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?

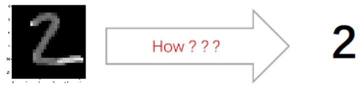


Fig 1. Essence of the problem.

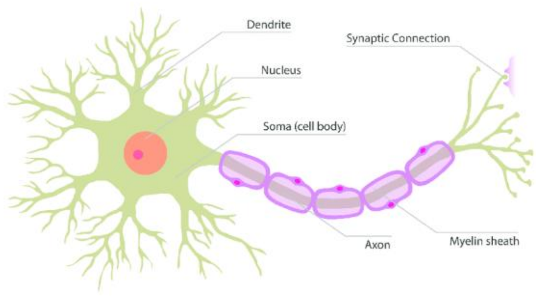
The problem in essence is, given an image grid of certain size (say, 28 X 28 = 784 pixels), how to detect the digit number (0-9) in the grid. A pixel is the basic square-shaped unit in an computer image and has a value of 0-1, denoting the brightness ratio. After some initial attempts with traditional 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 research and studies led me to Neural Nets (NN), computing systems inspired by the biological neural networks that consitute human brains. I was able to code up a python program that uses the Tensorflow 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 after 20 epochs. (1 epoch means one full pass/iteration of the dataset.)

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 trains and learns and my experimentation. I will explore the theoretical parts first, then follow up with the experimentation part.

Part One: Human Neurons and Artificial Neural Nets (NNs)

Scientists borrowed the idea of NNs from human brain's neurons and neural networks, so I also read up on the human counterparts before studying the artificial neural nets. The following are my findings.

1.1 Human Neurons and Neural Networks



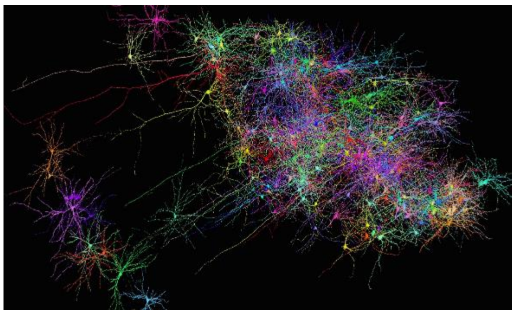
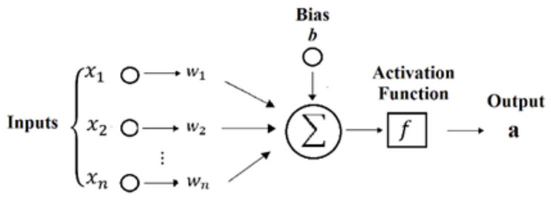


Fig 2a. Illustration of a neuron. Fig 2b. Imaging of human neural networks formed by neurons.

Human nervous system is made up of neurons, specialized cells that can receive and transmit chemical or electrical signals. Figure 2a illustrates the structure of a human neuron (London et al). The neuron transmits a signal from one to another by first taking inputs from the dendrites, then activating if the signal passes the threshold, and passing on along the axon the activated signal to next neuron through a joint structure called synapse. Inside a human brain, there are billions of neurons connecting to each other to form complex neural networks. Figure 2b is an image of human brain neural networks from just one cubic millimeter of human brain (Monique Brouillette).

1.2 Artificial Neural Nodes



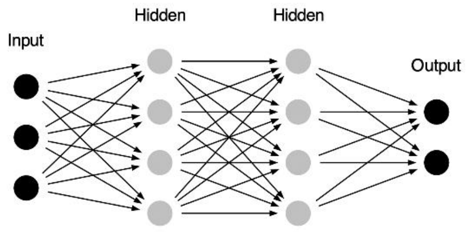


Fig 3a. A typical neural node in a NN

Fig 3b. 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. A neural node is illustrated in figure 3a, 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), i.e.

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

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

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

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

2. Activation of the input signal using function $f(z)$, with a as the output

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

In a NN, data flows from one layer of nodes to another layer of nodes. Within a particular layer of n nodes, all nodes share the same activation function; suppose the layer takes inputs from a vector X of size m, it then uses a vector B storing each node's biases, $b_{1,}b_{2,}\ b_{3,}\ \ldots, b_{n,}$ 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 $z_{1,}\ z_{2,}\ z_{3,}\ \ldots, z_n$). Like equation (2) which is for one node, we have the following for a layer of n nodes:

$$Z\ =\ X \cdot W + B \qquad \ldots (4)$$

Similarly, if we use vector A to store all the activation results ($a_{1,}\ a_{2,}\ a_{3,}\ \ldots$) of the nodes in the layer, we have:

$$A = f\ (Z) \qquad \ldots (5)$$

The activation function $f(z)$ is essential to the function of the neural node because it adds non-linearity to the signal, just like the selective activation mechanism of a human neuron. Table 1 lists some of the common ones.

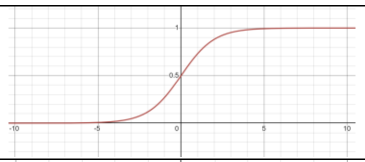
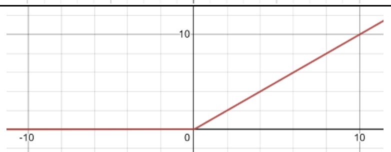
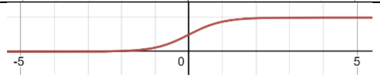
Name	$\boldsymbol{f(z)}$	**Graph**	**Range**
Sigmoid	$f(z) = \dfrac{1}{1+e^{-z}}$		(0,1)
Relu	$f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$		$(0, \infty)$
Softmax	$f(z_i) = \dfrac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}}$		(0,1)

Table 1. Summary of some common activation functions.

Sigmoid has a range of (0,1) and is popular in simple binary classification situations. 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 takes vectors of real numbers as inputs, and normalizes them into a probability distribution proportional to the exponentials of the input numbers. This is very useful in multi-classification.

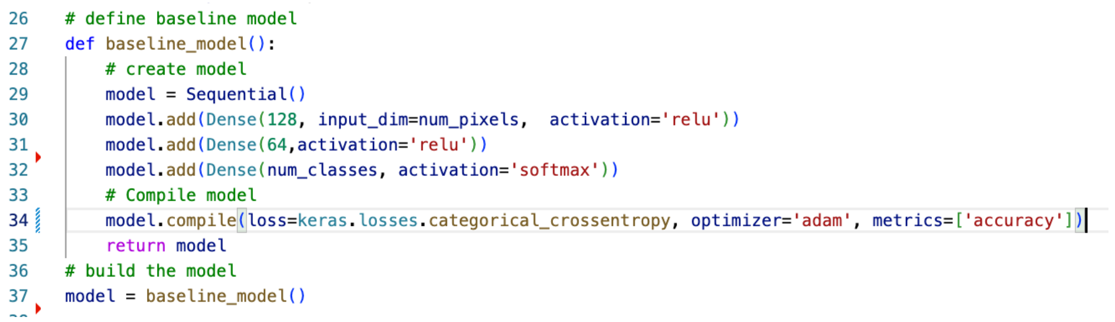
z scores	Softmax Activation	Probability Distribution
$\begin{bmatrix} \mathbf{2.0} \\ \mathbf{1.0} \\ \mathbf{0.1} \end{bmatrix}$	$f(z_i) = \dfrac{e^{z_i}}{\sum_{k=1}^{N} e^{z_k}}$	$\begin{bmatrix} 0.7 \\ 0.2 \\ 0.1 \end{bmatrix}$

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

1.3 Artificial Neural Nets

In Figure 3b, I drew a simple artificial NN with the following elements: 1 input layer, 2 hidden layers and 1 output layer. The input layer takes raw input from the domain. Nodes here just pass on the information (features) to the next hidden layer. The nodes of the hidden layers are not exposed. A hidden layer performs all kinds of computations on the features entered through the previous layer (input layer or hidden layer) and transfers the result to the next layer (a hidden layer or output layer). 3) 1 output Layer: It's the final layer of the network that brings the information learned through the hidden layers and delivers the final values as a result.

My python program defines the following NN for handwriting digits recognition:



```python
# define baseline model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(128, input_dim=num_pixels,  activation='relu'))
    model.add(Dense(64,activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss=keras.losses.categorical_crossentropy, optimizer='adam', metrics=['accuracy'])
    return model
# build the model
model = baseline_model()
```

Figure 4. My NN's definition in python code. * num_pixels = 28 * 28 = 784, num_classes = 10

As defined in the code, my NN has an input layer (I) that takes 784 input values (28 by 28) 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, finally flows the data to the output layer (O) which has 10 nodes and uses Softmax for activation, computing and outputting the probabilities of the input image belonging to digits 0-9. Figure 5 below has topological and hierarchical views of the model layers.

