\partial_2 X_3} \right\},$$

applied to the Column vector of Eq. (21) yields the following matrix:

$$\begin{bmatrix} \vec{\mathbf{W}}_{2,1} & \vec{\mathbf{W}}_{2,2} & \vec{\mathbf{W}}_{2,3} \\ \vec{\mathbf{W}}_{2,1} & \vec{\mathbf{W}}_{2,2} & \vec{\mathbf{W}}_{2,3} \end{bmatrix},$$

which upon inspection reveals that

$$\frac{\partial}{\partial_2 \vec{X}} (\vec{\mathbf{W}}_2 \vec{X}) = \vec{\mathbf{W}}_2.$$

Now we know how to evaluate all the terms in Eq. (20) and could simply plug them to obtain a result. Doing so reveals a pattern that can be generalized and will be explained shortly.

Evaluate Eq. (7):

$$\left(\frac{\partial L}{\partial \vec{\mathbf{B}}_1}\right) = \left(\frac{\partial L}{\partial \vec{\mathbf{Y}}}\right) \frac{\partial_2 \vec{f}}{\partial_2 \vec{Z}} \frac{\partial_2 \vec{Z}}{\partial_2 \vec{X}} \frac{\partial_1 \vec{f}}{\partial_1 \vec{Z}} \frac{\partial_1 \vec{Z}}{\partial_1 \vec{\mathbf{B}}_1} \quad (22)$$

We have seen all of the forms of the terms in this equation before. Can you see the repeating pattern in the terms?

The Back-Propagation algorithm - Putting it all together

Using the chain-rule to evaluate the Error Surface gradients due to Layer-1 reveals a pattern in the terms. The Back-Propagation algorithm starts at the Loss term of the network and works back to Layer-1. I'll introduce a Row vector called the Inter Layer (IL) Row vector to clarify the process. Take note of how Eq.(6) – Eq. (9) are being evaluated. The $\vec{\mathbf{IL}}$ Row vector is propagating the product of terms from the Loss Layer to Layer-2 (defined by Eq. (8) and Eq. (9)) and finally to Layer-1 (defined by Eq. (6) and Eq. (7)).

Starting with the Loss layer initialize the Inter Layer Row vector:

$${}_3\overleftarrow{L} = \overleftarrow{\left(\frac{\partial L}{\partial {}_2\vec{Y}}\right)}$$

The Inter Layer Row vector is passed to Layer-2 and the following computations are made:

$${}_2\tilde{\delta} = {}_3\overleftarrow{L} \left[\frac{\partial {}_2\vec{f}}{\partial {}_2\vec{Z}} \right], \quad (\text{The square brackets } [] \text{ denote a matrix and are used as a reminder})$$

where ${}_2\tilde{\delta}$ is a Row vector of the same dimension as ${}_3\overleftarrow{L}$.

$${}_2\overleftarrow{L} = {}_2\tilde{\delta} \left[\frac{\partial {}_2\vec{Z}}{\partial {}_2\vec{X}} \right] = {}_2\tilde{\delta} {}_2\mathbf{W}$$

$$\Delta_2\mathbf{W} = ({}_2\vec{X} {}_2\tilde{\delta})^T$$

$$\Delta_2\vec{B} = ({}_2\tilde{\delta})^T$$

The Inter Layer Row vector is passed to Layer-1, and we begin to see the Back-Propagation pattern. The following computations are made based on ${}_2\overleftarrow{L}$:

$${}_1\tilde{\delta} = {}_2\overleftarrow{L} \left[\frac{\partial {}_1\vec{f}}{\partial {}_1\vec{Z}} \right],$$

where ${}_1\tilde{\delta}$ is a Row vector of the same dimension as ${}_2\overleftarrow{L}$.

$${}_1\overleftarrow{L} = {}_1\tilde{\delta} \left[\frac{\partial {}_1\vec{Z}}{\partial {}_1\vec{X}} \right] = {}_1\tilde{\delta} {}_1\mathbf{W}$$

Note that ${}_1\overleftarrow{L}$ is never needed, as Back-Propagation ends at the top layer.

$$\Delta_1\mathbf{W} = ({}_1\vec{X} {}_1\tilde{\delta})^T$$

$$\Delta_1\vec{B} = ({}_1\tilde{\delta})^T$$

Generalizing, the Back-Propagation algorithm is initialized with the Loss function gradient Row vector, then the Inter Layer vector is propagated up through the network from Layer-N to Layer-1.

Network Forward Pass Algorithm

Given a training pair $\{\vec{X}_1, \hat{Y}\}$, use \vec{X}_1 as input into Layer-1. Feed the output of each layer into the input of the next layer. The final output is the network result which would be used when the network is applied. During training the final output is passed into the Loss function along with its label \hat{Y} . Internal states are stored during this pass.

