

Math Behind Digital Handwritten Recognition

James Huang

Introduction

Once I went to a bank, where I was presented a pad to write down the amount of money I intended to withdraw; I casually scribbled a number on the pad, and it instantaneously recognized my handwriting and converted to the correct numbers in digital format. This has had me wondering ever since: how exactly did the computer manage to recognize the intended number's digits from my handwriting?

Research Question: What is the math behind digital handwritten recognition?

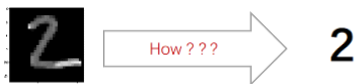


Fig 1. Essence of the problem.

The problem in essence is, given an image grid of certain size (say, $28 \times 28 = 784$ pixels), how to detect the digit number (0-9) in the grid. A pixel is the basic square-shaped unit in an computer image and has a value of 0-1, denoting the brightness ratio. After some initial attempts with traditional programming methods with if/else and for loops, I came to realize that they are not suitable to solve this. Somehow identifying digits is almost impossible to describe how to do though it is so easy for our brains to do. More investigation led me to Neural Nets (NN), computing systems inspired by the biological neural networks that constitute human brains. I was able to code up a python program that uses Tensorflow, the de-facto Artificial Intelligence (AI) framework, and employs a neural net to train and recognize handwritten digits in the 60,000 sample MNIST dataset (LeCun, Yann.) with 99.79% accuracy.

I further explored the NN's inner workings to understand how it really works, particularly the math principles and processes behind. In the following, I will use my exploration as an example to go over the math behind how a NN works in three parts: the basics, how it learns and my experimental verifications.

Part One: Neural Nets (NNs)

1.1 Artificial Neural Nodes

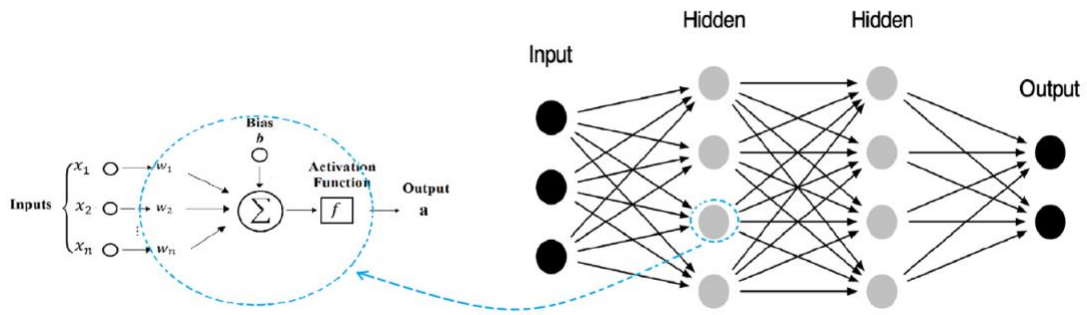


Fig 2. A neural node and a NN that has one input layer, two hidden layers and an output layer.

Designed to mimic human neural networks, artificial neural nets (NN) consist of connected neural nodes. Figure 2 illustrates a neural node from a NN. It takes inputs from vector $X (x_1, x_2, x_3, \dots x_n)$, has a node specific bias value (b) and maintains a weight value vector \vec{w} storing w_i for each input node (x_i). The bias (b) decides the threshold value of the node's activation, while the weight value decides the level of importance (weight) of an input. A neural node does two functions:

1. **Summation** (z) of the input values (x_i) with respective weights (w_i) and bias (b):

$$z = x_1 \cdot w_1 + x_2 \cdot w_2 + \dots + x_n \cdot w_n + b \quad \dots (1)$$

Or formally if we use the vectors X, \vec{w} , for a neural node, we have

$$z = X \cdot \vec{w} + b \quad \dots (2)$$

where \cdot is the **dot product** operand for vectors.

2. **Activation** of the input signal using function $f(z)$, adding non-linearity like the selective activation mechanism of a human neuron, with a as the output.

$$a = f(z) \quad \dots (3)$$

In a NN, data flows from one layer of nodes to another layer of nodes. Suppose the layer takes inputs from a vector X of size m , it then uses a vector B storing each node's biases, $b_1, b_2, b_3, \dots, b_n$, and a matrix W of dimensions $[m, n]$, to store all nodes' weight values for all of the input nodes. The summation results will be stored in a vector Z (storing $z_1, z_2, z_3, \dots, z_n$). Like equation (2) which is for one node, we have the following for a layer of n nodes:

$$Z = X \cdot W + B \quad \dots (4)$$

Similarly, if we use vector A to store all the activation results (a_1, a_2, a_3, \dots):

$$A = f(Z) \quad \dots (5)$$

Table 1 lists my NN's activation functions.

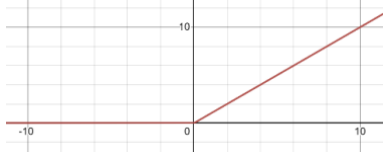

Name	$f(z)$	Graph	Range
Relu	$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$		$(0, \infty)$
Softmax	$f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^N e^{z_k}}$		$(0,1)$

Table 1. Summary of some common activation functions.

Relu has output 0 if the input is less than 0, and raw output otherwise. Sharing a high similarity to the biological neural activation, it is the most popular activation function for deep NNs. The Softmax Activation Function a vector of real numbers as inputs and normalizes them into a probability distribution proportional to the exponentials of the input numbers, which is very useful in multi-classification.

z scores	Softmax Activation	Probability Distribution
$\begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix}$	$f(z_i) = \frac{e^{z_i}}{\sum_{k=1}^N e^{z_k}}$	$\begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}$

Table 2. Softmax activation function converts scores to probability distributions adding up to 1.0.

1.2 Artificial Neural Nets

My python program defines a 4-layered NN for handwriting digits recognition. It has an input layer (I) that takes 784 pixel values (28x28) of an image as inputs, first passes them to the first hidden layer (H1) containing 128 nodes with a Relu activation function, then passes to the second hidden layer (H2) containing 64 nodes with another Relu activation function, then flows the data to the output layer (O) which has 10 nodes and uses Softmax for activation, finally computes and outputs the predicted probabilities of the input image belonging to digits 0-9. Figure 3 illustrates the NN.

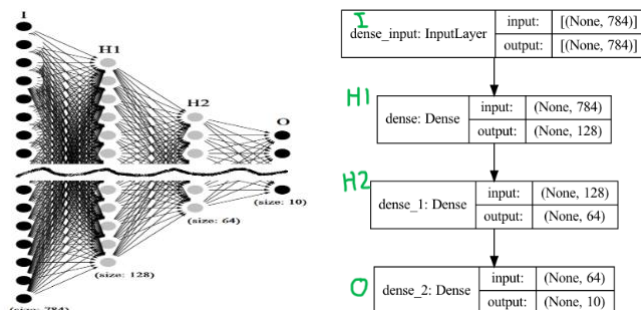


Figure 3. My NN's topology and hierarchy, with the curve representing the undrawn parts.

Part Two: How NNs Train and Learn

For my NN to recognize/predict with good accuracy, it needs to be “trained” by running through lots of training data examples with labels and “learn”. Through research study I learned that a NN's behavior is determined by its nodes' weights and biases. The weights represent the strength of edge connections between each node in layer $(l - 1)$ and each node in layer (l) . And each bias is an indication of whether its

node tends to be activated or not. A NN achieves training and learning through:

1. (For a new NN) initialize each node's weights with random values, biases with 0s
2. **feed forward** the training example data through all layers of the NN
3. calculate the **loss** using the outputs against the labelled results (target vector y)
4. use a mechanism called **gradient descent & back propagation** to adjust each node's weights and bias values to minimize the loss and improve precision
5. keep doing steps 2) to 4) for many epochs (1 epoch = 1 full dataset pass) until good precision and minimal loss are achieved, now the NN has a "golden" set of weights and biases.

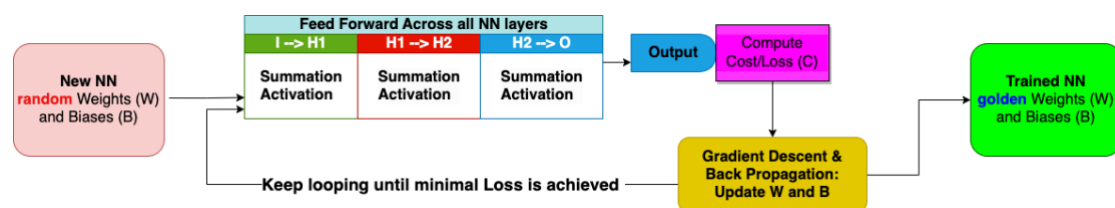


Figure 4. Workflow of how my NN trains and learns.

In the following I will start with a handwritten digit's image, investigate how this input data feeds forward across all the layers of my NN, how losses are computed, and how the NN learns through back-propagation.

2.1 The Feed Forward and Prediction

The 28 by 28 pixel image (array) of a handwritten digit, a 2 dimensional array, is first flattened into a one dimensional vector X of length $28 \times 28 = 784$, denoted as X [1,784], also called the Input layer (I).