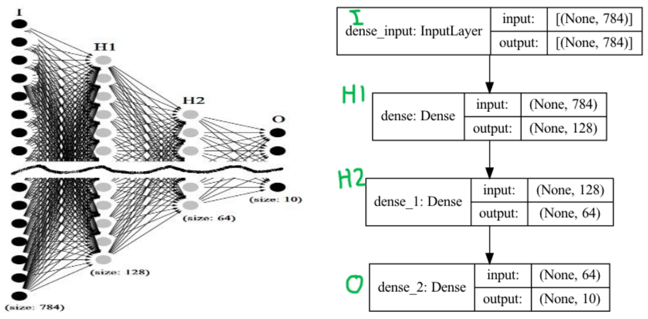


Figure 5. My NN's topology and hierarchy, with the curve representing the undrawn parts.

Part Two: How NNs Train and Learn

For my NN to work well, it needs to be "trained" by running through lots of training data examples with labels and "learn", before it can predict with good accuracy. How?

Through literature research I learned that a NN's behavior is determined by its weights and biases. The weights represent the strength of edge connections between each neuron in layer $(l-1)$ and each neuron in layer (l). And each bias is an indication of whether its neuron tends to be activated or not. A NN achieves training and learning through the following:

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 back propagation to adjust each node's weights and bias values to minimize the loss and improve precision of outputs
5. keep doing steps 1) to 3) until good precision and minimal loss are achieved.

In the following I will start with a handwritten digit's image (28 X 28 pixel), investigate how this input data feeds forward across all the layers of my NN, how losses (errors) are computed, and how the NN learns through back-propagation.

2.1 The Feed Forward and Prediction

Starting with a 28 by 28 pixel image (array) of a handwritten digit, the first thing I would need to do is to flatten the 2 dimensional array into a one dimensional vector X of length 28 * 28 = 784, denoted as X [1,784]. Figure 6 shows the process.

$$\begin{pmatrix} x_1 & x_2 & \cdots & x_{28} \\ x_{29} & x_{30} & \cdots & x_{56} \\ \vdots & \vdots & \ddots & \vdots \\ x_{757} & x_{758} & \cdots & x_{784} \end{pmatrix} \xrightarrow{Flatten} \begin{matrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{784} \end{matrix}$$

Figure 6. Flattening of the image data into vector X

2.1.1 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], we denote it as $W^{(1)}$, meaning the weight matrix for the first layer of the NN. 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., $w_{2,1}^{(1)}$ means the weight value in $W^{(1)}$ for the connection edge between the second node (x_2) in Input layer and the first node in H1. It is important to note that, for a new NN, all weight values are initialized with random numbers and all the biases as 0s.

According to equation (4), we have the summation equation as follows,

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

The underlying computation is illustrated below.

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

If we look at the third node of H1 layer and want to know the weight value of the second input value $\boldsymbol{x_2}$, the corresponding one would be $\boldsymbol{w_{2,3}^{(1)}}$. If we look at how all input values of X go through this node, the summation for this node will be:

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

The summation completes after each of the 128 nodes of H1 layer has run through each of the 784 input values. 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:

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

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.

$$\begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \\ z_4^{(1)} \\ \vdots \\ z_{128}^{(1)} \end{bmatrix} \xrightarrow{a_i^{(1)}=f\left(z_i^{(1)}\right)} \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ a_4^{(1)} \\ \vdots \\ a_{128}^{(1)} \end{bmatrix}$$

$Z^{(1)}$ [1,128] $A^{(1)}$ [1,128]

2.1.2 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

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

So, generally,

$$Z^{(l)} = A^{(l-1)} \cdot W^{(l)} + B^{(l)} \qquad \dots (8a)$$

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

The underlying computation is as follows.

$$\begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ a_4^{(1)} \\ \vdots \\ a_{128}^{(1)} \end{bmatrix} \cdot \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} & w_{1,3}^{(2)} & w_{1,4}^{(2)} & \cdots & w_{1,64}^{(2)} \\ w_{2,1}^{(2)} & w_{2,2}^{(2)} & w_{2,3}^{(2)} & w_{2,4}^{(2)} & \cdots & w_{2,64}^{(2)} \\ w_{3,1}^{(2)} & w_{3,2}^{(2)} & w_{3,3}^{(2)} & w_{3,4}^{(2)} & \cdots & w_{3,64}^{(2)} \\ w_{4,1}^{(2)} & w_{4,2}^{(2)} & w_{4,3}^{(2)} & w_{4,4}^{(2)} & \cdots & w_{4,64}^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{128,1}^{(2)} & w_{128,2}^{(2)} & w_{128,3}^{(2)} & w_{128,4}^{(2)} & \cdots & w_{128,64}^{(2)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \\ b_3^{(2)} \\ b_4^{(2)} \\ \vdots \\ b_{64}^{(2)} \end{bmatrix} \xrightarrow{yields} \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \\ z_4^{(2)} \\ \vdots \\ z_{64}^{(2)} \end{bmatrix}$$

$A^{(1)}$[1,128] $W^{(2)}$[128,64] $B^{(2)}$ [1,64] $Z^{(2)}$ [1,64]

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

$$\underset{Z^{(2)}\,[1,64]}{\begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \\ z_4^{(2)} \\ \vdots \\ z_{64}^{(2)} \end{bmatrix}} \xrightarrow{a_i^{(2)}=f\left(z_i^{(2)}\right)} \underset{A^{(2)}\,[1,64]}{\begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \\ a_4^{(2)} \\ \vdots \\ a_{64}^{(2)} \end{bmatrix}}$$

2.1.3 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 as well. Hence,

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

The underlying computation is as follows.

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

The activation function for O is Softmax, which I covered in section 1.3 above. Accordingly, we have

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

For each node in layer O, the activation step uses Softmax function to normalize its summation value against all 10 nodes and produces a probability:

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

I chose Softmax because it is most suited for multi-classification (Sharma, Sagar): my NN intends to classify an image into one of 10 classifications (0, 1, 2…, 9). The following shows the mapping between Z, A and classifications:

$$\underset{Z^{(3)}\ [1,10]}{\begin{bmatrix} z_1^{(3)} \\ \boldsymbol{z_2^{(3)}} \\ z_3^{(3)} \\ z_4^{(3)} \\ \vdots \\ z_{10}^{(3)} \end{bmatrix}} \xrightarrow{a_i^{(3)}=f\left(z_i^{(3)}\right)} \underset{A^{(3)}\ [1,10]}{\begin{bmatrix} a_1^{(3)} = 0.001 \\ \boldsymbol{a_2^{(3)} = 0.950} \\ a_3^{(3)} = 0.002 \\ a_4^{(3)} = 0.005 \\ \vdots \\ a_{10}^{(3)} = 0.012 \end{bmatrix}} \quad \cdots \quad \underset{\text{digits}}{\begin{bmatrix} 0 \\ \mathbf{1} \\ 2 \\ 3 \\ 4 \\ 5 \\ \vdots \\ 9 \end{bmatrix}}$$

After the activation $Z^{(3)}$: $A^{(3)}$, $A^{(3)}$ will be a vector of 10 prediction values that add up to 1.0, each of which is the computed prediction probability of the input image matching the corresponding digit of the same index in the classification vector (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.

2.2 Cost/Loss Function and “Adam” Stochastic 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 that labels the target probabilities (0 or 1) of the input matching the 10 digits (0-9). To measure the performance of my NN, I need a cost/loss function C to calculate the error between the prediction vector $A^{(3)}$ and target vector y.

$$\underset{A^{(3)}\ [1,10]}{\begin{matrix} a_1^{(3)} \\ \boldsymbol{a_2^{(3)}} \\ a_3^{(3)} \\ a_4^{(3)} \\ \vdots \\ a_{10}^{(3)} \end{matrix}} \quad \underset{C}{\longleftrightarrow} \quad \underset{y\ [1,10]}{\begin{bmatrix} y_1 = 0.000 \\ y_2 = 0.000 \\ y_3 = 0.000 \\ y_4 = 0.000 \\ \vdots \\ \vdots \\ y_{10} = 0.000 \end{bmatrix}}$$

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)

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

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)

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

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

I chose CCE over MSE, because 1) CCE excels in classification tasks where the decision boundary is large while MSE is strong in regression cases and does not punish misclassifications enough; 2) CCE works with probability values output by Softmax better.