- 1: **for** $n = 1$ to N **do**
 - 2: $\vec{Z}_n = \mathbf{W}_n \vec{X}_n + \vec{B}_n$
 - 3: $\vec{X}_{n+1} = f(\vec{Z}_n)$ ▷ Apply the Activation function f to vector Z .
 - 4: **end**
 - 5: $error = Loss(\vec{X}_{N+1}, \hat{Y})$
-

Back-Propagation Algorithm

Initialize the Inter-layer Row vector with the Loss function gradient. The Inter-layer Row vector is used to compute the layer delta Row vector, then the Inter-layer Row vector can be updated for use in the next higher layer where N is the number of layers in the network. Note that there is not a need to maintain each Inter-layer Row vector. One Inter-layer Row vector can be used and updated.

- 1: $\overleftarrow{L} = \left(\frac{\partial L}{\partial \vec{Y}_N} \right)$ ▷ Initialize the Inter-Layer row vector with the Loss gradient
 - 2: **for** $n = N$ to 1 **do**
 - 3: $\vec{\delta}_n = \overleftarrow{L} \left[\frac{\partial f_n}{\partial \vec{Z}_n} \right]$ ▷ Inter-Layer row vector * Activation Jacobian
 - 4: $\overleftarrow{L} = \vec{\delta}_n \left[\frac{\partial \vec{Z}_n}{\partial \vec{X}_n} \right] = \vec{\delta}_n \mathbf{W}_n$ ▷ Update the Inter-Layer row vector for use by the next higher layer
 - 5: $\Delta_n \mathbf{W} = (\vec{X}_n \vec{\delta}_n)^T$
 - 6: $\Delta_n \vec{B} = (\vec{\delta}_n)^T$
 - 7: **end**
-

Compute the Average Error Surface

The gradient of the Average Error Surface is computed by using each training pair one-by-one for a forward pass through the network, which establishes the values of the intermediate variables, then immediately making a Back-Propagation pass back through the network, all the while maintaining a few ancillary variables used to compute the averages of the delta variables $\overline{\Delta_k \mathbf{W}}$ and $\overline{\Delta_k \vec{B}}$. The averages may be accomplished by keeping a summation and dividing by the number of training pairs or rolling N into the step size of the update algorithm. I prefer to keep a running average and will outline that simple algorithm below.

Average Algorithm

Given a batch of P training pairs. Retrieve each training pair from the batch one-by-one and pass the input vector X into the network, then use the result along with the training label Y to compute the Loss function gradient for training pair p . Use the resultant gradient to build the average Loss gradient for each of N layers.

```
1: for  $p = 1$  to  $P$  do
2:    $(\vec{X}, \vec{Y}) = \text{TrainingPair}(p)$            ▷ Get the input vector and label for training pair  $p$ .
3:    $(\Delta_k \mathbf{W}, \Delta_k \vec{B}) = \text{ApplyNetwork}(\vec{X}, \vec{Y})$    ▷ Use the training pair for one training pass.
4:    $a = 1 / p$ 
5:    $b = 1 - a$ 
6:   for  $k = 1$  to  $N$  do
7:      $\overline{\Delta_k \mathbf{W}} = a \Delta_k \mathbf{W} + b \overline{\Delta_k \mathbf{W}}$ 
8:      $\overline{\Delta_k \vec{B}} = a \Delta_k \vec{B} + b \overline{\Delta_k \vec{B}}$ 
9:   end
10: end
```

The Update Algorithm

There is a lot written about the how to choose an update step. The size of the step can be tailored to each parameter of Weight Space and can be modified as the iterative process converges. The simplest approach is to use a fixed size step and that is what will be used in the section that follows. Once you grasp the concepts discussed in this paper you will be ready to understand and apply more complex step size selection algorithms.

Let Eta (η) represent the descent step size. The update process is simple, advance the value of the current point in Weight Space in the downhill direction. Remember, the gradient of the Error Surface at the current point in Weight Space tells us the direction of steepest **assent**. We want to advance in the downhill direction, so we go in the opposite (mathematically, negative) direction. We must still decide how far to go in the specified direction and that is given by η . Given η , the network solution is advanced by applying the following algorithm.

Update Algorithm

Having computed the Weight Space gradients of each network layer, iterate over each of N layers, and update the Weight matrix and Bias vector. The step size η is provided.

```
1: for  $k = 1$  to  $N$  do
2:    ${}_k \mathbf{W} = {}_k \mathbf{W} - \eta \overline{\Delta_k \mathbf{W}}$ 
3:    ${}_k \vec{B} = {}_k \vec{B} - \eta \overline{\Delta_k \vec{B}}$ 
4: end
```

If η is set too large the solution will not converge, rather the parameters will get larger and larger until a math overflow occurs. I have found for networks of a few layers starting η at $1E-5$ works well. Observe the network error for stability and check that it is getting smaller and smaller. If it is, multiply η by 10 and try again. If the solution remains stable, it should converge faster.

Gradient Descent Methods

There are three common ways the Error Surface gradient is used to minimize the Loss function, the Batch method, the Mini-Batch method, and Stochastic descent.