2.1.1 I->H1: From Input to First Layer (H1)

X then is passed on to the first hidden layer (H1), which has 128 nodes. The weight matrix for H1 is of dimensions [784, 128], denoted as $W^{(1)}$, meaning the weight matrix for layer H1. For consistency, the same naming convention is used for all

entities in this essay. For example, denotations of $W^{(l)}$, $B^{(l)}$, $l \in [1,3]$, represent the weight matrix and bias vector of layer l , respectively, i.e., If we look at the third node of H1 layer and want to know the weight value of the second input value x_2 , the corresponding one is $w_{2,3}^{(1)}$. According to equation (4), we have the summation equation as follows,

$$Z^{(1)} = X \cdot W^{(1)} + B^{(1)} \quad \dots (6)$$

The underlying computation is illustrated below.

$$\begin{array}{c}
 \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ \vdots \\ x_{784} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} & w_{1,4}^{(1)} & \cdots & w_{1,128}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} & w_{2,4}^{(1)} & \cdots & w_{2,128}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} & w_{3,4}^{(1)} & \cdots & w_{3,128}^{(1)} \\ w_{4,1}^{(1)} & w_{4,2}^{(1)} & w_{4,3}^{(1)} & w_{4,4}^{(1)} & \cdots & w_{4,128}^{(1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{784,1}^{(1)} & w_{784,2}^{(1)} & w_{784,3}^{(1)} & w_{784,4}^{(1)} & \cdots & w_{784,128}^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ b_4^{(1)} \\ \vdots \\ b_{128}^{(1)} \end{bmatrix} \xrightarrow{\text{yields}} \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \\ z_4^{(1)} \\ \vdots \\ z_{128}^{(1)} \end{bmatrix} \\
 X[1,784] \quad \quad \quad W^{(1)}[784,128] \quad \quad \quad B^{(1)}[1,128] \quad \quad Z^{(1)}[1,128]
 \end{array}$$

If we look at how all input values of X go through the third node of H1 layer, the summation for this node will be:

$$z_3^{(1)} = \sum_{i=1}^{784} x_i \cdot w_{i,3}^{(1)} + b_3^{(1)} \quad \dots (7)$$

The summation for H1 layer completes after each of the 784 input values has run through each of the 128 nodes of layer H1. After summation, at each node the activation ensues. H1 layer's activation function is Relu. I chose Relu because it filters out all the negative summation values and outputs the positive ones linearly without information loss (Sharma, Sagar). According to equation (5), the activation $Z^{(1)}: A^{(1)}$ works as:

$$A^{(1)} = f(Z^{(1)})$$

For each node, the activation is as follows:

$$a_i^{(1)} = f(z_i^{(1)}) = \begin{cases} 0, & z_i^{(1)} < 0 \\ z_i^{(1)}, & z_i^{(1)} \geq 0 \end{cases} \text{ where } i \in \{1, \dots, 128\} \quad \dots (7a)$$

The activation happens inside the same layer on each node, and the output $A^{(1)}$ vector contains the 128 output values and will serve as the input vector for next layer H2.

$$\begin{array}{ccc} \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \\ z_4^{(1)} \\ \vdots \\ z_{128}^{(1)} \end{bmatrix} & \xrightarrow{a_i^{(1)} = f(z_i^{(1)})} & \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ a_4^{(1)} \\ \vdots \\ a_{128}^{(1)} \end{bmatrix} \\ Z^{(1)} [1, 128] & & A^{(1)} [1, 128] \end{array}$$

2.1.2 H1->H2: From Layer H1 to Layer H2

The second layer H2 has 64 nodes and takes the 128 elements of $A^{(1)}$ as inputs. So, its weight matrix $W^{(2)}$ is of dimensions $[128, 64]$, and $B^{(2)}$ is a vector of 64 bias values.

The summation vector $Z^{(2)}$ is of size 64. Similarly, we have

$$Z^{(2)} = A^{(1)} \cdot W^{(2)} + B^{(2)} \quad \dots (8)$$

$$\text{So, generally,} \quad Z^{(l)} = A^{(l-1)} \cdot W^{(l)} + B^{(l)} \quad \dots (8a)$$

where l denotes the layer number, $l \in [1, L]$, A^0 is the input vector X

I again chose Relu as the activation function for H2 for the same good reasons above.

After the activation $Z^{(2)}$: $A^{(2)}$, $A^{(2)}$ will be a vector of 64 values which then feeds forward to the third layer as input.

2.1.3 H2->O: From Layer H2 to Layer O -- Prediction

The last layer O has 10 nodes, corresponding to the 10 digits (0,1,2,...,9). It takes the 64 elements of $A^{(2)}$ as inputs. So $W^{(3)}$ is of dimensions $[64, 10]$, and $B^{(3)}$ is a vector of 10 biases. The summation $Z^{(3)}$ is a vector of size 10. Hence,

$$Z^{(3)} = A^{(2)} \cdot W^{(3)} + B^{(3)} \quad \dots (9)$$

The underlying computation is as follows.

$$\begin{matrix}
 \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \\ a_4^{(2)} \\ \vdots \\ a_{64}^{(2)} \end{bmatrix} & \cdot & \begin{bmatrix} w_{1,1}^{(3)} & w_{1,2}^{(3)} & w_{1,3}^{(3)} & w_{1,4}^{(3)} & \cdots & w_{1,10}^{(3)} \\ w_{2,1}^{(3)} & w_{2,2}^{(3)} & w_{2,3}^{(3)} & w_{2,4}^{(3)} & \cdots & w_{2,10}^{(3)} \\ w_{3,1}^{(3)} & w_{3,2}^{(3)} & w_{3,3}^{(3)} & w_{3,4}^{(3)} & \cdots & w_{3,10}^{(3)} \\ w_{4,1}^{(3)} & w_{4,2}^{(3)} & w_{4,3}^{(3)} & w_{4,4}^{(3)} & \cdots & w_{4,10}^{(3)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{64,1}^{(3)} & w_{64,2}^{(3)} & w_{64,3}^{(3)} & w_{64,4}^{(3)} & \cdots & w_{64,10}^{(3)} \end{bmatrix} & + & \begin{bmatrix} b_1^{(3)} \\ b_2^{(3)} \\ b_3^{(3)} \\ b_4^{(3)} \\ \vdots \\ b_{10}^{(3)} \end{bmatrix} & \xrightarrow{\text{yields}} & \begin{bmatrix} z_1^{(3)} \\ z_2^{(3)} \\ z_3^{(3)} \\ z_4^{(3)} \\ \vdots \\ z_{10}^{(3)} \end{bmatrix} \\
 A^{(2)} [1,64] & & W^{(3)} [64,10] & & B^{(3)} [1,10] & & Z^{(3)} [1,10]
 \end{matrix}$$

Accordingly, we have

$$A^{(3)} = f(Z^{(3)}) \quad \dots (10)$$

The activation function for layer O is chosen as Softmax, because it is most suited for multi-classification (Sharma, Sagar) as my NN intends to classify an image into one of 10 classifications (0, 1, 2..., 9). For each node in layer O, the activation step uses Softmax function to normalize and produce a prediction probability:

$$a_i^{(3)} = f(z_i^{(3)}) = \frac{e^{z_i^{(3)}}}{\sum_{k=1}^{10} e^{z_k^{(3)}}} \quad \dots (11)$$

After the activation $Z^{(3)}: A^{(3)}$, $A^{(3)}$ will be a vector of 10 prediction probabilities that add up to 1.0, each matching the corresponding digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). For instance, if $a_2^{(3)}$ has a value of **0.95**, that means my NN predicts that the input handwritten digit has a **95%** probability to be digit **1**. The following shows the mapping between Z, A and corresponding classifications:

$$\begin{matrix}
 \begin{bmatrix} z_1^{(3)} \\ \color{red}{z_2^{(3)}} \\ z_3^{(3)} \\ z_4^{(3)} \\ \vdots \\ z_{10}^{(3)} \end{bmatrix} & \xrightarrow{a_i^{(3)} = f(z_i^{(3)})} & \begin{bmatrix} a_1^{(3)} = 0.001 \\ \color{red}{a_2^{(3)} = 0.950} \\ a_3^{(3)} = 0.002 \\ a_4^{(3)} = 0.005 \\ \vdots \\ a_{10}^{(3)} = 0.012 \end{bmatrix} & \dots & \begin{bmatrix} 0 \\ \color{red}{1} \\ 2 \\ 3 \\ 4 \\ 5 \\ \vdots \\ 9 \end{bmatrix} \\
 Z^{(3)} [1,10] & & A^{(3)} [1,10] & & \text{digits}
 \end{matrix}$$

2.2 Cost/Loss Function and Gradient Descent

2.2.1 Cost/Loss Function

For each training data example, along with the input X vector there is also a target vector y of size 10 truth-labelling with which of 10 digits (0-9) the input image matches. For example, $y_3 = 1.000$ means the input image is digit **2**. To measure the performance of my NN, I need a cost/loss function C to calculate the error between the output prediction vector $A^{(3)}$ and target vector y .

$$\begin{array}{ccc}
 \begin{array}{c} a_1^{(3)} \\ a_2^{(3)} \\ a_3^{(3)} \\ a_4^{(3)} \\ \vdots \\ a_{10}^{(3)} \\ A^{(3)} [1,10] \end{array} & \xleftrightarrow{C} & \begin{array}{c} y_1 = 0.000 \\ y_2 = 0.000 \\ \mathbf{y_3 = 1.000} \\ y_4 = 0.000 \\ \vdots \\ y_{10} = 0.000 \\ \text{target } y [1,10] \end{array} \quad \dots \quad \begin{array}{c} 0 \\ 1 \\ \mathbf{2} \\ 3 \\ 4 \\ 5 \\ \vdots \\ 9 \\ \text{digits} \end{array}
 \end{array}$$

When designing the NN, there are two popular loss functions I considered: Mean Square Error (MSE) and Categorical Cross-Entropy (CCE).