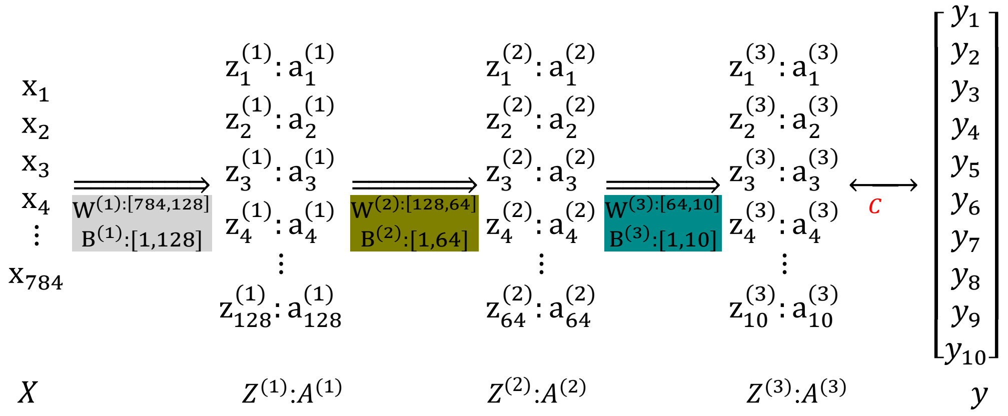
The function C is for computing the loss of one training example and reflects how much the NN needs to improve to be more precise. If the NN trains on a batch of N examples, then the average cost is calculated as:

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

where N is the size of the training batch.

2.2.2 Gradient Descent, SGD and "Adam" Optimization

Now that my NN has completed one full pass of feed forward of the training example and determined the corresponding 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 (along with the three fixed activation functions) across the layers, so we need to find the set of weights and biases that minimize the cost function – this is essentially what "learning" is about.



Suppose the cost function takes just one weight as variable, from calculus we know that the minima of $C(w)$ is at where its derivative $\frac{dC}{dw}(w) = 0$, which is easy to calculate. However, 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 109,184 + 202 = 109,386 variables! For such a complicated cost function, computing the exact minimum directly isn't going to work. How to find its minimum then?

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 minimum or really near it. It is like a ball rolling down a hill. Figure 7 illustrates the gradient descent process for one variable and multi-variable functions.

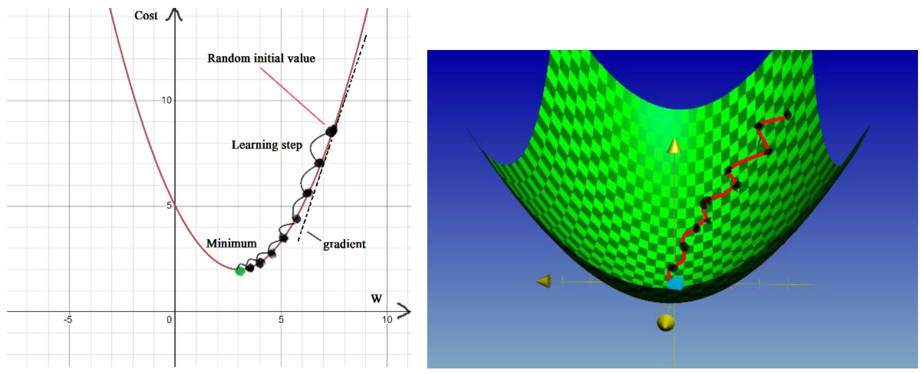


Figure 7. Illustration of gradient descent for 1 variable (left) and multi-variable (right) functions.

In multivariable calculus, a gradient vector ∇C containing partial derivatives is used to represent the direction of 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)

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

where

- ➢ η is the learning rate, which can be a configurable constant like 0.01, 0.1 or an adaptive value in cases like the "Adam" optimization, which my NN uses
- ➢ 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,

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

There are basically three types of gradient descent: Batch Gradient Descent (GD), Minibatch Gradient Descent (Minibatch) and Stochastic Gradient Descent (SGD). GD computes accumulative gradients on the whole dataset per iteration and then update the weights and biases; Minibatch divides the data set into mini-batches, computes accumulative gradients on a mini-batch and then updates; SGD takes one random example each time, computes the gradient and then updates the weights and biases. GD is thorough but is slow to converge and requires large memory, SGD updates frequently and converges fast with small memory fingerprints but could be choppy, Minibatch has vectorization processing capabilities but is rigid in execution order (lacks randomness) when running through the batches. On top of them the AI community has built different optimization extensions, such as AdaGrad, RmsProp, Adam, SGD, etc.

I chose "Adam", an extension that combines strengths of SGD and Minibatch as my

NN's gradient optimizer, so it has Minibatch's vectorized processing capability and SGD's frequent updating and fast gradient converging on large datasets. Another reason I like "Adam" (Kingma, Diederik) is that it uses adaptive learning rate instead of a fixed one which could be too slow or causes overshooting problems. (See Appendix-A for Adam's Algorithm in details) Figure 8 illustrates how learning rates affect the training loss as the iterations (epochs) go on.

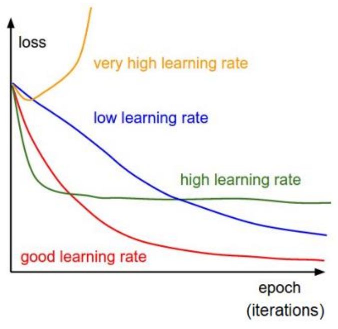


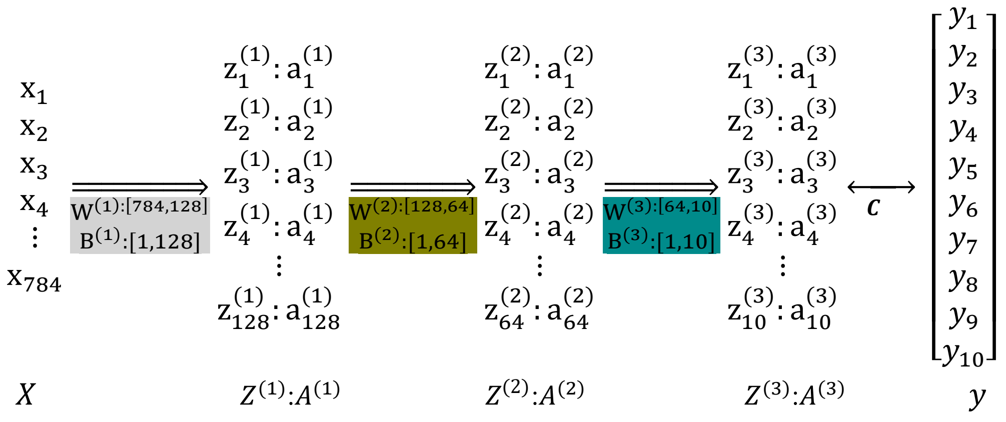
Figure 8. How learning rates affect loss/cost of training. (Zulkifli, Hafidz.)

The entries of the gradient vector ∇C contain the partial derivatives of the cost function w.r.t all the weights and biases of the NN, such entries 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 go over how my NN calculates ∇C, which also takes the form of $\nabla_\theta C(W)$ w.r.t W and $\nabla_\theta C(B)$ w.r.t B and has a total of 109,386 entries.