The Batch Method

The Batch method utilizes the entire training set to compute the Average Error surface gradient. Then the gradient, along with a step value, is used to determine the next point in Weight Space. This is the method that was outlined in this paper in the Average Algorithm above.

Stochastic Gradient Descent

We noted that each training pair yields an Error Surface and that a gradient is computed for each of these surfaces in our effort to compute the Average Error Surface gradient. The Stochastic method does not bother to compute the Average Error Surface gradient. It moves the network solution along based on each individual training pair. It updates the layer's Weight matrices and Bias vectors based on each Error Surface. This method iterates towards a solution in a zig-zag motion, but it has been shown to converge to a solution the fastest for many network topologies. It is not difficult to implement. For each training pair, skip the average section, or make an average of one, then move the solution forward with the Update step.

The Mini-Batch Method

The mini-batch method is simply a combination of the two methods mentioned above. The mini-batch method computes the average gradient of a batch of training pairs and takes an iterative step based on each mini-batch, progressing from batch to batch until the entire training set is utilized. The mini-batch method itself has a few different implementations.

The simplest implementation is to work through the training set in batches and when you run out of training pairs go to the top of the list and run through it again.

It has been suggested that a better method is to randomly sample from the training set. Whatever the mini-batch size is, randomly sample from the training set to fill the mini-batch, compute the mini-batch Average Error Surface gradient, advance the solution, then repeat the process.

Augmented Layer Equation

There is a more compact way to write the layer equation that is useful for coding a Neural Network solver. Consider the Neural Network Layer equation written with the augmented variable dimensions that we saw above.

$$(a \times 1) = (a \times 1) [(a \times b) * (b \times 1) + (a \times 1)]$$
$$Y = f [\mathbf{W} * X + B]$$

The technique focuses on how the linear part of the equation, the input to the Activation function, is written.

$$(a \times b) * (b \times 1) + (a \times 1)$$
$$\mathbf{W} * X + B$$

Define a matrix **G** and call it the Augmented Weight matrix and define a vector H called the Augmented input vector. The Augmented layer equation is given by:

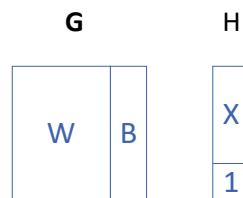
$$(a \times c) * (c \times 1) = (a \times b) * (b \times 1) + (a \times 1)$$

$$\mathbf{G} * \mathbf{H} = \mathbf{W} * \mathbf{X} + \mathbf{B},$$

where $c = b + 1$.

Note that matrix **G** has one more column than **W**; and vector H has one more row than X; and the Bias vector is not on the left side of the equation. To make the above equation true:

- Set the left side block of **G** equal to **W**, and place the Bias vector B into the additional column of matrix **G**.
- Set the top b rows of H equal to X and set the bottom row of H equal to one.



Both the forward pass and back-propagation steps work without modification on the Augmented layer equation. The Augmented layer equation operates on the weight matrix **W** and Bias vector B together.

The SimpleNet code base uses the Augmented layer formulation.

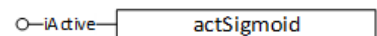
Implementation and Verification

The mathematics and algorithms discussed in this paper are implemented in the SimpleNet code library. SimpleNet is a C++ codebase designed to be easily understandable and extensible. It utilizes the popular Eigen matrix library which provides abstractions for Row vector, Column vector, and Matrix, so the code very closely follows the descriptions found in this paper. The entire library is implemented in one header file making it easy to add to a C++ project. It is object oriented and uses an Interface based approach to compose the parts of a Neural Network. A Neural Network layer is represented by the Layer class. The things that make one layer different from another are the input and output dimensions, and the type of Activation function. The iActive Interface, defined by a pure virtual class, provides the method signatures that must be implemented by a concrete Activation class. An instance of an Activation class is used by the Layer class to apply the Activation function to a Column vector, and to provide the Activation function Jacobian matrix.

```

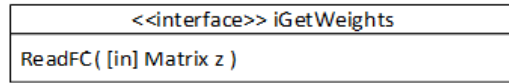
<<interface>> iActive
Resize( [in] size : int )
Eval( [in/out] z : ColVector ) : ColVector
Jacobian( [in] z : ColVector ) : Matrix
    
```

Some Activation function implementations:

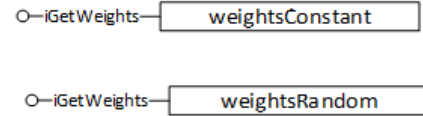


The iGetWeights Interface provides the method signatures that must be implemented to create an Initialization class. Initialization classes are used by the Layer class to initialize the Layer's Weight matrix and Bias vector. The Layer class is instantiated by providing input and output dimensions for the

network layer, and by passing it an iActive interface to an instance of an Activation object, and by passing it an iGetWeights interface to an Initialization object.