For MSE, the total cost is computed as: (Mazur, Matt)

$$C = \sum_{i=1}^{10} (y_i - a_i^{(3)})^2 \quad \dots (12)$$

where y_i is the truth label (taking a value 0 or 1) from the target vector y .

For Categorical Cross-Entropy, the total cost is: (Loech, Kiprono Elijah)

$$C = -\sum_{i=1}^{10} (y_i \log_e a_i^{(3)}) \quad \dots (13)$$

where y_i is the truth label (taking a value 0 or 1) from the target vector y .

I chose CCE over MSE, because CCE excels in multi-classification tasks where the decision boundary is large while MSE is strong in regression cases and does not punish misclassifications enough (Kurbiel). In addition, CCE works with probability

values output by Softmax better. The function C is for computing the loss of one training example and reflects how much the NN needs to improve to be more precise. If the NN trains on a batch of N examples, then the average cost is calculated as, (N is the size of the batch)

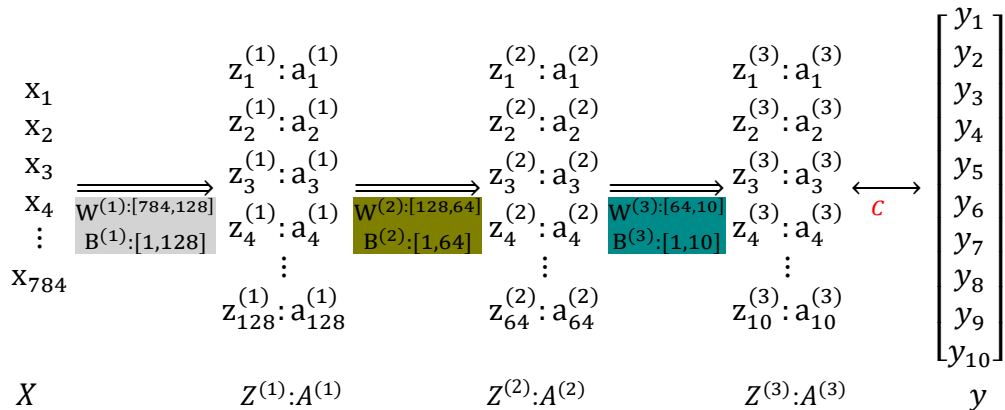
$$\bar{C} = -\frac{1}{N} \sum_{i=1}^{10} (y_i \log_e a_i^{(3)}) \quad \dots (14)$$

2.2.2 Gradient Descent

Now that my NN has completed one full pass of feed forward of the training example and determined the corresponding loss C , it should strive to minimize C next. C is a function of $A^{(3)}$, which depends on $Z^{(3)}$, which further depends on all the weights and biases across the layers, so I need to find the set of weights and biases that minimizes the cost function – this is essentially what “learning” is about. Suppose the cost function takes just one weight as variable, $C(w)$ arrives at **minima** when

$$\frac{dC}{dw} = 0$$

My NN has a total of $784 \times 128 + 128 \times 64 + 64 \times 10 = 109,184$ weights and $128 + 64 + 10 = 202$ biases, so the cost function has 109,386 variables! How to find its minima?



My research led to an approach called “**gradient descent**” (Patrikar, Sushant). It goes like this: we start at a random input and figure out which direction to step to yield a

lower function value; specifically, we take the **derivative/slope** of the function at the point, if the **slope is negative**, shift a “learning step” to the right. If the **slope is positive**, shift a “learning step” to the left. At the new position we check the slope and do the above repeatedly until the **derivative is 0** or nearly 0, meaning we are at the minima or near it. Figure 5 illustrates the gradient descent process for one variable and multi-variable functions.

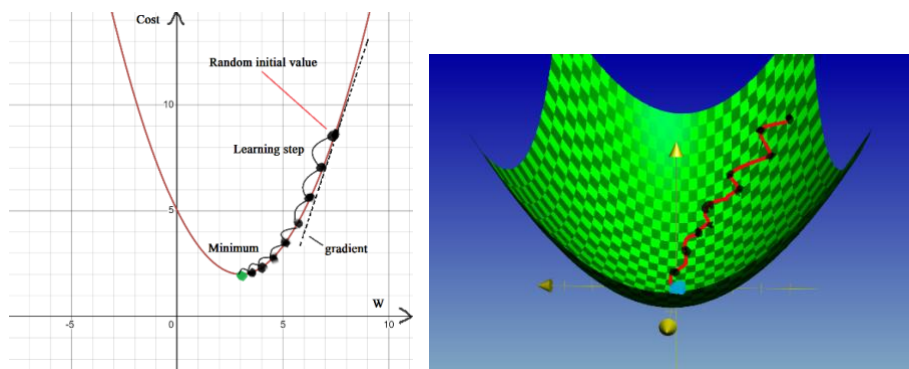


Figure 5. Illustration of gradient descent for 1 variable (left) and multi-variable (right) functions.

In multivariable calculus, a gradient vector ∇C containing **partial derivatives** is used to represent the direction of the steepest ascent; the negative of the gradient, $-\nabla C$, gives the downhill direction. In the case of my NN, ∇C will have 109,386 **partial derivatives** of the cost function with respect to all the different weights and biases.

$$\nabla C = \left[\frac{\partial C}{\partial w_{1,1}^{(1)}}, \frac{\partial C}{\partial w_{2,1}^{(1)}}, \dots, \frac{\partial C}{\partial w_{784,128}^{(1)}}, \frac{\partial C}{\partial w_{2,1}^{(1)}}, \dots, \frac{\partial C}{\partial b_{128}^{(1)}}, \dots, \frac{\partial C}{\partial w_{64,10}^{(3)}}, \dots, \frac{\partial C}{\partial b_{10}^{(3)}} \right]$$

So we have this formula for weight updates: (Mazur, Matt)

$$W_{new} = W - \eta \nabla_{\theta} C(W) \quad \dots (15)$$

where

- η is the learning rate, which can be a configurable constant like 0.01 or an adaptive value that changes bases on the current and previous weight values
- W is the current weight matrix, W_{new} is the new weight matrix

- $\nabla_{\theta} C(W)$ is the **partial derivative** of the cost function w.r.t weights W , it is a vector containing $\frac{\partial C}{\partial W}$ entries

Similarly,

$$B_{new} = B - \eta \nabla_{\theta} C(B) \quad \dots (16)$$

The entries of the gradient vector ∇C are needed for updating the weights and biases, therefore, how to compute ∇C is at the heart of the training and learning of a NN.

2.3 Back Propagation

In this section, I will investigate how my NN calculates the entries for gradient vector ∇C , which also takes the form of $\nabla_{\theta} C(W)$ w.r.t W and $\nabla_{\theta} C(B)$ w.r.t B :

$$\nabla_{\theta} C(W) = \frac{\partial C}{\partial W}, \quad \nabla_{\theta} C(B) = \frac{\partial C}{\partial B}$$

So far, my input image data has traversed across all the layers all the way to layer O, so I have these values $X, W^{(1)}, B^{(1)}, Z^{(1)}, A^{(1)}, W^{(2)}, B^{(2)}, Z^{(2)}, A^{(2)}, W^{(3)}, B^{(3)}, Z^{(3)}, A^{(3)}$.

With equation (13) and target vector y , I could also calculate the cost/lost C . Now I want to calculate the gradient for weights of the third layer H2, $W^{(3)}$. According to the chain rule of calculus, we have

$$\frac{\partial C}{\partial W^{(3)}} = \frac{\partial Z^{(3)}}{\partial W^{(3)}} \frac{\partial A^{(3)}}{\partial Z^{(3)}} \frac{\partial C}{\partial A^{(3)}} \quad \dots (17)$$

Looking at the chaining, I have come to the realization that a nudge in $W^{(3)}$ could have rippling effects: it first leads to changes to $Z^{(3)}$, further leading to changes in $A^{(3)}$, and finally changing C . Gradient computation takes the route of **Back**

Propagation, from C all the way back to the interested weight or bias. To generalize, to calculate the gradient for weights of layer l , the chain rule works:

$$\frac{\partial C}{\partial W^{(l)}} = \frac{\partial Z^{(l)}}{\partial W^{(l)}} \frac{\partial A^{(l)}}{\partial Z^{(l)}} \frac{\partial C}{\partial A^{(l)}} \quad \dots (18)$$

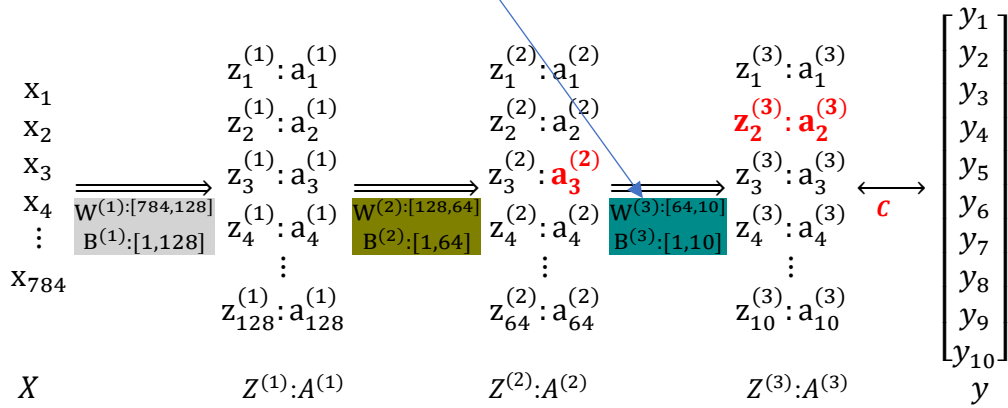
where $l \in [1,3]$

The detailed theoretical deductions of the computing formulas for $\frac{\partial z^{(l)}}{\partial w^{(l)}}$, $\frac{\partial A^{(l)}}{\partial z^{(l)}}$ and $\frac{\partial C}{\partial A^{(l)}}$

(equations 18a to 21) are in APPENDIX-B. I will use the following examples to show how to compute the gradients for each layer.