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



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 output 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, conceptually, we have

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

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:

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

$where$

$$l \in [1, L], L\ being\ the\ output\ layer's\ layer\ number$$

✧ 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:

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

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

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

In the case of layer H1 and H2, my NN uses Relu:

$$\because f\left(z_i^{(l)}\right) = \begin{cases} 0, & z_i^{(l)} < 0 \\ z_i^{(l)}, & z_i^{(l)} \geq 0 \end{cases}, \therefore f'\left(z_i^{(l)}\right) = \begin{cases} 1\ when\ z_i^{(l)} \geq 0; \\ 0\ when\ z_i^{(l)} < 0; \end{cases} \qquad \ldots (18b)$$

For output layer O, I get: (See Appendix-B for details)

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

✧ To compute $\frac{\partial C}{\partial A^{(l)}}$, (see details in Appendix-B)

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.*

When $1 \leq l < L$, I get the recursive form:

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

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

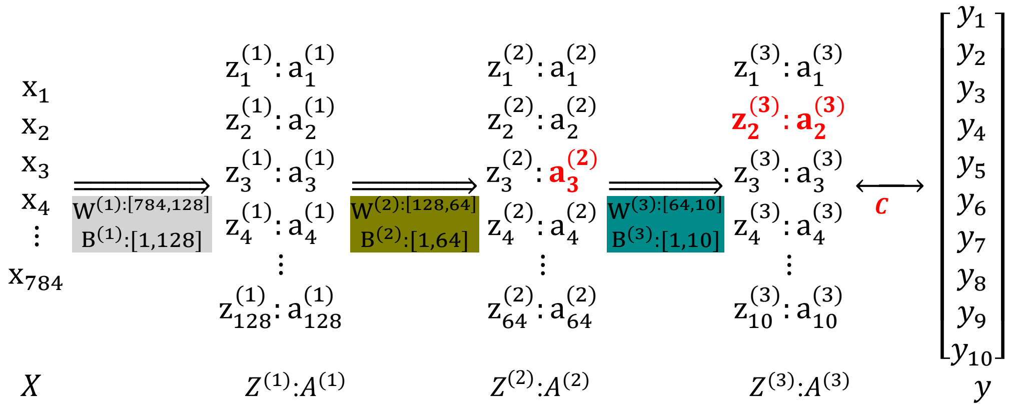
$$\frac{\partial C}{\partial a_i^{(3)}} = \left[-\sum_{i=1}^{10}\left(y_i \log_e a_i^{(3)}\right)\right]' = -\frac{y_i}{a_i^{(3)}} \qquad \ldots (20b)$$

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

✧ When $l = 3$, for the remaining part of equation (18), $\frac{\partial C}{\partial Z^{(l)}}$ could be directly calculated (See Appendix-B for details.) as follows

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

I decided to further pursue the definitive gradient formulas for each layer's weights and biases. Suppose I want to compute the weight gradient $\frac{\partial \boldsymbol{C}}{\partial \boldsymbol{w}_{3,2}^{(3)}}$ in ∇C,



x_1 x_2 x_3 x_4 ⋮ x_{784}
$W^{(1)}$:[784,128] $B^{(1)}$:[1,128]
$W^{(2)}$:[128,64] $B^{(2)}$:[1,64]
$W^{(3)}$:[64,10] $B^{(3)}$:[1,10]
$z_1^{(1)}:a_1^{(1)}$ … $z_{128}^{(1)}:a_{128}^{(1)}$
$z_1^{(2)}:a_1^{(2)}$ … $z_{64}^{(2)}:a_{64}^{(2)}$
$z_1^{(3)}:a_1^{(3)}$ … $z_{10}^{(3)}:a_{10}^{(3)}$
C
y_1 … y_{10}
X
$Z^{(1)}:A^{(1)}$
$Z^{(2)}:A^{(2)}$
$Z^{(3)}:A^{(3)}$
y

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

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

From the simplification in equation (21), we get

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

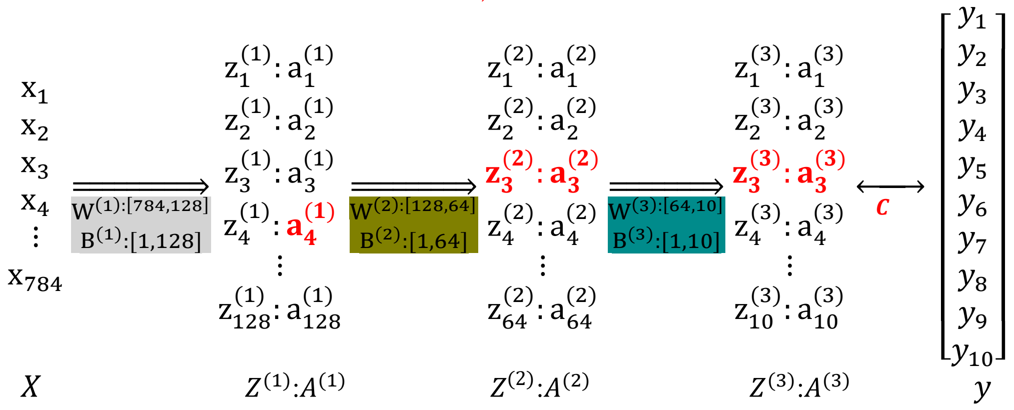
It is easy to generalize for any weight gradient $w_{i,j}^{(3)}$ in layer O:

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

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

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

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



x_1 x_2 x_3 x_4 ⋮ x_{784}
$W^{(1)}$:[784,128] $B^{(1)}$:[1,128]
$W^{(2)}$:[128,64] $B^{(2)}$:[1,64]
$W^{(3)}$:[64,10] $B^{(3)}$:[1,10]
$z_1^{(1)}:a_1^{(1)}$ … $z_{128}^{(1)}:a_{128}^{(1)}$
$z_1^{(2)}:a_1^{(2)}$ … $z_{64}^{(2)}:a_{64}^{(2)}$
$z_1^{(3)}:a_1^{(3)}$ … $z_{10}^{(3)}:a_{10}^{(3)}$
C
y_1 … y_{10}
X
$Z^{(1)}:A^{(1)}$
$Z^{(2)}:A^{(2)}$
$Z^{(3)}:A^{(3)}$
y

Again, according to the chain rule of calculus,

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

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

$$\because a_3^{(2)} = f(z_3^{(2)}), \therefore \frac{\partial a_3^{(2)}}{\partial z_3^{(2)}} = f'\left(z_3^{(2)}\right)$$

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

$$\because \frac{\partial a_3^{(3)}}{\partial z_3^{(3)}} \frac{\partial C}{\partial a_3^{(3)}} = \frac{\partial C}{\partial z_3^{(3)}} = a_3{}^{(3)} - y_3 \quad \text{(see equation (21))}$$

Hence,

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

Similarly for any gradient $w_{i,j}^{(2)}$,

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

I can then use the *j-th* output node of layer O to calculate the *i-th* bias of layer H2, I can easily compute the gradient of the *i-th* bias in layer H2.

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

$$\frac{\partial C}{\partial b_i^{(2)}} = f'\left(z_i^{(2)}\right) \cdot w_{i,j}^{(3)} \cdot \left(a_j{}^{(3)} - y_j\right) \qquad \ldots (22d)$$

Finally, I will seek 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 as the input layer is reached. It is also the most complex case as the weight gradient for a 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 9 illustrates the hammock-shaped process, with the affected edges and nodes in red.

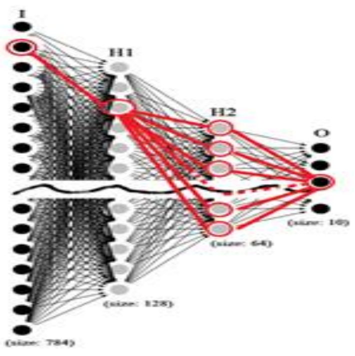
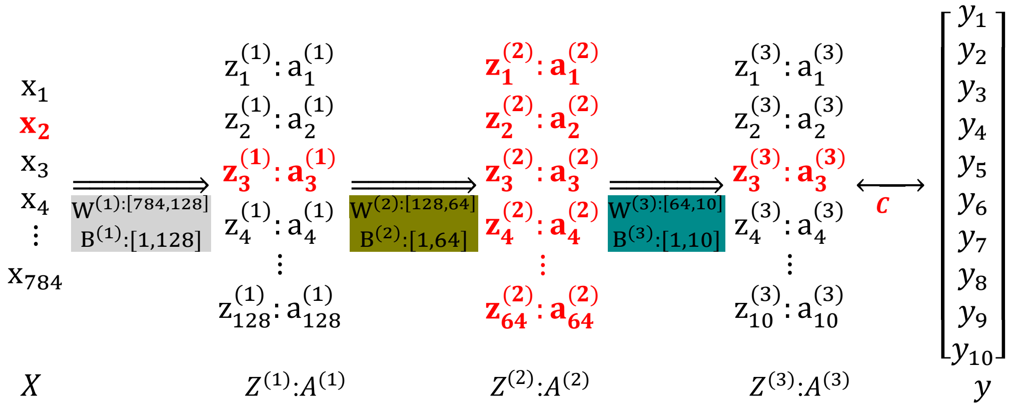


Figure 9. A weight edge propagates its rippling effects on C in a hammock-shaped pattern.