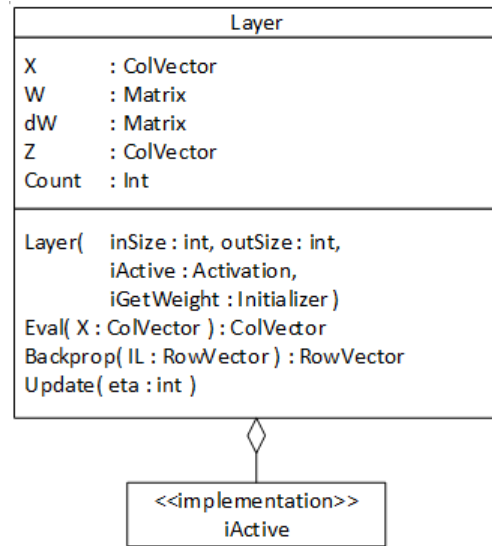


The Loss layer is represented by classes that implement the iLossLayer interface.



A multi-layer network is created by defining a bi-directional container, such as a doubly linked list, and pushing instances of appropriately sized Layer objects onto it. The Loss layer is implemented as a separate object. To iterate towards a solution the Layers are initialized, and the layer list is navigated from front to back, first by passing the input vector into the first (top) layer's Eval method. The resulting Column vector is then passed to the Eval method of the next layer and so on until the end of the list is reached. Then the final layer output Column vector is passed to the Loss layer object's Eval method along with the training vector. The result of this method call is a scalar value indicating the measure of error.

Back-Propagation is accomplished by immediately calling the Loss layer's LossGradient method which returns a Row vector. Then the layer list is navigated from the last to the first layer. The Loss layer Row vector is passed to the last layer's Backprop method which returns a Row vector. This is the inter-layer Row vector. That Row vector is passed to the next layer's Backprop method and so on until the top layer is arrived at. That process updates the delta matrix of each layer. An update step is taken by navigating through the layer list again and calling each layer's Update method.



See Appendix 3 for details about the SimpleNet library.

Algorithm and Code Verification

A simple test is devised to test the method and the implementation of the algorithm described in this paper. The goal of the Back-Propagation method is to compute the gradient of the Loss function with respect to the Weight Space parameters. We can compute and store the gradient of Weight Space at a point in Weight Space, then we can compute the gradient using a finite difference approximation and compare the results. A second order approximation was used, given by:

$$\frac{\partial L}{\partial W_{i,j}} = \frac{(L(W_{i,j}^+) - L(W_{i,j}^-))}{2\epsilon},$$

where

$$W_{i,j}^+ = W_{i,j} + \epsilon$$

$$W_{i,j}^- = W_{i,j} - \epsilon$$

The finite difference approximation to the gradient of L is computed by perturbing matrix \mathbf{W} one element at a time. Each element is perturbed by $(+\epsilon)$ and then the value of L is computed, then it is

perturbed by $(-\epsilon)$ and the value of L is computed again, then the two perturbed values are used to compute the derivative of L with respect to element $W_{i,j}$.

To check the accuracy of this algorithm a multi-layer network was created and initialized, but only the Weight matrix of the top level was checked. If the implementation or algorithm in the lower layers is wrong or inaccurate that problem will be made evident in the top layer.

Massive Subtractive Cancellation Errors

To understand the results of the verification test we must understand what can be expected of the results. The computation was performed in double precision on a 64-bit PC and compiled with Microsoft Visual Studio to C++ 17 standard. MS C++ implements *double* data types according to the IEEE-754 standard. The IEEE-754 standard will provide from 15 to 17 significant digits (8).

Finite difference computation accuracy is a function of the step size (ϵ) . The centered difference approximation is $O(\epsilon^2)$. However, the accuracy of the approximation is also a function of subtractive cancellation (ξ) and given by ξ/ϵ . If ϵ is too small the subtractive cancellation error will exceed the approximation error (9) (10).

Results

The Binary Classification problem we saw earlier was used to test the algorithm. That problem had a two-element input vector and a 1 element output vector. To test the Back-Propagation through hidden layers the following network was used.

Layer	Input size	Output size	Activation function
1	2	5	Sigmoid
2	5	9	Sigmoid
3	9	11	Sigmoid
4	11	1	Sigmoid

The L2 Loss was used for the Loss layer.

The Back-Propagation method should provide the true value of the derivative. The absolute value of the difference between the Back-Propagation computed delta matrix (dW) and the finite difference delta matrix was computed and the maximum of the difference was recorded. The following table shows the results relative to various values of ϵ .

ϵ	Max Error
1.0E-3	5.5033E-08
1.0E-4	5.50505e-10
1.0E-5	1.21861e-11
1.0E-6	4.00341e-11

1.0E-7	2.92281e-10
1.0E-8	3.2825e-09

The values in the table above show the expected U-shaped result due to subtractive cancellation. For optimally sized ϵ a result better than $O(\epsilon^2)$ is achieved giving a good indication that both the method and the implementation are correct.