First, I want to compute a gradient in layer O, i.e., the edge connecting the 3rd node of

layer H1 and 2nd node of layer H2, $\frac{\partial C}{\partial w_{3,2}^{(3)}}$ in ∇C ,



According to the chain rule, $\frac{\partial C}{\partial w_{3,2}^{(3)}} = \frac{\partial z_2^{(3)}}{\partial w_{3,2}^{(3)}} \frac{\partial a_2^{(3)}}{\partial z_2^{(3)}} \frac{\partial C}{\partial a_2^{(3)}}$

Given that (equation 8a)

$$\because z_2^{(3)} = a_3^{(2)} \cdot w_{3,2}^{(3)} + b_2^{(3)}, \quad \because \frac{\partial C}{\partial w_{3,2}^{(3)}} = a_3^{(2)}$$

From equation (21) in Appendix-B,

$$\because \frac{\partial a_2^{(3)}}{\partial z_2^{(3)}} \frac{\partial C}{\partial a_2^{(3)}} = \frac{\partial C}{\partial z_2^{(3)}} = a_2^{(3)} - y_2, \quad \because \frac{\partial C}{\partial w_{3,2}^{(3)}} = a_3^{(2)} \cdot (a_2^{(3)} - y_2)$$

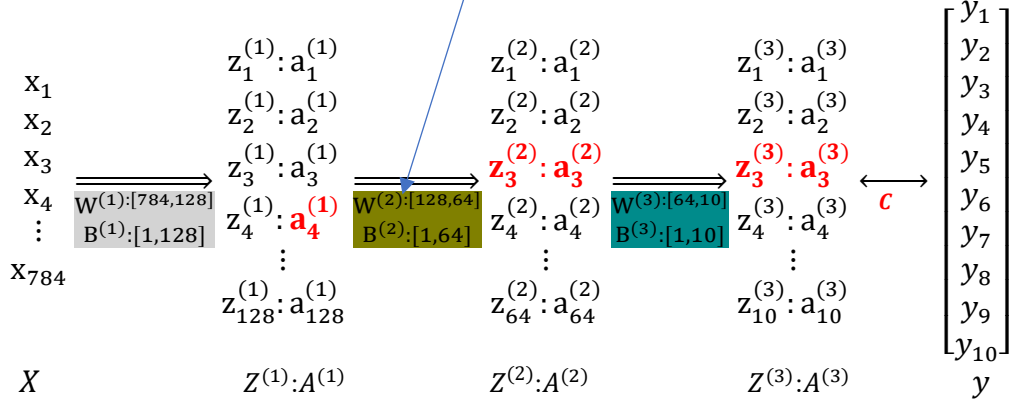
It is easy to generalize for any weight gradient $w_{i,j}^{(3)}$ in layer O:

$$\frac{\partial C}{\partial w_{i,j}^{(3)}} = a_i^{(2)} \cdot (a_j^{(3)} - y_j) \quad \dots (22a)$$

Similarly, to calculate the j -th bias gradient in layer O:

$$\frac{\partial C}{\partial b_j^{(3)}} = \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial C}{\partial a_j^{(3)}} = (a_j^{(3)} - y_j) \quad \dots (22b)$$

Next, I need to compute a gradient in layer H2, i.e., the edge connecting the fourth node of layer H1 and third node of layer H2, $\frac{\partial C}{\partial w_{4,3}^{(2)}}$ in ∇C .



Again, according to the chain rule of calculus,

$$\frac{\partial C}{\partial w_{4,3}^{(2)}} = \frac{\partial z_3^{(2)}}{\partial w_{4,3}^{(2)}} \frac{\partial a_3^{(2)}}{\partial z_3^{(2)}} \frac{\partial z_3^{(3)}}{\partial a_3^{(2)}} \frac{\partial a_3^{(3)}}{\partial z_3^{(3)}} \frac{\partial C}{\partial a_3^{(3)}}$$

$$\because z_3^{(2)} = a_4^{(1)} w_{4,3}^{(2)} + b_3^{(2)}, \therefore \frac{\partial z_3^{(2)}}{\partial w_{4,3}^{(2)}} = a_4^{(1)}$$

$$\because a_3^{(2)} = f(z_3^{(2)}), \therefore \frac{\partial a_3^{(2)}}{\partial z_3^{(2)}} = f'(z_3^{(2)}) = \begin{cases} 1 & \text{when } z_3^{(2)} \geq 0; \\ 0 & \text{when } z_3^{(2)} < 0; \end{cases}$$

$$\because z_3^{(3)} = a_3^{(2)} w_{3,3}^{(3)} + b_3^{(3)}, \therefore \frac{\partial z_3^{(3)}}{\partial a_3^{(2)}} = w_{3,3}^{(3)}$$

$$\because \frac{\partial a_3^{(3)}}{\partial z_3^{(3)}} \frac{\partial C}{\partial a_3^{(3)}} = \frac{\partial C}{\partial z_3^{(3)}} = a_3^{(3)} - y_3 \quad (\text{according to equation (21) in Appendix-B})$$

$$\text{Hence, } \frac{\partial C}{\partial w_{4,3}^{(2)}} = a_4^{(1)} \cdot f'(z_3^{(2)}) \cdot w_{3,3}^{(3)} \cdot (a_3^{(3)} - y_3)$$

Similarly for any layer H2 node's gradient $w_{i,j}^{(2)}$,

$$\frac{\partial C}{\partial w_{i,j}^{(2)}} = a_i^{(1)} \cdot f'(z_j^{(2)}) \cdot w_{j,j}^{(3)} \cdot (a_j^{(3)} - y_j) \quad \dots (22c)$$

I can then use the j -th output node of layer O to calculate the i -th bias of layer H2,

$$\frac{\partial C}{\partial b_i^{(2)}} = \frac{\partial z_i^{(2)}}{\partial b_i^{(2)}} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial z_j^{(3)}}{\partial a_i^{(2)}} \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial C}{\partial a_j^{(3)}} = 1 \cdot f'(z_i^{(2)}) \cdot w_{i,j}^{(3)} \cdot (a_j^{(3)} - y_j)$$

$$\therefore \frac{\partial C}{\partial b_i^{(2)}} = f'(z_i^{(2)}) \cdot w_{i,j}^{(3)} \cdot (a_j^{(3)} - y_j) \quad \dots (22d)$$

Finally, I have included in APPENDIX-C on how to compute the gradients for the edge connecting a node in input layer and a node of layer H1, i.e., $\frac{\partial C}{\partial w_{2,3}^{(1)}}$ in ∇C . This is the longest back propagation case of my NN. The computation is a bit complex because the weight gradient for an input node and an H1 node has the ripple effect on each of H2 layer's 64 nodes' summation and activation, which all contribute to the output layer's summation and activation.

Till now, I have been able to formulate the equations for each layer's weight or bias gradient, using the back propagation method.

Part Three: Experimental Verification

To prove the math findings of my investigations, I decided to perform experimental by-hand verifications on the Feed Forward, Back Propagation and Prediction steps.

1) Feed Forward Verification: Output by-AI vs Output by-hand

Starting with the same randomly initialized set of weights and biases (0s), with an input digital image of “5”, I ran a full feed forward with by-hand calculations and compared the prediction result by AI/tensorflow run. (I wrote a simple calculator to help do by-hand arithmetic and vector calculations as they get extensive quickly.)

By-hand run illustration of I->H1:

$$\begin{bmatrix} 0.000 \\ 0.000 \\ 0.000 \\ 0.000 \\ \vdots \\ x_{784} \end{bmatrix} \cdot \begin{bmatrix} -0.043 & 0.008 & 0.025 & 0.049 & \dots & -0.057 \\ 0.042 & 0.074 & 0.063 & -0.012 & \dots & 0.079 \\ 0.044 & 0.077 & 0.014 & -0.047 & \dots & -0.011 \\ -0.009 & -0.028 & -0.021 & 0.028 & \dots & 0.049 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -0.028 & -0.074 & 0.072 & -0.030 & \dots & -0.010 \end{bmatrix} + \begin{bmatrix} 0.000 \\ 0.000 \\ 0.000 \\ 0.000 \\ \vdots \\ 0.000 \end{bmatrix} \xrightarrow{\text{yield}} \begin{bmatrix} 0.065 \\ 0.275 \\ -0.046 \\ 0.243 \\ \vdots \\ 0.468 \end{bmatrix} \xrightarrow{f() } \begin{bmatrix} 0.065 \\ 0.275 \\ 0.000 \\ 0.243 \\ \vdots \\ 0.468 \end{bmatrix}$$

$X[1,784] \quad W^{(1)}[784,128] \quad B^{(1)}[1,128] \quad Z^{(1)}[1,128] \quad A^{(1)}[1,128]$