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

$$\frac{\partial C}{\partial w_{2,3}^{(1)}} = \frac{\partial z_3^{(1)}}{\partial w_{2,3}^{(1)}} \frac{\partial a_3^{(1)}}{\partial z_3^{(1)}} \frac{\partial C}{\partial a_3^{(1)}} = \frac{\partial z_3^{(1)}}{\partial w_{2,3}^{(1)}} \frac{\partial a_3^{(1)}}{\partial z_3^{(1)}} \sum_{i=1}^{64} \left[w_{3,i}^{(2)} \, f'\left(z_i^{(2)}\right) \frac{\partial z_3^{(3)}}{\partial a_i^{(2)}} \right] \frac{\partial C}{\partial z_3^{(3)}}$$

$$= \frac{\partial z_3^{(1)}}{\partial w_{2,3}^{(1)}} \frac{\partial a_3^{(1)}}{\partial z_3^{(1)}} \sum_{i=1}^{64}[w_{3,i}^{(2)} f'(z_i^{(2)}) \frac{\partial z_3^{(3)}}{\partial a_i^{(2)}}] \frac{\partial a_3^{(3)}}{\partial z_3^{(3)}} \frac{\partial C}{\partial a_3^{(3)}}$$

$$\because z_3^{(1)} = a_2^{(0)} w_{2,3}^{(1)} + b_3^{(1)}, \therefore \frac{\partial z_3^{(1)}}{\partial w_{2,3}^{(1)}} = a_2^{(0)} = x_2$$

$$\because a_3^{(1)} = f(z_3^{(1)}), \therefore \frac{\partial a_3^{(1)}}{\partial z_3^{(1)}} = f'\left(z_3^{(1)}\right)$$

$$\because z_3^{(3)} = \sum_{i=1}^{64} a_i^{(2)} w_{i,3}^{(3)} + b_3^{(3)}, \therefore \frac{\partial z_3^{(3)}}{\partial a_i^{(2)}} = w_{i,3}^{(3)}$$

$$\because \frac{\partial a_3^{(3)}}{\partial z_3^{(3)}} \frac{\partial C}{\partial a_3^{(3)}} = \frac{\partial C}{\partial z_3^{(3)}} = {a_3}^{(3)} - y_3 \quad \text{(see equation (21))}$$

$$\therefore \frac{\partial C}{\partial w_{2,3}^{(1)}} = x_2 \cdot f'\left(z_3^{(1)}\right) \cdot \sum_{i=1}^{64}[w_{3,i}^{(2)} \cdot f'(z_i^{(2)}) \cdot \boldsymbol{w_{i,3}^{(3)}}] \cdot ({a_3}^{(3)} - y_3)$$

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

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

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:

$$\begin{aligned}
\frac{\partial C}{\partial b_i^{(1)}} &= \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \frac{\partial C}{\partial a_i^{(1)}} = \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \sum_{k=1}^{64}[w_{i,k}^{(2)} f'(z_k^{(2)}) \frac{\partial C}{\partial a_k^{(2)}}] \\
&= \frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial z_i^{(1)}} \sum_{k=1}^{64}[w_{i,k}^{(2)} f'(z_k^{(2)}) \frac{\partial z_j^{(3)}}{\partial a_k^{(2)}}] \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial C}{\partial a_j^{(3)}} \\
&= 1 \cdot f'\left(z_i^{(1)}\right) \sum_{k=1}^{64}[w_{i,k}^{(2)} f'(z_k^{(2)}) w_{k,j}^{(3)}] \cdot ({a_j}^{(3)} - y_j) \\
&= f'\left(z_i^{(1)}\right) \sum_{k=1}^{64}[w_{i,k}^{(2)} f'(z_k^{(2)}) w_{k,j}^{(3)}] \cdot ({a_j}^{(3)} - y_j) \qquad \ldots \text{(22f)}
\end{aligned}$$

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

Part Three: Experimental Exploration and Verification

It has been fun to explore the theoretical parts of the math behind my NN. Below is my plan of experimental verification:

1) **Feed Forward Verification**:
   Starting with a common training image as input, and a common set of random weights and biases for the NN, do a math run-through by hand and a Tensorflow run-through of the NN, record and compare their prediction result sets

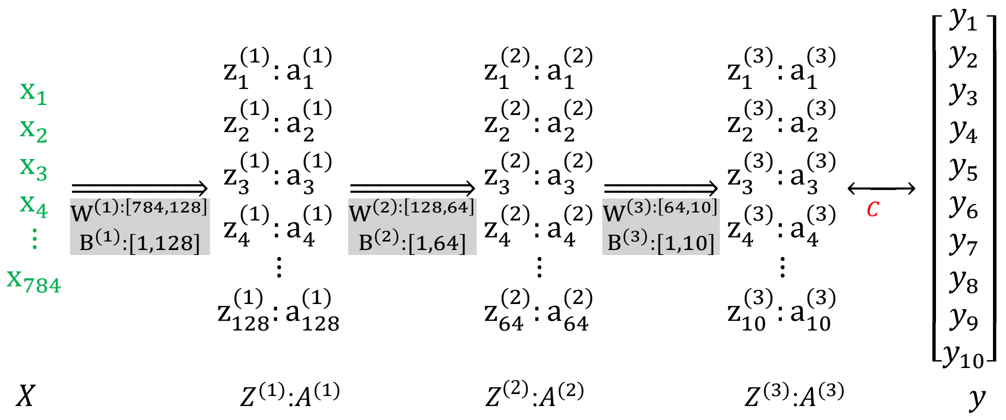
2) **Back Propagation Verification**:
   Compute by-hand costs, and gradients of a few weights and biases, their training ratios, and projected update values of the runs in 1); compare these with the values generated by the Tensorflow run

3) **Computer Training with MNIST Dataset:** Do a complete Tensorflow training on my NN with MNIST dataset, save the final set of Weights & Bias

4) **Prediction Verification**:
   Use another test image, and do prediction with math by hand with the final set of

weights and bias in 3), to see how it predicts

3.1 Feed Forward Verification:



Because my NN initializes the weights random numbers and biases with 0, it is important that I start with the same random set of weights and biases for both by-hand and computer execution. After the model initialization by Tensorflow, I set the debugger to break in and was able to dump out and save the newly initialized weights and biases: $W^{(1)}, B^{(1)}, W^{(2)}, B^{(2)}, W^{(3)}, B^{(3)}$.

Then I used the handwritten image in Figure 6 as input X to feed forward by hand into layer H1 of my NN. According to equation (6), $Z^{(1)} = X \cdot W^{(1)} + B^{(1)}$, and equation (7a) with Relu as the activation function, $a_i^{(1)} = f\left(z_i^{(1)}\right)$, also with known $X, W^{(1)}, B^{(1)}$, I calculated $\boldsymbol{Z^{(1)}}$, $\boldsymbol{A^{(1)}}$ using the following:

$$\underset{X[1,784]}{\begin{bmatrix}0.000\\0.000\\0.000\\0.000\\\vdots\\x_{784}\end{bmatrix}} \cdot \underset{W^{(1)}[784,128]}{\begin{bmatrix}-0.043 & 0.008 & 0.025 & 0.049 & \cdots & -0.057\\0.042 & 0.074 & 0.063 & -0.012 & \cdots & 0.079\\0.044 & 0.077 & 0.014 & -0.047 & \cdots & -0.011\\-0.009 & -0.028 & -0.021 & 0.028 & \cdots & 0.049\\\vdots & \vdots & \vdots & \vdots & \ddots & \vdots\\-0.028 & -0.074 & 0.072 & -0.030 & \cdots & -0.010\end{bmatrix}} + \underset{B^{(1)}[1,128]}{\begin{bmatrix}0.000\\0.000\\0.000\\0.000\\\vdots\\0.000\end{bmatrix}} \xrightarrow{yield} \underset{Z^{(1)}[1,128]}{\begin{bmatrix}0.065\\0.275\\-0.046\\0.243\\\vdots\\0.468\end{bmatrix}} \xrightarrow{f()} \underset{A^{(1)}[1,128]}{\begin{bmatrix}\mathbf{0.065}\\\mathbf{0.275}\\\mathbf{0.000}\\\mathbf{0.243}\\\vdots\\\mathbf{0.468}\end{bmatrix}}$$

**3 significant numbers are used; X vector is a flatten vector of the 28-by-28 pixel handwritten digit "5", many elements (including the beginning ones and trailing ones) are 0.000s.*