Summary and Future work

A vector-based description of a Feed Forward Neural Network was presented and used to derive the Back-Propagation method. Particular attention was paid to forming the partial derivatives used for Back-Propagation, most notably derivatives with respect to the Weight matrix, $\frac{\partial_k \vec{Z}}{\partial_k \mathbf{W}}$. This attention to detail will be necessary when applied to the development of a Convolutional Neural Network, which I plan to write about in another paper. Algorithms for encoding the mathematics into a computer program were presented and the SimpleNet implementation discussed, including a method to test the accuracy of the mathematics and the implementation.

References

1. **Thomas and Finney.** *Calculus and Analytic Geometry.* s.l. : Assison Wesley, 1981.
2. **Baker, A. J.** *Finite Element Computational Fluid Mechanics.* s.l. : Taylor & Francis, 1983. ISBN1-56032-245-4.
3. **Ong, Jeremy.** *C++ Neural Network in a Weekend.*
4. *A Theoretical Framework for Back-Propagation.* **Cun, Yann le.** Pittsburg, PA : Proceedings of the 1988 Connectionist Models Summer School, 1988.
5. **Dolhansky, Brian.** *Artificial Neural Networks.* s.l. : Brian Dolhansky, 2014.
6. **Zabarauskas, Manfred.** *Backpropagation Tutorial.* 2011.
7. **Rumelhart, D. E., Hinton, G. E. and Williams, R. J.** *Learning internal representations by error propagation.* San Diego : California Univ San Diego La Jolla Inst for Cognitive Science, 1985. rumelhart1985learning.
8. *IEEE 754.* s.l. : Wikipedia. https://en.wikipedia.org/wiki/IEEE_754.
9. **Baydin, Atilim, et al.** *Automatic Differentiation in Machine Learning: a Survey.* s.l. : arXiv, 2018. 1502.05767v4.
10. **Conte, Samuel D. and de Boor, Carl.** *Elementary Numerical Analysis.* s.l. : McGraw-Hill, 1981. ISBN 0-07-012447-7.

Appendix 1 – Common Activation Functions

An Activation function always returns a vector of the same dimension as the input vector.

Linear Activation Function

The Linear activation function does not alter the input data, it returns it unaltered.

Function:

The base function is given by:

$$f(x) = x .$$

The vector form is given by:

$$\vec{f}(\vec{Z}) = \vec{Z} ,$$

where it is understood that the vector \vec{Z} is returned.

Jacobian:

The Jacobian of the Linear Activation function is the Identity matrix with rows and columns equal to the rows of \vec{Z} .

Sigmoid Activation Function

Function:

The base function is given by:

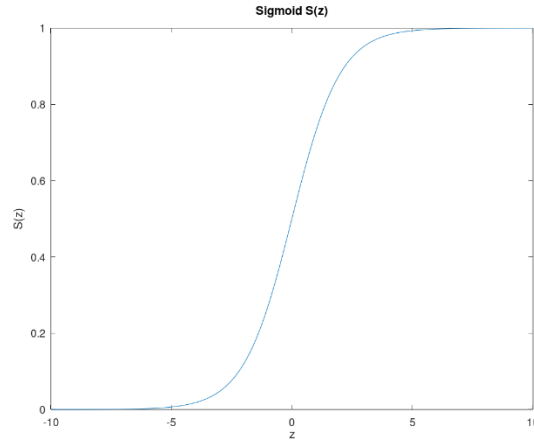
$$f(x) = \frac{1}{1 + e^{-x}}$$

The vector form is given by:

$$\vec{S}(\vec{Z}) = \frac{1}{1 + e^{-\vec{Z}}} ,$$

where it is understood that the left side of the equation is applied to \vec{Z} element by element.

The Sigmoid function approaches its minimum and maximum asymptotically.



Jacobian:

$$Q_i(\vec{Z}) = S_i(\vec{Z}) (1 - S_i(\vec{Z})),$$

where $i = 1$ to N , where N is the number of elements (rows) in \vec{Z} .

The Jacobian matrix is a diagonal matrix with \vec{Q} on the main diagonal. In Linear Algebra terms, it is given by $\mathbf{I} \vec{Q}$, where \mathbf{I} is the Identity matrix.

ReLU Activation Function

ReLU stands for rectified linear unit.

Function:

$$rlu(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

$$\vec{f}(\vec{Z}) = rlu(\vec{Z})$$

Jacobian:

The Jacobian matrix is a diagonal matrix with rows and columns equal to the number of rows of \vec{Z} . The elements on the diagonal are either zero or one and are given by:

$$J_{ii} = \begin{cases} 1, & Z_i \geq 0 \\ 0, & Z_i < 0 \end{cases}$$

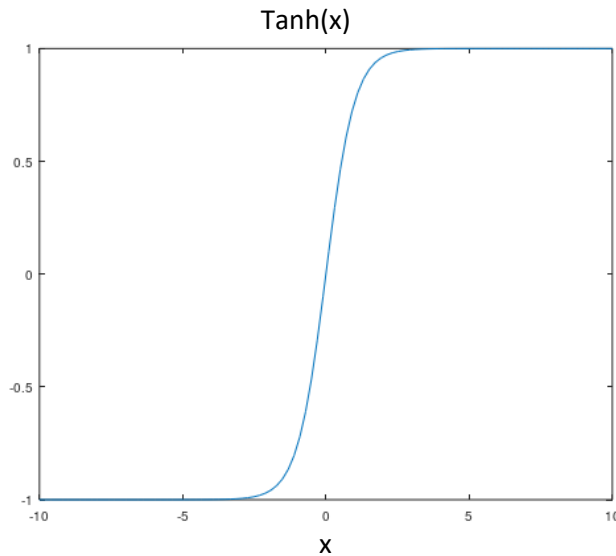
Tanh Activation Function

Function:

I think most software languages include the Hyperbolic Tangent but to get the value another way it is this:

$$\tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

$$\vec{f}(\vec{Z}) = \tanh(\vec{Z})$$



Jacobian:

The Jacobian matrix is a diagonal matrix with rows and columns equal to the number of rows of \vec{Z} . The elements on the diagonal are given by:

$$J_{ii} = 1 - \text{Tanh}^2(Z_i)$$

Softmax Activation Function

The Softmax Activation function is used for multi-label classification problems. The output of a multi-label classification neural network is a vector where each position in the vector is a label. The binary classification problem we saw earlier could be recast as a multi-label classification network. That network would output a two-element vector. Element one could be the label for blue and element two the label for yellow. The values of a multi-label model output vector represent the probability of the label associated with that position being the right classification, but the values are not constrained and though the element values represent probability, the input vector is not a proper probability distribution. Softmax transforms the input vector into a probability distribution.

Function:

$$\vec{f}(\vec{Z}) = \frac{\vec{Z}}{\sum_i^N e^{Z_i}}$$

Written another way,

$$f_j(\vec{Z}) = \frac{Z_j}{\sum_i^N e^{Z_i}}$$

Jacobian:

The Jacobian of the Softmax Activation is a full matrix.

$$J_{i,j} = \begin{cases} f_i(\vec{Z})(1 - f_i(\vec{Z})), & i = j \\ -f_i(\vec{Z})f_j(\vec{Z}), & i \neq j \end{cases}$$

Appendix 2 – Common Loss Functions

L2 Loss Function

The L2 Loss function is the L2 norm, a.k.a. the Euclidean norm. Sometimes there is a $1/2$ in front of the Dot product so that the derivative does not have a 2 in front of it. I think it is 6 of one, a half dozen of the other.

Function:

$$f(\vec{N\bar{Y}}, \vec{i\check{Y}}) = \vec{Q}^T \vec{Q}, \text{ where } \vec{Q} = \vec{N\bar{Y}} - \vec{i\check{Y}}.$$

Gradient Vector Derivative:

$$\overline{\left(\frac{\partial f}{\partial \vec{N\bar{Y}}}\right)} = 2 \vec{Q}^T$$

Cross Entropy Loss Function

The Cross Entropy Loss function is designed to compare two probability distribution functions. It does not work with element values greater than 1. It should always be preceded by the SoftMax Activation function.

Function:

$$f(\vec{N\bar{Y}}, \vec{i\check{Y}}) = - \sum_j^m i\check{Y}_j \log(\max(N\bar{Y}_j, \epsilon))$$

Where m is the number of rows of the input vector and ϵ is a small number to prevent overflow in the case that $N\bar{Y}_j$ is zero. ϵ could be set to machine precision but if you want to limit the maximum gradient value then you might consider setting it an order of magnitude higher than machine precision.

Gradient Vector Derivative:

$$\overline{\left(\frac{\partial f}{\partial \vec{N\bar{Y}}}\right)}_j = \begin{cases} i\check{Y}_j / N\bar{Y}_j, & N\bar{Y}_j > \epsilon \\ i\check{Y}_j / \epsilon, & N\bar{Y}_j \leq \epsilon \end{cases}$$

Appendix 3 – SimpleNet

In this section we will discuss how to use the SimpleNet library to build neural network projects. The library was developed on a Windows 10 computer running Visual Studio Community 2019 with the C++ compiler targeting the C++ 14 standard. The library has also been compiled under the C++ 17 standard. There has been no attempt to insure cross-platform compatibility, however, the library only minimally accesses the file system to store and read weight matrices. Adding SimpleNet to a project is well... pretty simple, so to avoid complications no “make” file is provided, I’ll go over what you need to do to use SimpleNet in your projects.

Project Setup

The SimpleNet library is implemented in a header file called “Layer.h”. To use the library include it in your project code. “Layer.h” includes the Eigen header file. The Eigen library is also implemented as a header file, so no other files need to be included in your projects.

Add Include Paths to your C++ Project

To use the SimpleNet library you must download and place the Eigen library somewhere on your file system. In your C++ project add an include path to the Eigen folder. If you intend to build more than one project, you should put the SimpleNet library in its own folder. If you do that you should add a include path to the SimpleNet folder as well. To do this using Visual Studio, right-click on your project and click on Properties in the popup menu. In the dialog left panel, click on C/C++, then General, then in the right panel add the paths to the “Additional Include Directories”.