By-hand run illustration of H2->O:

$$\begin{matrix} \begin{bmatrix} 0.065 \\ 0.275 \\ 0.000 \\ 0.243 \\ \vdots \\ 0.468 \end{bmatrix} \\ A^{(2)} [1,64] \end{matrix} \cdot \begin{matrix} \begin{bmatrix} -0.153 & 0.027 & 0.088 & 0.171 & \dots & -0.132 \\ 0.041 & -0.095 & 0.017 & -0.183 & \dots & -0.259 \\ 0.137 & 0.164 & -0.090 & -0.175 & \dots & -0.197 \\ -0.195 & -0.205 & 0.252 & -0.215 & \dots & -0.174 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.199 & -0.165 & 0.180 & 0.236 & \dots & 0.134 \end{bmatrix} \\ W^{(3)} [64,10] \end{matrix} + \begin{matrix} \begin{bmatrix} 0.000 \\ 0.000 \\ 0.000 \\ 0.000 \\ \vdots \\ 0.000 \end{bmatrix} \\ B^{(3)} [1,10] \end{matrix} \xrightarrow{\text{yield}} \begin{matrix} \begin{bmatrix} -0.001 \\ -0.200 \\ 0.679 \\ 0.734 \\ \vdots \\ -0.310 \end{bmatrix} \\ Z^{(3)} [1,10] \end{matrix} \xrightarrow{f_O} \begin{matrix} \begin{bmatrix} 0.082 \\ 0.067 \\ 0.162 \\ 0.171 \\ \vdots \\ 0.060 \end{bmatrix} \\ A^{(3)} [1,10] \end{matrix}$$

Final prediction/output result comparison: AI (by-tensorflow) vs Hand (by-hand):

digit	by-hand	by-Tensorflow
0	0.082	8.207457512617111206e-02
1	0.067	6.729900836944580078e-02
2	0.162	1.620792597532272339e-01
3	0.171	1.712321043014526367e-01
4	0.118	1.180373430252075195e-01
5	0.043	4.260931909084320068e-02
6	0.104	1.038775667548179626e-01
7	0.113	1.132866963744163513e-01
8	0.079	7.921724021434783936e-02
9	0.060	6.028690561652183533e-02

Table 3. Comparison of prediction result sets of the NN run by-hand and by-Tensorflow.

I was thrilled to see that they were identical except for different significant numbers!

This proves that my understanding of the math behind the NN is correct. Though

shortly I realized that these predictions are lousy -- the input image is a handwritten

“5” digit: a probability of 0.043 is not good at all. This is because my NN had not

learned (been trained). So I decided to verify the learning (back propagation) next.

2) **Back Propagation Verification:** Gradient updates by-AI vs by-hand

First, I needed to calculate the ∇C . Since the backpropagation goes from right to left, I

decided to follow the same direction, ∇C for B3, W3, then B2, W, and then B1, W1.

According to equation (22a), equation 22b),

$$\frac{\partial C}{\partial w_{i,j}^{(3)}} = a_i^{(2)} \cdot (a_j^{(3)} - y_j) , \quad \frac{\partial C}{\partial b_j^{(3)}} = (a_j^{(3)} - y_j)$$

In vector arithmetic terms, these translate to:

$$\frac{\partial C}{\partial W^{(3)}} = A^{(2)}(A^{(3)} - y) , \quad \frac{\partial C}{\partial B^{(3)}} = (A^{(3)} - y)$$

My calculator was able to compute the ∇C results quickly. Next, based on equations

(15) and (16), update values were computed as below.

$$W_{new} = W - \eta \nabla_{\theta} C(W), \quad B_{new} = B - \eta \nabla_{\theta} C(B)$$

	$W^{(3)}[64,10]$						$B^{(3)}[1,10]$
Original $W^{(3)}, B^{(3)}$	-0.153	0.027	0.088	0.171	...	-0.132	0.000
	0.041	-0.095	0.017	-0.183	...	-0.259	0.000
	0.137	0.164	-0.090	-0.175	...	-0.197	0.000
	-0.195	-0.205	0.252	-0.215	...	-0.174	0.000
	\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
	0.199	-0.165	0.180	0.236	...	0.134	0.000
∇C (gradient)	0.030	0.025	0.059	0.063	...	0.022	0.082
	0.041	0.000	0.000	0.000	...	0.000	0.067
	0.008	0.006	0.015	0.016	...	0.006	0.162
	0.000	0.000	0.000	0.000	...	0.000	0.104
	\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
	0.000	0.000	0.000	0.000	...	0.000	0.060
$\eta \nabla C$ (learning ratio gradient)	7.441e-4	7.441e-4	7.441e-4	7.441e-4	...	7.441e-4	7.441e-4
	0.000	0.000	0.000	0.000	...	0.000	7.441e-4
	7.441e-4	7.441e-4	7.441e-4	7.441e-4	...	7.441e-4	7.441e-4
	0.000	0.000	0.000	0.000	...	0.000	7.441e-4
	\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
	0.000	0.000	0.000	0.000	...	0.000	0.000
New	-0.153	0.026	0.088	0.170	...	-0.133	-7.441e-4
	0.041	-0.095	0.017	-0.183	...	-0.259	-7.441e-4
	0.136	0.163	-0.091	-0.176	...	-0.198	-7.441e-4
	-0.195	-0.205	0.252	-0.215	...	-0.174	-7.441e-4
	\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\vdots
	0.199	-0.165	0.180	0.236	...	0.134	0.000

Table 4. Calculation of ∇C , η , and the new $W^{(3)}$ and $B^{(3)}$. Updated values are in red color. Because my NN uses adaptive learning ratios, the calculation of η follows Adam's algorithm (Kingma, Diederik), see details in APPENDIX-A.

Then I examined the new $W^{(3)}$, And $B^{(3)}$ generated by Tensorflow (in Figure 6a and 6b) and compared with my hand-computed values (Row "New") in Table 4., they are identical except for the significant numbers. This verifies back propagation by-hand!

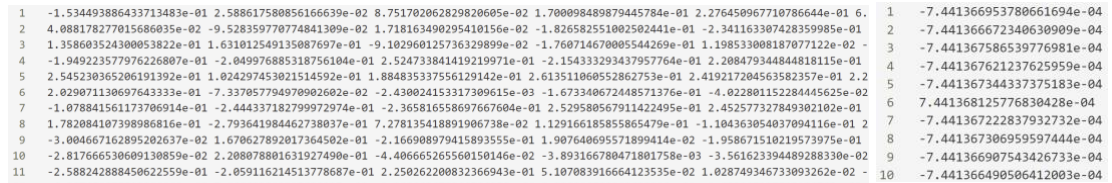


Figure 6a. (left) updated $W^{(3)}$ after 1 iteration of training with input image. Figure 6b. (right) updated $B^{(3)}$ after 1 iteration of training with input image.

Furthermore, I was able to calculate layer H2's and layer H1's gradients and updates for weights and biases and verify each set with the corresponding Tensorflow-generated set of values. (See Appendix-D)

3) Prediction Verification by-hand with AI-trained weights and biases

I then used the 60,000 MNIST dataset to train the NN, with 20 full iterations. From the training log, I can see the precision arrived at optima halfway. Thus, I saved the final “golden” set of Weights & Bias.

W1	778 -1.042288914322853088e-01 2.985296547412872314e-01 -1.063398197293281555e-01 779 -2.264201194047927856e-01 1.911010891199111938e-01 -8.375514298677444458e-02 780 -1.407545898109674454e-02 3.446314856410026550e-02 -9.254395216703414917e-02 781 6.881675869226455688e-02 3.690807521343231201e-02 6.979725509881973267e-02 1. 782 2.774556726217269897e-02 -2.322284132242202759e-02 3.203346580266952515e-02 - 783 5.349981039762496948e-02 3.281154483556747437e-02 2.971933037042617798e-02 -1 784 -1.137443631887435913e-02 5.910747498273849487e-02 -7.451979070901870728e-02 785
B2	60 8.953846991062164307e-02 61 -2.757667005062103271e-02 62 7.635259628295898438e-02 63 3.115259669721126556e-02 64 1.432386189699172974e-01 65
W3	59 -2.002087533473968506e-01 4.063977599143981934e-01 -6.626137495040893555e-01 60 6.077141035348176956e-03 -1.439681053161621094e-01 -2.970965020358562469e-02 61 -1.757010370492935181e-01 -1.644565537571907043e-02 1.128494441509246826e-01 62 3.324718773365020752e-01 -9.427405893802642822e-02 1.268849819898605347e-01 63 2.985727190971374512e-01 1.093931794166564941e-01 -3.905469784513115883e-03 64 -3.340143561363220215e-01 8.533613383769989014e-02 -3.560056388378143311e-01 65

Table 5. Selected partial snapshots of the “golden” weights and biases.

Then I used another test image, a handwritten digit “7”, and ran through feed forward process by-hand with the “golden” weights and biases. Below is its ouput:

Output
[
1.26511553e-11
3.34848538e-11
2.27058594e-09
7.83632743e-07
1.64589414e-12
1.47751115e-13
2.11127085e-19
9.99999166e-01
1.92158459e-11
1.27989965e-08
]

The final output points to the 8th entry of the digits

(0,1,2,3,4,5,6,**7**,8,9), with 99.9999166% probability, which

gives me the same level of confidence to claim the prediction

verification a success. This confirms that my understanding

of the math steps on prediction is correct.

Table 6. Calculation output by-hand.

Conclusion

When I first started the investigation on the Research Question “What is the math behind digital handwritten recognition?”, I was not sure how much math I would find in this topic. Now I have found **functions, derivatives, partial derivatives, vectors and matrices!** They work together to solve complex real-world problems.