I did not really use a pencil and a sheet of paper to do all the calculate these things, as they get extensive quickly. Instead, I wrote a "calculator" in python using library named Numpy that supports vector/matrix arithmetic operations. Repeating the same process for Layer H2, and according to equation (8), with Relu as the activation function, $Z^{(2)} = A^{(1)} \cdot W^{(2)} + B^{(2)}$, $a_i^{(2)} = f\left(z_i^{(2)}\right)$, I got:

$$\underset{A^{(1)}[1,128]}{\begin{bmatrix}0.065\\0.275\\0.000\\0.243\\\vdots\\0.468\end{bmatrix}} \cdot \underset{W^{(2)}[128,64]}{\begin{bmatrix}-0.095 & 0.017 & 0.055 & 0.106 & \cdots & 0.158\\0.153 & -0.029 & 0.045 & -0.152 & \cdots & -0.125\\0.092 & 0.161 & 0.138 & -0.025 & \cdots & -0.169\\0.069 & -0.023 & -0.125 & -0.166 & \cdots & 0.172\\\vdots & \vdots & \vdots & \vdots & \ddots & \vdots\\0.026 & 0.080 & 0.098 & -0.054 & \cdots & 0.145\end{bmatrix}} + \underset{B^{(2)}[1,64]}{\begin{bmatrix}0.000\\0.000\\0.000\\0.000\\\vdots\\0.000\end{bmatrix}} \xrightarrow{yield} \underset{Z^{(2)}[1,64]}{\begin{bmatrix}0.365\\-0.455\\0.094\\-0.804\\\vdots\\-0.380\end{bmatrix}} \xrightarrow{f()} \underset{A^{(2)}[1,64]}{\begin{bmatrix}\mathbf{0.365}\\\mathbf{0.000}\\\mathbf{0.094}\\\mathbf{0.000}\\\vdots\\\mathbf{0.000}\end{bmatrix}}$$

And similarly for Layer O, according to equation (9) and equation (11), with Softmax as activation function, $Z^{(3)} = A^{(2)} \cdot W^{(3)} + B^{(3)}$, $a_i^{(3)} = f\left(z_i^{(3)}\right)$, I was able to calculate $Z^{(3)}$ and the prediction probability vector $A^{(3)}$:

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

The same input X also ran through my NN program (via tensorflow framework, with this input as the sole sample, batch size=1) and generated a prediction output:

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 proven to be correct! Though shortly I realized that these predictions are lousy -- the input image is a handwritten "5" digit, and a good result set should have a high probability for digit 5. A probability of 0.043 (by hand) or the like (by Tensorflow) 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.

3.2 Back Propagation Verification

First, I needed to calculate the ∇C, which has **109,386** entries, one for each weight, one for each bias. Since the backpropagation goes from right to left, I decided to follow the same direction to do the calculations, ∇C for B3, W3, then B2, W, and then B1, W1. According to equation (22a), equation 22b),

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

In vector arithmetic, these translate to:

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

Next, my calculator was able to cook up the results quickly. I also wrote a small implementation of the Adam optimization algorithm (Kingma, Diederi) 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 (15) and (16).

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

Because Adam optimization algorithm calculates and keeps a specific learning rate that adapts per iteration of training, η has 1:1 mapping with ∇, so

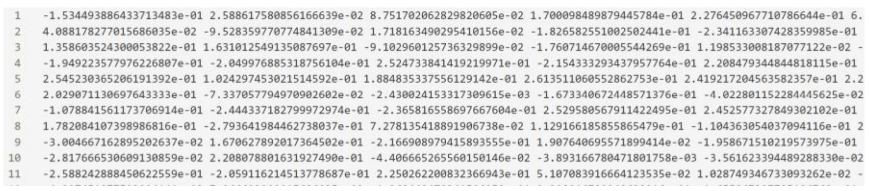
$$W_{new} = W - \eta \odot \nabla_\theta C(W), B_{new} = B - \eta \odot \nabla_\theta C(B)$$

where ⊙ denotes the Hadamard product operand. Table 4 shows the calculated values.

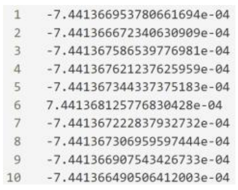
		$W^{(3)}[64,10]$	$B^{(3)}[1,10]$
Original $W^{(3)}, B^{(3)}$		$\begin{matrix} -0.153 & 0.027 & 0.088 & 0.171 & \cdots & -0.132 \\ 0.041 & -0.095 & 0.017 & -0.183 & \cdots & -0.259 \\ 0.137 & 0.164 & -0.090 & -0.175 & \cdots & -0.197 \\ -0.195 & -0.205 & 0.252 & -0.215 & \cdots & -0.174 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.199 & -0.165 & 0.180 & 0.236 & \cdots & 0.134 \end{matrix}$	$\begin{matrix} 0.000 \\ 0.000 \\ 0.000 \\ 0.000 \\ \vdots \\ 0.000 \end{matrix}$
∇C (gradient)		$\begin{matrix} 0.030 & 0.025 & 0.059 & 0.063 & \cdots & 0.022 \\ 0.041 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ 0.008 & 0.006 & 0.015 & 0.016 & \cdots & 0.006 \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \end{matrix}$	$\begin{matrix} 0.082 \\ 0.067 \\ 0.162 \\ 0.104 \\ \vdots \\ 0.060 \end{matrix}$
$\eta \odot \nabla C$ (learning ratio)		$\begin{matrix} 7.441e-4 & 7.441e-4 & 7.441e-4 & 7.441e-4 & \cdots & 0.7.441e-4 \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ 7.441e-4 & 7.441e-4 & 7.441e-4 & 7.441e-4 & \cdots & 0.7.441e-4 \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \end{matrix}$	$\begin{matrix} 7.441e-4 \\ 7.441e-4 \\ 7.441e-4 \\ 7.441e-4 \\ \vdots \\ 0.000 \end{matrix}$
New		$\begin{matrix} -0.153 & 0.026 & 0.088 & 0.170 & \cdots & -0.133 \\ 0.041 & -0.095 & 0.017 & -0.183 & \cdots & -0.259 \\ 0.136 & 0.163 & -0.091 & -0.176 & \cdots & -0.198 \\ -0.195 & -0.205 & 0.252 & -0.215 & \cdots & -0.174 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.199 & -0.165 & 0.180 & 0.236 & \cdots & 0.134 \end{matrix}$	$\begin{matrix} -7.441e-4 \\ -7.441e-4 \\ -7.441e-4 \\ -7.441e-4 \\ \vdots \\ 0.000 \end{matrix}$

Table 4. Calculation of ∇C, η, and the new $W^{(3)}$ *and* $B^{(3)}$. The weight and bias values that are updated are in red color.

Then I examined the new $W^{(3)}$, And $B^{(3)}$ generated by Tensorflow and compared with my hand-computed values in Table 4., they are identical except for the significant numbers (See Figure 10a and 10b). This verifies that my computations by-hand are correct, and that my implementation of the Adam algorithm is correct too.



```
-1.534493886433713483e-01 2.588617580856166639e-02 8.751702062829820605e-02 1.700098489879445784e-01 2.276450967710786644e-01 6.
4.088178277015686035e-02 -9.528359770774841309e-02 1.718163490295410156e-02 -1.826582551002502441e-01 -2.341163307428359985e-01
1.358603524300053822e-01 1.631012549135087697e-01 -9.102960125736329899e-02 -1.760714670005544269e-01 1.198533008187077122e-02 -
-1.949223577976226807e-01 -2.049976885318756104e-01 2.524733841419219971e-01 -2.154333293437957764e-01 2.208479344844818115e-01
2.545230365206191392e-01 1.024297453021514592e-01 1.884835337556129142e-01 2.613511060552862753e-01 2.419217204563582357e-01 2.2
2.029071130697643333e-01 -7.337057794970902602e-02 -2.430024153317309615e-03 -1.673340672448571376e-01 -4.022801152284445625e-02
-1.078841561173706914e-01 -2.444337182799972974e-01 -2.365816558697667604e-01 2.529580567911422495e-01 2.452577327849302102e-01
1.782084107398986816e-01 -2.793641984462738037e-01 7.278135418891906738e-02 1.129166185855865479e-01 -1.104363054037094116e-01 2
-3.004667162895202637e-02 1.670627892017364502e-01 -2.166908979415893555e-01 1.907640695571899414e-02 -1.958671510219573975e-01
-2.817666530609130859e-02 2.208078801631927490e-01 -4.406665265560150146e-02 -3.893166780471801758e-03 -3.561623394489288330e-02
-2.588242888450622559e-01 -2.059116214513778687e-01 2.250262200832366943e-01 5.107083916664123535e-02 1.028749346733093262e-02 -
```



```
-7.441366953780661694e-04
-7.441366672340630909e-04
-7.441367586539776981e-04
-7.441367621237625959e-04
-7.441367344337375183e-04
7.441368125776830428e-04
-7.441367222837932732e-04
-7.441367306959597444e-04
-7.441366907543426733e-04
-7.441366490506412003e-04
```

Figure 10a. updated $W^{(3)}$ after 1 iteration of training with input image.