Getting the Code

You can also go to the Blog post that hosts this paper to download the SimpleNet library (A single header file), a project example (the one used for the Binary Classification Example, and Eigen version 3.37. Here’s the URL:

http://www.raberfamily.com/scottblog/neural_net_1.htm

For more details about Eigen and to get the latest version here’s the Eigen URL:

<https://eigen.tuxfamily.org>

Basic use

The SimpleNet library uses the Eigen library for matrix math. The SimpleNet library defines these three important object types:

Matrix	This is a double precision Eigen Matrix class. Creation examples: <pre>Matrix m(100,100); Matrix m; m.resize(10,10); Matrix n; n = m;</pre>
ColVector	This is a double precision Eigen Vector class. At its base it is a one column matrix.

	<pre>ColVector x(2); x(0)=1.0; x(1)=2.0; ColVector b; b = x; ColVector y; y.resize(10);</pre>
RowVector	<p>This is a double precision Eigen RowVector class. At its base it is a one row matrix.</p> <pre>RowVector x(2); x(0)=1.0; x(1)=2.0; RowVector b; b = x; RowVector y; y.resize(10);</pre>

The above object types are used throughout the SimpleNet library and are needed in your program to interact with the library.

The Layer Class

Constructor

```
Layer(int input_size, int output_size, iActive* _pActive, shared_ptr<iGetWeights> _pInit )
```

Parameter	Description						
input_size	This parameter specifies the number of elements in the input vector to the layer. If the layer is the top layer, then this parameter should be set to the size of the model input vector.						
output_size	This parameter specifies the number of elements in the the output vector of the layer. If this is the last layer, then this size should be set to the output size of the model.						
_pActive	<p>This is a pointer to an iActive interface, which is just a C++ class pointer. Use new to create a pointer to an implementation class. Don't hold on to this pointer, there is no fancy reference counting, the Layer class expects to be the sole owner of the interface and will call delete on it during Layer class destruction.</p> <p>Use one of the following Activation classes or implement your own.</p> <table border="1" style="margin-left: 20px;"> <tr><td>actSigmoid</td></tr> <tr><td>actTanh</td></tr> <tr><td>actLinear</td></tr> <tr><td>actReLU</td></tr> <tr><td>actLeakyReLU</td></tr> <tr><td>actSoftMax</td></tr> </table>	actSigmoid	actTanh	actLinear	actReLU	actLeakyReLU	actSoftMax
actSigmoid							
actTanh							
actLinear							
actReLU							
actLeakyReLU							
actSoftMax							

_pInit	<p>This is a smart pointer to an iGetWeights interface. The Layer class will use this pointer in the constructor then discard it.</p> <p>Use one of the following classes or implement your own.</p> <table border="1"> <thead> <tr> <th>Class</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>IWeightsToConstants</td> <td>Initialize network weight and bias vector to constants.</td> </tr> <tr> <td>IWeightsToRandom</td> <td>Initialize network weights to random values in a range.</td> </tr> <tr> <td>IWeightsToNormDist</td> <td>Initialize network weights to random values utilizing the Xavier or Kanning methods to calculate variance.</td> </tr> <tr> <td>IOWeightsBinaryFile</td> <td>Initialize network to stored weights. This implementation class also implements the iPutWeights interface and should be used to create the weight files.</td> </tr> </tbody> </table>	Class	Description	IWeightsToConstants	Initialize network weight and bias vector to constants.	IWeightsToRandom	Initialize network weights to random values in a range.	IWeightsToNormDist	Initialize network weights to random values utilizing the Xavier or Kanning methods to calculate variance.	IOWeightsBinaryFile	Initialize network to stored weights. This implementation class also implements the iPutWeights interface and should be used to create the weight files.
Class	Description										
IWeightsToConstants	Initialize network weight and bias vector to constants.										
IWeightsToRandom	Initialize network weights to random values in a range.										
IWeightsToNormDist	Initialize network weights to random values utilizing the Xavier or Kanning methods to calculate variance.										
IOWeightsBinaryFile	Initialize network to stored weights. This implementation class also implements the iPutWeights interface and should be used to create the weight files.										