One of the surprising findings is that NNs use the near zeroing of the **partial derivatives** of the loss function C , $\frac{\partial C}{\partial W}$, $\frac{\partial C}{\partial B}$ to find the minima of C . Though I have learned this principle in math AA HL, I was pleasantly surprised to see its application here. A second surprise is application of calculus' **chaining rule** when computing these gradients, for example, $\frac{\partial C}{\partial W^{(3)}} = \frac{\partial Z^{(3)}}{\partial W^{(3)}} \frac{\partial A^{(3)}}{\partial Z^{(3)}} \frac{\partial C}{\partial A^{(3)}}$, which accounts for the “rippling effects” of any subtle changes in a weight or bias on the final loss. A third surprise is the rather creative use of **logarithm** in Softmax function to normalize the probabilities. A fourth surprise is the extensive usage of **vectors** to digitalize information – any information can be represented this way, truly eye-opening!

The scope of my study is not big, as it revolves around a simple neural net that I developed to recognize handwritten digits. It could be expanded to recognize more handwritten forms: math symbols, or letters, or non-English characters. The training dataset I used is from MNIST, and it would be helpful to try more training datasets from different sources, so the model gets more diversified training and becomes more adaptive. Also, the NN can be improved to work with higher resolution images.

Today the technology of AI is transforming our world at record pace in areas like face-recognition, real-time translation, diagnosis of diseases, robotics, etc. There must be a lot more fascinating math behind them. It would be cool to learn more on such topics when I get more time. Math is art, power, and magic!

Bibliography

Brain, Neural Network

<https://www.sciencephoto.com/media/638977/view/brain-neural-network>

Brouillette, Monique. "New Brain Maps Can Predict Behaviors". *Quantamagazine*, 6 Dec, 2021. <https://www.quantamagazine.org/new-brain-maps-can-predict-behaviors-20211206/>

Kingma, Diederik. "Adam: a method for stochastic optimization." *arXiv preprint arXiv: 1412.6980v9*, 2017 <https://arxiv.org/pdf/1412.6980.pdf>

Kurbiel, Thomas. "Derivative of the Softmax Function and the Categorical Cross-Entropy Loss". 23 April, 2021. <https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1>

LeCun, Yann. "The MNIST Database of handwriting digits"
<http://yann.lecun.com/exdb/mnist/>

Loech, Kiprono Elijah. "Cross-Entropy Loss Function". 3 Oct, 2020.
<https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e>

London, & Fountas, Zafeirios. Imperial College Spiking Neural Networks for Human-like Avatar Control in a Simulated Environment. 2022.

Mazur, Matt. "A Step by Step Backpropagation Example". 3 Sept, 2021.
<https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>

"Neurons and Glial Cells."
<https://openstax.org/books/biology/pages/35-1-neurons-and-glial-cells>

"Overview of neuron structure and function."
<https://www.khanacademy.org/science/biology/human-biology/neuron-nervous-system/a/overview-of-neuron-structure-and-function>

Patrikar, Sushant. "Batch, Mini Batch & Stochastic Gradient Descent". 1 Oct, 2019.
<https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a>

Pramoditha, Rukshan. "The concept of Artificial Neurons (Perceptrons) in Neural Networks."
<https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc>

Sanderson, Grant. "Neural Networks from the ground up." June 2022.
<http://www.3blue1brown.com/lessons/neural-networks>

Seth, Neha. "Fundamentals Concepts of Neural Network & Deep Learning."

<https://www.analytixlabs.co.in/blog/fundamentals-of-neural-networks/>

Sharma, Sagar. "Activation Functions in Neural Networks". 6 Sept, 2017.

<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

Zulkifli, Hafidz. "Understanding Learning Rates and How It Improves Performance in Deep Learning". 22 Jan, 2018.

<https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10>

APPENDIX-A: Adam's Algorithm

I wrote a small implementation of the Adam optimization algorithm (Kingma, Diederik) to calculate the corresponding learning ratio for each weight or bias item based on current and previous gradient values. It takes a matrix or vector as inputs, calculates the learning rate η (a matching matrix or vector) and computes the new weight vector or bias vector, based on equations:

$$W_{new} = W - \eta \nabla_{\theta} C(W), \quad B_{new} = B - \eta \nabla_{\theta} C(B)$$

Below is Adam's algorithm to calculate a dynamic learning rate, from my understanding of Kingma's classic paper:

Coefficients:

α : step size

β_1 : first moment, $\in [0,1)$

β_2 : second moment, $\in [0,1)$

ϵ : a small constant

Default: ($\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1 * 10^{-8}$)

Require:

$f(\theta)$: Stochastic objective function with parameter θ

θ_0 : Initial parameter vector

m_0 : 1st moment vector, initialize to 0

v_0 : 2nd moment vector, initialize to 0

t : timestep, initialize to 0

while θ_t not converged do

$t = t + 1$

$g_t = \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t stochastic objective at timestep t)

$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second moment estimate)

$\hat{m}_t = \frac{m_t}{(1 - \beta_1^t)}$ (Compute bias-corrected first moment estimate)

$\hat{v}_t = \frac{v_t}{(1 - \beta_2^t)}$ (Compute bias-corrected second raw moment estimate)

$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{(\sqrt{\hat{v}_t} + \epsilon)}$ (Update parameters)

end while

return θ_t (Resulting parameters)

In the case of my NN, $f(\theta)$ will be $C(W)$ or $C(B)$, and θ will be W or B , ∇_{θ} will be $\nabla_{\theta} C(W)$ or $\nabla_{\theta} C(B)$ – the partial derivative vector containing $\frac{\partial C}{\partial W}$ or $\frac{\partial C}{\partial B}$, and α will replace η in equations (15) and (16). From the algorithm, I can see that it automatically adapts the learning rate for each input variable for the objective function and further smooths the search process by using an exponentially decreasing moving average of the gradient to make updates to variables.

APPENDIX-B: Theoretic calculation of gradients of my NN

To calculate the gradient for weights of layer l , according to the chain rule:

$$\frac{\partial \mathcal{C}}{\partial w^{(l)}} = \frac{\partial Z^{(l)}}{\partial w^{(l)}} \frac{\partial A^{(l)}}{\partial Z^{(l)}} \frac{\partial \mathcal{C}}{\partial A^{(l)}} \quad \dots (18)$$

where

$$l \in [1,3]$$

To compute $\frac{\partial Z^{(l)}}{\partial w^{(l)}}$, from equation (8a), conceptually we have

$$Z^{(l)} = A^{(l-1)} \cdot W^{(l)} + B^{(l)} \longrightarrow \frac{\partial Z^{(l)}}{\partial w^{(l)}} = A^{l-1}$$

where when $l = 1$, $A^{l-1} = A^0 = X$.

Hence, the partial derivative of the cost function w.r.t the weight connecting the i -th node on layer $(l-1)$ and the j -th node on layer (l) is calculated as:

$$\frac{\partial \mathcal{C}}{\partial w_{i,j}^{(l)}} = a_i^{(l-1)} \quad \dots (18a)$$

✧ To compute $\frac{\partial A^{(l)}}{\partial Z^{(l)}}$, according to equation (5):

$$A^{(l)} = f(Z^{(l)}) \longrightarrow \frac{\partial A^{(l)}}{\partial Z^{(l)}} = f'(Z^{(l)})$$

Since activation function f varies by layer, f' varies by layer:

In the case of layer H1 and H2, my NN uses Relu. Relu takes one input and output one value, a simple 1:1 mapping. So, I can easily calculate its derivative:

$$\begin{aligned} \because f(z_i^{(l)}) &= \begin{cases} 0, & z_i^{(l)} < 0 \\ z_i^{(l)}, & z_i^{(l)} \geq 0 \end{cases} \\ \therefore f'(z_i^{(l)}) &= \begin{cases} 1 & \text{when } z_i^{(l)} \geq 0; \\ 0 & \text{when } z_i^{(l)} < 0; \end{cases} \end{aligned} \quad \dots (18b)$$

In the case of Softmax in the output layer, it is more complex because it takes a vector $Z^{(3)}$ of size 10 as input and output a vector $A^{(3)}$ of size 10.

$$a_i^{(3)} = f(z_i^{(3)}) = \frac{e^{z_i^{(3)}}}{\sum_{k=1}^{10} e^{z_k^{(3)}}}, i \in [1,10]$$

So, the partial derivative $\frac{\partial A^{(l)}}{\partial Z^{(l)}}$ will take the form of $\frac{\partial a_i^{(3)}}{\partial z_j^{(3)}}$, $i, j \in [1,10]$

$$\frac{\partial a_i^{(3)}}{\partial z_j^{(3)}} = \frac{\frac{\partial e^{z_i^{(3)}}}{\sum_{k=1}^{10} e^{z_k^{(3)}}}}{\frac{\partial \sum_{k=1}^{10} e^{z_k^{(3)}}}{\partial z_j^{(3)}}}$$

For simplicity, we use Σ to stand for $\sum_{k=1}^{10} e^{z_k^{(3)}}$, using the quotient rule of derivatives,

- When $i = j$,

$$\begin{aligned} \therefore \frac{\partial a_i^{(3)}}{\partial z_j^{(3)}} &= \frac{\frac{\partial e^{z_i^{(3)}}}{\Sigma}}{\frac{\partial \Sigma}{\partial z_j^{(3)}}} = \frac{e^{z_i^{(3)}} \Sigma - e^{z_j^{(3)}} e^{z_i^{(3)}}}{\Sigma^2} = \frac{e^{z_i^{(3)}}}{\Sigma} \left(1 - \frac{e^{z_j^{(3)}}}{\Sigma} \right) \\ \therefore \frac{\partial a_i^{(3)}}{\partial z_j^{(3)}} &= f(z_i^{(3)}) (1 - f(z_j^{(3)})) \end{aligned}$$