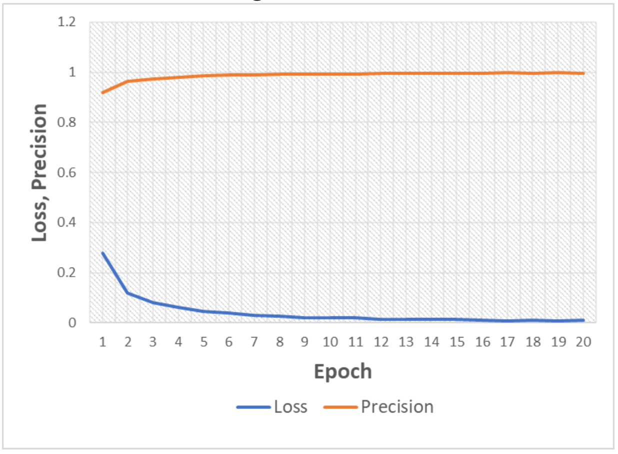
Figure 10b. updated $B^{(3)}$ after 1 iteration of training with input image.

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 – they were the same. (See Appendix-C for more details.)

So far, I have been able to verify the forward propagation and back propagation with one single handwritten digital image. Obviously my NN needs a ton of data to train and learn. So I went ahead and did more extensive training of the NN.

3.3 Computer Training with MNIST Dataset:

For the full training, I used the 60,000 MNIST dataset to train the NN, with batch size set to 64, and epochs set to 20 (1 full data pass counts as 1 epoch). For each epoch the NN will do 60,000/64 ~=938 times of backward propagation computations on weights and biases updates. For 20 epochs there will be a total of 18,760 times of updates, while each update does extensive computations (as show above) and modifies 109,386 variables! When doing a batch of N samples, as show in equation (14), the cost will be accumulated and then divided by N; an average cost is used to compute the gradients. Figure 11 below shows the Loss vs Epoch and Precision vs Epoch trends of the training.



I can see the Loss ratio declined much faster in the beginning iterations and converged to near optima after ~ 10 epochs. The precision value has similar trend. This is Adam optimizer's strength – converging really fast on large datasets.

Figure 11. Loss~Epoch, Precision~Epoch plots. BatchSize=64, Epochs=10

Now that my NN has been extensively trained, I wanted to use its trained weights and biases to do prediction verification by-hand. So I used the same tricks to save them, namely: $W^{(1)}$, $B^{(1)}$, $W^{(2)}$, $B^{(2)}$, $W^{(3)}$, $B^{(3)}$. Below are some partial snapshots of the "golden" weights and biases.

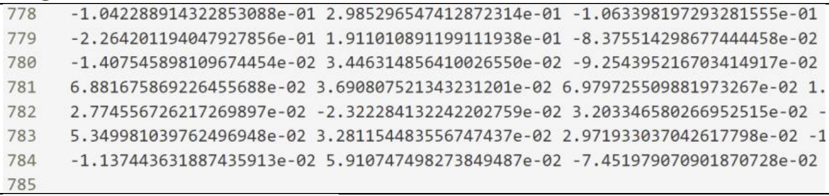
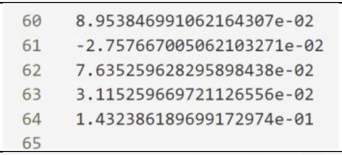
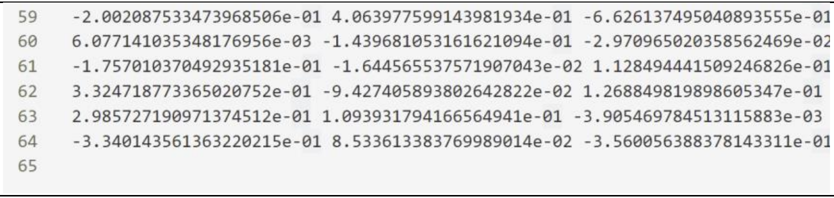
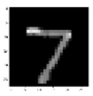
W1	 -1.042288914322853088e-01 2.985296547412872314e-01 -1.063398197293281555e-01 -2.264201194047927856e-01 1.911010891199111938e-01 -8.375514298677444458e-02 -1.407545898109674454e-02 3.446314856410026550e-02 -9.254395216703414917e-02 6.881675869226455688e-02 3.690807521343231201e-02 6.979725509881973267e-02 1. 2.774556726217269897e-02 -2.322284132242202759e-02 3.203346580266952515e-02 - 5.349981039762496948e-02 3.281154483556747437e-02 2.971933037042617798e-02 -1 -1.137443631887435913e-02 5.910747498273849487e-02 -7.451979070901870728e-02
B2	 8.953846991062164307e-02 -2.757667005062103271e-02 7.635259628295898438e-02 3.115259669721126556e-02 1.432386189699172974e-01
W3	 -2.002087533473968506e-01 4.063977599143981934e-01 -6.626137495040893555e-01 6.077141035348176956e-03 -1.439681053161621094e-01 -2.970965020358562469e-02 -1.757010370492935181e-01 -1.644565537571907043e-02 1.128494441509246826e-01 3.324718773365020752e-01 -9.427405893802642822e-02 1.268849819898605347e-01 2.985727190971374512e-01 1.093931794166564941e-01 -3.905469784513115883e-03 -3.340143561363220215e-01 8.533613383769989014e-02 -3.560056388378143311e-01

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

3.4 Prediction Verification:


is my test image, a 28 by 28 pixel handwritten digit 7. The image's data is flattened to an input vector of size 784 (28*28 = 784), and I used my "calculator" to carry on the feed forward process. Pasted below is a script I

used to do the calculations; I simplified its logic so it is more readable.

Script	Output
# $Z^{(1)} = X \cdot W^{(1)} + B^{(1)}$	`[`
`test_z1 = add(dot(X,W1), B1)`	`1.26511553e-11`
# $A^{(1)} = f(Z^{(1)})$	`3.34848538e-11`
`test_a1 = Relu(test_z1)`	`2.27058594e-09`
# $Z^{(2)} = X \cdot W^{(2)} + B^{(2)}$	`7.83632743e-07`
`test_z2 = add(dot(test_a1, W2), B2)`	`1.64589414e-12`
# $A^{(2)} = f(Z^{(2)})$	`1.47751115e-13`
`test_a2 = Relu(test_z2)`	`2.11127085e-19`
# $Z^{(3)} = X \cdot W^{(3)} + B^{(3)}$	**`9.99999166e-01`**
`test_z3 = add(dot(test_a2, W3), B3)`	`1.92158459e-11`
# $A^{(2)} = f(Z^{(2)})$	`1.27989965e-08`
`test_a3 = Softmax(test_z3)`	`]`
`print(test_a3)`	

Table 8. Calculation script and output.

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.

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 know there are a ton of them behind the NNs! To name a few, there are functions, derivatives, partial derivatives, vectors, matrices, exponential moving average, etc. These are all interconnected and work together to solve complex real-world problems. It is the multi-disciplinary joint efforts involving math, biology and math made it possible for NNs to work.

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 (than 28 by 28 pixels).