Example 1

This example creates a pointer to a Layer class. This instance takes a 2-element input vector and outputs a 5-element vector. It uses the Sigmoid activation functions and is randomly initialized. The weight class instantiation is a bit cumbersome.

```
Layer* lyr = new Layer(2, 5,
    new actSigmoid(),make_shared<IWeightsToNormDist>(IWeightsToNormDist::Xavier,1) );
```

Example 2

This example shows how to use the Layer class. The library does support a graph (connection) model, you supply that in your code. Fully Connected networks are easy to put together because one layer calls the next. In the example below a two-layer network is setup. The network takes a 2-element input vector and outputs a 1-element vector.

```
typedef vector< shared_ptr<Layer> > layer_list;
layer_list LayerList;
// a1 specifies the weights (nodes) in a hidden layer.
int a1 = 9;
LayerList.push_back(
    make_shared<Layer>(2, a1, new actSigmoid(),
        make_shared<IWeightsToNormDist>(IWeightsToNormDist::Xavier,1))
);

LayerList.push_back(
    make_shared<Layer>(a1, 1, new actSigmoid(),
        make_shared<IWeightsToNormDist>(IWeightsToNormDist::Xavier,1))
);
```

Eval

```
ColVector Eval(const ColVector& _x)
```

Eval takes a ColVector as input and outputs a ColVector not necessarily of the same size. It is used for the forward pass through the network.

Example:

Given initialized ColVector X for input to the network, and using the LayerList STL Vector from the example above, apply the network. On completion X will be resized and contain the result of the network. During training the output of the network should be passed to the Loss object along with the training label (vector).

```
for (layer_list::iterator l = LayerList.begin();
     l != LayerList.end();
     l++) {
    X = (*l)->Eval(X);
}
loss->Eval(X, Y);
```

There is a fancy way to do this using modern C++ that is much more readable.

```
for (const auto& lit : LayerList) {
    X = lit->Eval(X);
}
loss->Eval(X, Y);
```

BackProp

```
RowVector BackProp(const RowVector& child_grad, bool want_layer_grad = true )
```

The method BackProp is used for the backward pass through the network. After a forward pass, call a Loss function to initialize the Back-Propagation inter-layer Row vector, then pass that into the BackProp method of the last (bottom) Layer and iterate through the network from the bottom to the top.

Example:

During network training a code block similar to that below should be run directly after a forward pass through the network.

```
RowVector g = loss->LossGradient();
for (layer_list::reverse_iterator riter = LayerList.rbegin();
     riter != LayerList.rend();
     riter++) {
    g = (*riter)->BackProp(g);
}
```

Update

```
void Update(double eta)
```

Call Update to apply the delta matrix, $\overline{\Delta_k \mathbf{W}}$, to the current weight matrix and reset the average counter to zero. You can call Update after processing a full training batch, a mini-batch, or after each training label (stochastic descent).

Example:

```
double eta = 0.75;
for (const auto& lit : LayerList) {
    lit->Update(eta);
}
```

Save

```
void Save(shared_ptr<iPutWeights> _pOut)
```

Use the Save method to save the layer weights to file so that they can be loaded later and used or further updated. The current methods of saving the layer weights results in a separate file for each layer. Keep these files together to load them back into the network. This is admittedly and purposefully an unsophisticated method of saving the network parameters. This simple method also lends itself well to outputting the weight matrices to CSV file for display or reading by other applications.

Use one of the storage class or build your own by implementing the iPutWeights interface.

Class	Description
IOWeightsBinaryFile	Use this class to write the layer matrices to file for later recovery.
OWeightsCSVFile	Use this class to write the layer matrices to CSV file.

Example 1:

A unique file name needs to be generated for each layer. One way to do this is to enumerate the file name. In the following example each layer is save to both binary and CSV format. The string variable *path* has been initialized to the file path where the files should be written.

```
int l = 0;
for (const auto& lit : LayerList) {
    l++;
    string layer = to_string(l);
    lit->Save(make_shared<OWeightsCSVFile>(path, "parallel." + layer ));
    lit->Save(make_shared<IOWeightsBinaryFile>(path, "parallel." + layer ));
}
```

Example 2:

This example demonstrates how to restore a network weight from previously stored weights. In the example below the network construction is setup with an inline conditional statement. If *restore* is set to *true* then the IOWeightsBinaryFile class will be used to restore the layer weights from file.

```
std::string model_name = "FC32";
```

```

LayerList.clear();
bool restore = true;

// FC Layer 1 -----
// Type: FC Layer
int size_in = MNISTReader::DIM;
int size_out = 32;

int l = 1; // Layer counter
LayerList.push_back(make_shared<Layer>(size_in, size_out, new actReLU(size_out),
    restore ? dynamic_pointer_cast<iGetWeights>(
        make_shared<IOWeightsBinaryFile>(path, model_name + "." + to_string(l))
    ) :
    dynamic_pointer_cast<iGetWeights>(
        make_shared<IWeightsToNormDist>(IWeightsToNormDist::Kanning, 1)))
    );
l++;

// FC Layer 2 -----
// Type: FC Layer
size_in = size_out;
size_out = 10;
LayerList.push_back(make_shared<Layer>(size_in, size_out, new actSoftMax(size_out),
    restore ? dynamic_pointer_cast<iGetWeights>(
        make_shared<IOWeightsBinaryFile>(path, model_name + "." + to_string(l))
    ) :
    dynamic_pointer_cast<iGetWeights>(
        make_shared<IWeightsToNormDist>(IWeightsToNormDist::Xavier, 1)))
    );
l++;

```