- When $i \neq j$,

$$\begin{aligned} \therefore \frac{\partial a_i^{(3)}}{\partial z_j^{(3)}} &= \frac{\frac{\partial e^{z_i^{(3)}}}{\Sigma}}{\frac{\partial \Sigma}{\partial z_j^{(3)}}} = \frac{0 - e^{z_j^{(3)}} e^{z_i^{(3)}}}{\Sigma^2} = -\frac{e^{z_i^{(3)}}}{\Sigma} \frac{e^{z_j^{(3)}}}{\Sigma} \\ \therefore \frac{\partial a_i^{(3)}}{\partial z_j^{(3)}} &= -f(z_i^{(3)}) f(z_j^{(3)}) \end{aligned}$$

To sum up, for softmax function's partial derivative w.r.t $z_i^{(3)}$, we have

$$\frac{\partial a_i^{(3)}}{\partial z_j^{(3)}} = f(z_i^{(3)}) \cdot \left(1\{i = j\} - f(z_j^{(3)})\right) = a_i^{(3)} \cdot (1\{i = j\} - a_j^{(3)}) \quad \dots (19)$$

where condition $\{i = j\}$ takes value of 1 if true, or value of 0 if false.

✧ To compute $\frac{\partial C}{\partial A^{(l)}}$, according to equation (13):

$$C = -\sum_{i=1}^{10} (y_i \log_e a_i^{(3)})$$

$$\text{when } l < L, \frac{\partial C}{\partial A^{(l)}} = \sum_{i=0}^{n_{l+1}-1} W^{l+1} f'(z^{(l+1)}) \frac{\partial C}{\partial A^{(l+1)}}$$

where n is the node number of a layer;

note we have a recursion here: $\frac{\partial C}{\partial A^{(l)}}$ calling $\frac{\partial C}{\partial A^{(l+1)}}$ on all nodes of layer $(l + 1)$ and sums them up, this makes perfect sense because during the feed forward process $A^{(l)}$ is fed into all nodes of layer $(l + 1)$; $\frac{\partial C}{\partial A^{(l+1)}}$ then further calls $\frac{\partial C}{\partial A^{(l+2)}}$ on all nodes of layer $(l + 2)$ and sums them up, ..., the recursion goes until the output layer is reached.

Suppose we are trying to find gradient for weight $w_{jk}^{(l)}$, the edge connecting the k -th node on layer $l-1$ and j -th node on layer l ,

Hence, when $1 \leq l < L$, we have the recursive form:

$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{i=0}^{n_{l+1}-1} w_{ji}^{(l+1)} f'(z_i^{(l+1)}) \frac{\partial C}{\partial a_i^{(l+1)}} \quad \dots (20a)$$

When $l = L$, or, $l = 3$,

$$\frac{\partial C}{\partial a_i^{(3)}} = \left[-\sum_{i=1}^{10} (y_i \log_e a_i^{(3)}) \right]' = -\frac{y_i}{a_i^{(3)}} \quad \dots (20b)$$

where i is the node index number of the output layer L

At the output layer, the recursion reaches the bottom case.

✧ Better yet, when $l = L$, or, $l = 3$, if I try to compute $\frac{\partial A^{(l)}}{\partial Z^{(l)}} \frac{\partial C}{\partial A^{(l)}}$ together, that is, to compute $\frac{\partial C}{\partial Z^{(3)}}$ directly, we can greatly simplify the calculations (Kurbiel, Thomas).

$$\begin{aligned} C &= -\sum_{i=1}^{10} (y_i \log_e a_i^{(3)}), \\ \frac{\partial C}{\partial z_j^{(3)}} &= -\frac{\partial}{\partial z_j^{(3)}} \sum_{i=1}^{10} (y_i \log_e a_i^{(3)}) = -\sum_{i=1}^{10} y_i \cdot \frac{\partial}{\partial z_j^{(3)}} \log_e a_i^{(3)} \\ &= -\sum_{i=1}^{10} \frac{y_i}{a_i^{(3)}} \cdot \frac{\partial a_i^{(3)}}{\partial z_j^{(3)}} \end{aligned}$$

From equation (19), we can replace $\frac{\partial a_i^{(3)}}{\partial z_j^{(3)}}$ with $a_i^{(3)} \cdot (1\{i = j\} - a_j^{(3)})$, hence

$$\begin{aligned} \frac{\partial C}{\partial z_j^{(3)}} &= -\sum_{i=1}^{10} \frac{y_i}{a_i^{(3)}} \cdot a_i^{(3)} \cdot (1\{i = j\} - a_j^{(3)}) = -\sum_{i=1}^{10} y_i \cdot (1\{i = j\} - a_j^{(3)}) \\ \frac{\partial C}{\partial z_j^{(3)}} &= \sum_{i=1}^{10} y_i \cdot a_j^{(3)} - \sum_{i=1}^{10} y_i \cdot 1\{i = j\} \end{aligned}$$

The indicator function $1\{i = j\}$ takes a value of 1 for $i = j$ and 0 in other cases:

$$\frac{\partial C}{\partial z_j^{(3)}} = \sum_{i=1}^{10} y_i \cdot a_j^{(3)} - y_j$$

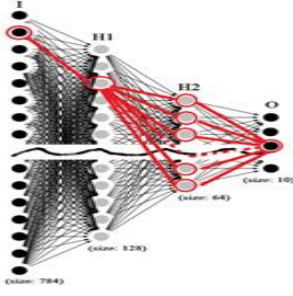
Because target vector y_i sums to 1, so we have

$$\frac{\partial C}{\partial z_j^{(3)}} = a_j^{(3)} - y_j \quad \dots (21)$$

This is a much-simplified approach as it saves two layers of derivative calculations. I have found it so much useful in the following studies.

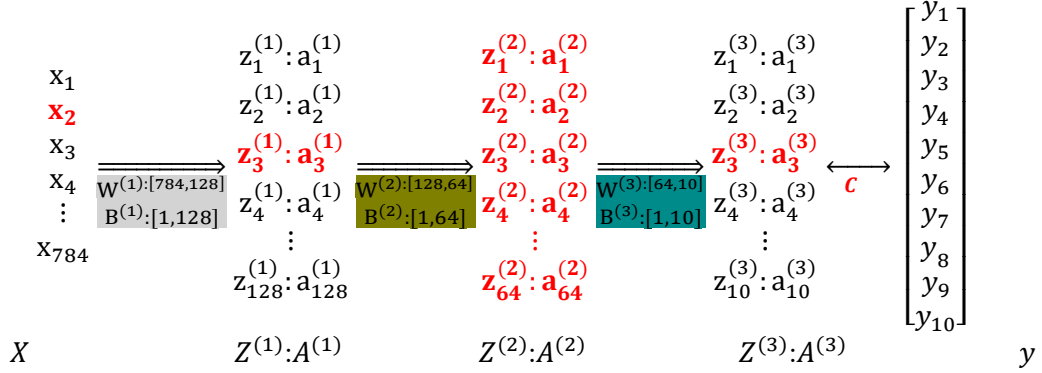
APPENDIX-C. Computation of gradients for layer H1

For example, I want to compute the gradient for the edge connecting the 2nd value of input layer X and third node of layer H1, $\frac{\partial C}{\partial w_{2,3}^{(1)}}$ in ∇C . This is the longest back propagation case of



my NN. It is also the most complex case as the weight gradient for an input node and an H1 node has the ripple effect on each of H2 layer's 64 nodes' summation and activation, which all contribute to the output layer's summation and activation. For example, the third node of layer O will sum, activate and output based on 64 edges of inputs from H2. Figure C-1 illustrates this.

Figure C-1. A weight edge propagates its rippling effects on C in a hammock-shaped pattern.



Let i be the node index for H2 layer, according to equation (17) and equation (20a),

$$\begin{aligned} \frac{\partial C}{\partial w_{2,3}^{(1)}} &= \frac{\partial z_3^{(1)}}{\partial w_{2,3}^{(1)}} \frac{\partial a_3^{(1)}}{\partial z_3^{(1)}} \frac{\partial C}{\partial a_3^{(1)}} = \frac{\partial z_3^{(1)}}{\partial w_{2,3}^{(1)}} \frac{\partial a_3^{(1)}}{\partial z_3^{(1)}} \sum_{i=1}^{64} [w_{3,i}^{(2)} f'(z_i^{(2)}) \frac{\partial z_3^{(3)}}{\partial a_i^{(2)}}] \frac{\partial C}{\partial z_3^{(3)}} \\ &= \frac{\partial z_3^{(1)}}{\partial w_{2,3}^{(1)}} \frac{\partial a_3^{(1)}}{\partial z_3^{(1)}} \sum_{i=1}^{64} [w_{3,i}^{(2)} f'(z_i^{(2)}) \frac{\partial z_3^{(3)}}{\partial a_i^{(2)}}] \frac{\partial a_3^{(3)}}{\partial z_3^{(3)}} \frac{\partial C}{\partial a_3^{(3)}} \\ \because z_3^{(1)} &= a_2^{(0)} w_{2,3}^{(1)} + b_3^{(1)}, \quad \therefore \frac{\partial z_3^{(1)}}{\partial w_{2,3}^{(1)}} = a_2^{(0)} = x_2 \\ \because a_3^{(1)} &= f(z_3^{(1)}), \quad \therefore \frac{\partial a_3^{(1)}}{\partial z_3^{(1)}} = f'(z_3^{(1)}) \\ \because z_3^{(3)} &= \sum_{i=1}^{64} a_i^{(2)} w_{i,3}^{(3)} + b_3^{(3)}, \quad \therefore \frac{\partial z_3^{(3)}}{\partial a_i^{(2)}} = w_{i,3}^{(3)} \\ \because \frac{\partial a_3^{(3)}}{\partial z_3^{(3)}} \frac{\partial C}{\partial a_3^{(3)}} &= \frac{\partial C}{\partial z_3^{(3)}} = a_3^{(3)} - y_3 \quad (\text{see equation (21)}) \\ \therefore \frac{\partial C}{\partial w_{2,3}^{(1)}} &= x_2 \cdot f'(z_3^{(1)}) \cdot \sum_{i=1}^{64} [w_{3,i}^{(2)} \cdot f'(z_i^{(2)}) \cdot w_{i,3}^{(3)}] \cdot (a_3^{(3)} - y_3) \end{aligned}$$