Today the technology of AI is transforming our world at record pace in areas like high resolution face-recognition, real-time translation, AI 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, math is power, and math is 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

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

```
Coefficients:
        $\alpha$: step size
        $\beta_1$: first moment, $\in [0,1)$
        $\beta_2$: second moment, $\in [0,1)$
        $\epsilon$:  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 $\theta$
    $\theta_0$: 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 $\boldsymbol{\theta_t}$ 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)
        $\widehat{m_t} = \frac{m_t}{(1-\beta_1^t)}$   (Compute bias-corrected first moment estimate)
        $\widehat{v_t} = \frac{v_t}{(1-\beta_2^t)}$   (Compute bias-corrected second raw moment estimate)
        $\theta_t = \theta_{t-1} - \alpha \cdot \frac{\widehat{m_t}}{(\sqrt{\widehat{v_t}}+\epsilon)}$   (Update parameters)
    end while
    return $\boldsymbol{\theta_t}$ (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. I will measure its loss vs epoch data in the experiment section.

APPDENDIX-B: Theoretic calculation of gradients of my NN

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

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

$where$

$$l \in [1, L], L\ being\ the\ output\ layer's\ layer\ number$$

✧ 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:

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

✧ 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:

$$\because f\left(z_i^{(l)}\right) = \begin{cases} 0, & z_i^{(l)} < 0 \\ z_i^{(l)}, & z_i^{(l)} \geq 0 \end{cases}$$

$$\therefore f'\left(z_i^{(l)}\right) = \begin{cases} 1\ when\ z_i^{(l)} \geq 0; \\ 0\ when\ z_i^{(l)} < 0; \end{cases} \qquad \dots (18b)$$

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 $\frac{\partial a_i{}^{(3)}}{\partial z_j{}^{(3)}}, i, j \in [1,10]$

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

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

- When $i = j$,

$$\because \frac{\partial a_i{}^{(3)}}{\partial z_j{}^{(3)}} = \frac{\partial \frac{e^{z_i^{(3)}}}{\Sigma}}{\partial z_j{}^{(3)}} = \frac{e^{z_i^{(3)}} \Sigma - e^{z_j^{(3)}} e^{z_i^{(3)}}}{\Sigma^2} = \frac{e^{z_i^{(3)}}}{\Sigma}\left(1 - \frac{e^{z_j^{(3)}}}{\Sigma}\right)$$

$$\therefore \frac{\partial a_i{}^{(3)}}{\partial z_j{}^{(3)}} = f(z_i^{(3)})(1 - f(z_j^{(3)}))$$

- When i$\neq j$,

$$\because \frac{\partial a_i{}^{(3)}}{\partial z_j{}^{(3)}} = \frac{\partial \frac{e^{z_i^{(3)}}}{\Sigma}}{\partial z_j{}^{(3)}} = \frac{0 - e^{z_j^{(3)}} e^{z_i^{(3)}}}{\Sigma^2} = -\frac{e^{z_i^{(3)}}}{\Sigma}\frac{e^{z_j^{(3)}}}{\Sigma}$$

$$\therefore \frac{\partial a_i{}^{(3)}}{\partial z_j{}^{(3)}} = -f(z_i^{(3)})f(z_j^{(3)})$$

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

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

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):

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

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

$where\ n\ is\ the\ node\ number\ of\ a\ layer$;

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:

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

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

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

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

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).

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

$$\frac{\partial C}{\partial z_j^{(3)}} = -\frac{\partial}{\partial z_j^{(3)}} \sum_{i=1}^{10} \left(y_i \log_e a_i^{(3)}\right) = -\sum_{i=1}^{10} y_i \cdot \frac{\partial}{\partial z_j^{(3)}} \log_e a_i^{(3)}$$

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

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

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

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

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

$$\frac{\partial C}{\partial z_j^{(3)}} = a_j{}^{(3)} - y_j \qquad \text{... (21)}$$

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: Calculation of layer H2's and layer H1's gradients and updates for weights and biases.

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

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

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

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:

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

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

Where $\odot$ denotes Hadamard product operand, $\cdot$ denotes dot product operand.

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 $W^{(2)}, B^{(2)}$	$\begin{matrix} -0.043 & 0.008 & 0.025 & 0.049 & \cdots & -0.057 \\ 0.042 & 0.074 & 0.063 & -0.012 & \cdots & 0.079 \\ 0.044 & 0.077 & 0.014 & -0.047 & \cdots & -0.011 \\ -0.009 & -0.028 & -0.021 & 0.028 & \cdots & 0.049 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -0.028 & -0.074 & 0.072 & -0.030 & \cdots & -0.010 \end{matrix}$	$\begin{matrix} 0.000 \\ 0.000 \\ 0.000 \\ 0.000 \\ \vdots \\ 0.000 \end{matrix}$
∇C (gradient)	$\begin{matrix} -0.004 & -0.000 & 0.006 & 0.000 & \cdots & 0.000 \\ -0.018 & -0.000 & 0.026 & 0.000 & \cdots & 0.000 \\ -0.000 & -0.000 & 0.000 & 0.000 & \cdots & 0.006 \\ -0.016 & -0.000 & 0.023 & 0.000 & \cdots & 0.000 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -0.031 & -0.000 & 0.044 & 0.000 & \cdots & 0.000 \end{matrix}$	$\begin{matrix} -0.067 \\ -0.000 \\ 0.093 \\ 0.000 \\ \vdots \\ 0.000 \end{matrix}$
$\eta \odot \nabla C$ (delta: learning ratio$\odot$*gradient*)	$\begin{matrix} -7.441e-4 & 0.000 & 7.441e-4 & 0.000 & \cdots & 0.000 \\ -7.441e-4 & 0.000 & 7.441e-4 & 0.000 & \cdots & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ -7.441e-4 & 0.000 & 7.441e-4 & 0.000 & \cdots & 0.000 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -7.441e-4 & 0.000 & 7.441e-4 & 0.000 & \cdots & 0.000 \end{matrix}$	$\begin{matrix} -7.441e-4 \\ 0.000 \\ 7.441e-4 \\ 0.000 \\ \vdots \\ 0.000 \end{matrix}$
New	$\begin{matrix} -0.095 & 0.017 & 0.055 & 0.106 & \cdots & 0.158 \\ 0.153 & -0.029 & \mathbf{0.044} & -0.152 & \cdots & -0.125 \\ 0.092 & 0.161 & 0.138 & -0.025 & \cdots & -0.169 \\ 0.069 & -0.023 & -0.125 & -0.166 & \cdots & 0.172 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0.027} & 0.080 & \mathbf{0.097} & -0.054 & \cdots & 0.145 \end{matrix}$	$\begin{matrix} \mathbf{7.441e-4} \\ 0.000 \\ \mathbf{-7.441e-4} \\ 0.000 \\ \vdots \\ 0.000 \end{matrix}$

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

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

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

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

Similarly, I translated them to vectorized forms,

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

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

Where $\odot$ denotes Hadamard product operand, $\cdot$ denotes dot product operand.

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 $W^{(2)}, B^{(2)}$	$\begin{matrix} -0.043 & 0.008 & 0.025 & 0.049 & \cdots & -0.057 \\ 0.042 & 0.074 & 0.063 & -0.012 & \cdots & 0.079 \\ 0.044 & 0.077 & 0.014 & -0.047 & \cdots & -0.011 \\ -0.009 & -0.028 & -0.021 & 0.028 & \cdots & 0.049 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -0.028 & -0.074 & 0.072 & -0.030 & \cdots & -0.010 \end{matrix}$	$\begin{matrix} 0.000 \\ 0.000 \\ 0.000 \\ 0.000 \\ \vdots \\ 0.000 \end{matrix}$
∇C (gradient)	$\begin{matrix} -0.000 & -0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ -0.000 & -0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ -0.000 & -0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ -0.000 & -0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -0.0 & -0.000 & 0.000 & 0.000 & \cdots & 0.000 \end{matrix}$	$\begin{matrix} -\ 0.072 \\ -\ 0.025 \\ 0.000 \\ -0.033 \\ \vdots \\ 0.000 \end{matrix}$
$\eta \odot \nabla C$ (delta: learning ratio$\odot gradient$)	$\begin{matrix} 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0.000 & 0.000 & 0.000 & 0.000 & \cdots & 0.000 \end{matrix}$	$\begin{matrix} -7.441e-4 \\ -7.441e-4 \\ 0.000 \\ -7.441e-4 \\ \vdots \\ 0.000 \end{matrix}$
New	$\begin{matrix} -0.043 & 0.008 & 0.025 & 0.049 & \cdots & -0.057 \\ 0.042 & 0.074 & 0.063 & -0.012 & \cdots & 0.079 \\ 0.044 & 0.077 & 0.014 & -0.047 & \cdots & -0.011 \\ -0.009 & -0.028 & -0.021 & 0.028 & \cdots & 0.049 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -0.028 & -0.074 & 0.072 & -0.030 & \cdots & -0.010 \end{matrix}$	$\begin{matrix} \mathbf{7.441e-4} \\ \mathbf{7.441e-4} \\ 0.000 \\ \mathbf{7.441e-4} \\ \vdots \\ 7.441e-4 \end{matrix}$

Table A-2. Calculation of ∇C, η, and the new $W^{(1)}$ and $B^{(1)}$. The weight and bias values that are updated are in red color.

One thing to notice, is that the updates for $W^{(1)}$ are mostly 0s for the displayed parts here. This makes sense because X vector has 0s for such areas and ∇C depends on X.