Similarly for any weight gradient $w_{i,j}^{(1)}$, let's use k to count the node index in H2,

$$\frac{\partial C}{\partial w_{i,j}^{(1)}} = x_i \cdot f'(z_j^{(1)}) \cdot \sum_{k=1}^{64} [w_{j,k}^{(2)} \cdot f'(z_k^{(2)}) \cdot w_{k,j}^{(3)}] \cdot (a_j^{(3)} - y_j) \quad \dots (22e)$$

I can then use the j -th output node of layer O to calculate the i -th bias of layer H2, I can also compute the gradient of bias in layer H1:

$$\begin{aligned} \frac{\partial C}{\partial b_i^{(1)}} &= \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \frac{\partial C}{\partial a_i^{(1)}} = \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \sum_{k=1}^{64} [w_{i,k}^{(2)} f'(z_k^{(2)}) \frac{\partial C}{\partial a_k^{(2)}}] \\ &= \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \sum_{k=1}^{64} [w_{i,k}^{(2)} f'(z_k^{(2)}) \frac{\partial z_3^{(3)}}{\partial a_k^{(2)}}] \frac{\partial a_3^{(3)}}{\partial z_3^{(3)}} \frac{\partial C}{\partial a_3^{(3)}} \\ &= f'(z_i^{(1)}) \sum_{k=1}^{64} [w_{i,k}^{(2)} f'(z_k^{(2)}) w_{k,j}^{(3)}] \cdot (a_j^{(3)} - y_j) \quad \dots (22f) \end{aligned}$$

APPENDIX-D: Calculation of layer H2's and layer H1's gradients and updates

I further took a look at layer H2's gradients and updates for weights and biases, according to equations (22c) and (22d), for individual items, I get

$$\frac{\partial C}{\partial w_{i,j}^{(2)}} = a_i^{(1)} \cdot f' \left(z_j^{(2)} \right) \cdot w_{j,j}^{(3)} \cdot (a_j^{(3)} - y_j)$$

$$\frac{\partial C}{\partial b_i^{(2)}} = f' \left(z_i^{(2)} \right) \cdot w_{i,j}^{(3)} \cdot (a_j^{(3)} - y_j)$$

if I use vectorized computation, which is my favorite way now because it is so powerful, the above are reduced to the following vector/matrix operations:

$$\frac{\partial C}{\partial W^{(2)}} = A^{(1)} \cdot (Z^{(2)} > 0) \cdot (W^{(3)} \cdot (A^{(3)} - y))$$

$$\frac{\partial C}{\partial B^{(2)}} = (Z^{(2)} > 0) \cdot (W^{(3)} \cdot (A^{(3)} - y))$$

I was able to calculate the new weights and biases for H2 layer and verify them with tensorflow generated values – they are the same. Table A-1 shows the calculations.

	$W^{(2)}[128,64]$	$B^{(2)}[1,64]$
Original $W^{(2)}, B^{(2)}$	<div> <div> -0.043 0.008 0.025 0.049 ... -0.057 0.042 0.074 0.063 -0.012 ... 0.079 0.044 0.077 0.014 -0.047 ... -0.011 -0.009 -0.028 -0.021 0.028 ... 0.049 ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ -0.028 -0.074 0.072 -0.030 ... -0.010 </div> </div>	<div> <div> 0.000 0.000 0.000 0.000 ⋮ 0.000 </div> </div>
∇C (gradient)	<div> <div> -0.004 -0.000 0.006 0.000 ... 0.000 -0.018 -0.000 0.026 0.000 ... 0.000 -0.000 -0.000 0.000 0.000 ... 0.006 -0.016 -0.000 0.023 0.000 ... 0.000 ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ -0.031 -0.000 0.044 0.000 ... 0.000 </div> </div>	<div> <div> -0.067 -0.000 0.093 0.000 ⋮ 0.000 </div> </div>
$\eta \nabla C$ (delta: learning ratio · gradient)	<div> <div> -7.441e-4 0.000 7.441e-4 0.000 ... 0.000 -7.441e-4 0.000 7.441e-4 0.000 ... 0.000 0.000 0.000 0.000 0.000 ... 0.000 -7.441e-4 0.000 7.441e-4 0.000 ... 0.000 ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ -7.441e-4 0.000 7.441e-4 0.000 ... 0.000 </div> </div>	<div> <div> -7.441e-4 0.000 7.441e-4 0.000 ⋮ 0.000 </div> </div>
New	<div> <div> -0.095 0.017 0.055 0.106 ... 0.158 0.153 -0.029 0.044 -0.152 ... -0.125 0.092 0.161 0.138 -0.025 ... -0.169 0.069 -0.023 -0.125 -0.166 ... 0.172 ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ 0.027 0.080 0.097 -0.054 ... 0.145 </div> </div>	<div> <div> 7.441e-4 0.000 -7.441e-4 0.000 ⋮ 0.000 </div> </div>

Table A-1. Calculation of ∇C , η , and the new $W^{(2)}$ and $B^{(2)}$. Updated values in red.

For layer H1, according to equations (22e) and (22f),

$$\frac{\partial C}{\partial w_{i,j}^{(1)}} = x_i \cdot f' \left(z_j^{(1)} \right) \cdot \sum_{k=1}^{64} [w_{j,k}^{(2)} \cdot f'(z_k^{(2)}) \cdot w_{k,j}^{(3)}] \cdot (a_j^{(3)} - y_j)$$

$$\frac{\partial C}{\partial b_i^{(2)}} = f' \left(z_i^{(1)} \right) \sum_{k=1}^{64} [w_{i,k}^{(2)} f'(z_k^{(2)}) w_{k,j}^{(3)}] \cdot (a_j^{(3)} - y_j)$$

Similarly, I translated them to vectorized forms,

$$\frac{\partial C}{\partial W^{(1)}} = X \cdot ((Z^{(1)} > 0) \cdot (W^{(2)} \cdot ((Z^{(2)} > 0) \cdot W^{(3)} \cdot (A^{(3)} - y)))$$

$$\frac{\partial C}{\partial B^{(1)}} = (Z^{(1)} > 0) \cdot (W^{(2)} \cdot ((Z^{(2)} > 0) \cdot W^{(3)} \cdot (A^{(3)} - y)))$$

Again, I was able to calculate the new weights and biases for H1 layer and verify them with tensorflow generated values. Table A-2 shows the calculations.

	$W^{(1)}[784,128]$	$B^{(1)}[1,128]$
Original $W^{(2)}, B^{(2)}$	<div><div><div>−0.0430.0080.0250.049…−0.057 0.0420.0740.063−0.012…0.079 0.0440.0770.014−0.047…−0.011 −0.009−0.028−0.0210.028…0.049 ⋮⋮⋮⋮⋮⋮ −0.028−0.0740.072−0.030…−0.010</div></div></div>	<div><div><div>0.000 0.000 0.000 0.000 ⋮ 0.000</div></div></div>
∇C (gradient)	<div><div><div>−0.000−0.0000.0000.000…0.000 −0.000−0.0000.0000.000…0.000 −0.000−0.0000.0000.000…0.000 −0.000−0.0000.0000.000…0.000 ⋮⋮⋮⋮⋮⋮ −0.0−0.0000.0000.000…0.000</div></div></div>	<div><div><div>−0.072 −0.025 0.000 −0.033 ⋮ 0.000</div></div></div>
$\eta \nabla C$ (delta: learning ratio <i>gradient</i>)	<div><div><div>0.0000.0000.0000.000…0.000 0.0000.0000.0000.000…0.000 0.0000.0000.0000.000…0.000 0.0000.0000.0000.000…0.000 ⋮⋮⋮⋮⋮⋮ 0.0000.0000.0000.000…0.000</div></div></div>	<div><div><div>−7.441e−4 −7.441e−4 0.000 −7.441e−4 ⋮ 0.000</div></div></div>
New	<div><div><div>−0.0430.0080.0250.049…−0.057 0.0420.0740.063−0.012…0.079 0.0440.0770.014−0.047…−0.011 −0.009−0.028−0.0210.028…0.049 ⋮⋮⋮⋮⋮⋮ −0.028−0.0740.072−0.030…−0.010</div></div></div>	<div><div><div>7.441e−4 7.441e−4 0.000 7.441e−4 ⋮ 7.441e−4</div></div></div>