

Structure of a Neural Network:

The most basic structure of a NN is the node. A node is a scalar value, i.e. a single number. Examples of scalars are 1.618, 3.14, 2.71, etc.

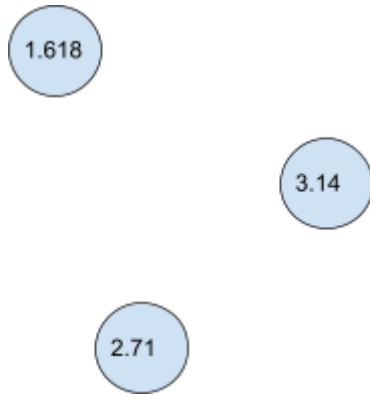


Fig 1: Visual Representation of Nodes

The next structure of a NN is a layer. A layer is a vector of nodes. Vectors can be thought of as a list of numbers. Layers have sizes that represent how many nodes are inside of them. For example, a layer of size 144 has 144 nodes in it. Layers in an MLP fall into three categories. The input layer is where you input either your training data or the data you want to process. After the input layer are the hidden layers which take the values of the input layers and process them. After the hidden layers is the output layer which is the layer that you read to get the results of the NN.

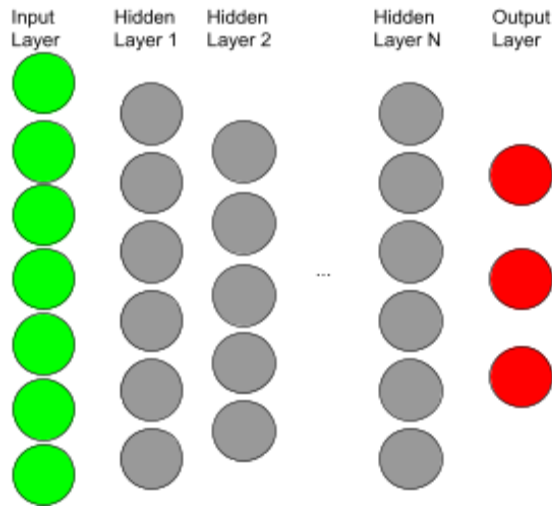


Fig 2: Visual Representation of Layers

The reason for the layered structure is that each layer recognizes some features of the data which influences the next layer which notices other aspects of the data and so on. The connections between the layers is governed by the equation $Y = WX + b$, where Y is the output of the layer, W is the weight matrix of the layer, X is the input from the previous layer or raw data, and b is the bias vector of the layer.

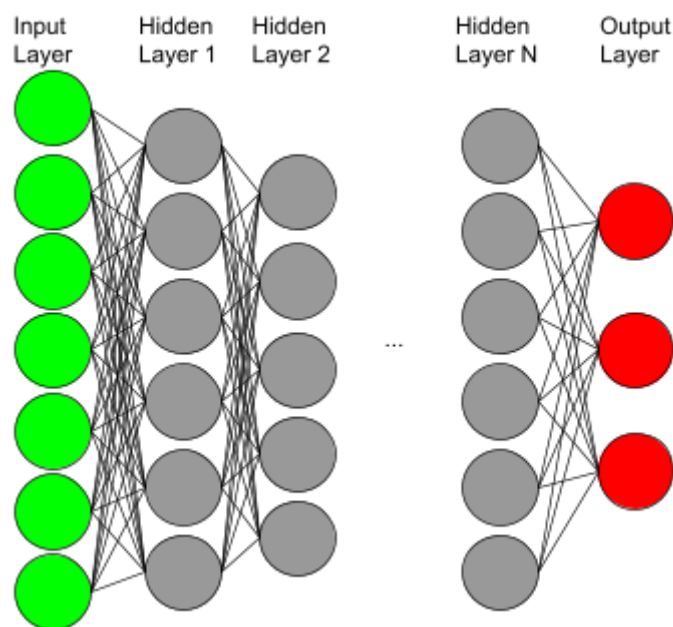
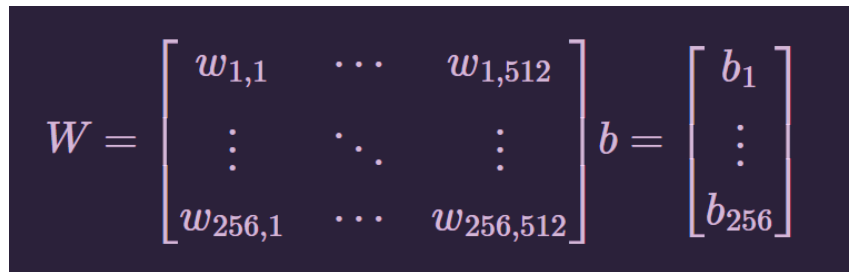


Fig 3: Visual Representation of Neural Network

The weights and biases of the layer is what makes learning possible for NNs. By changing the weights and biases, the output of the NN can be altered to fit the dataset provided. The shape of the weight matrix is the number of nodes in the previous layer by the number of nodes in the current layer. For example if the previous layer's size is 512 and the current layer's size is 256, the shape of the matrix is 256 by 512. For reference, a matrix is essentially a 2 dimensional vector where each value in the matrix corresponds with a row and column. The shape of the bias vector is the size of the current layer. Shape meaning the dimension of the vector or matrix. The larger the layer size the more features it can identify in the data. But, larger layer sizes means there's a greater chance for overfitting.



$$W = \begin{bmatrix} w_{1,1} & \cdots & w_{1,512} \\ \vdots & \ddots & \vdots \\ w_{256,1} & \cdots & w_{256,512} \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_{256} \end{bmatrix}$$

Fig 4: Visualization of Weight (W) matrix and Bias (b) vector

In many modern NNs there are activation functions after each layer. An activation function turns the linear output from the equation $Y = WX + b$ into a nonlinear output. The reason for the activation functions is to make the NN generalize better, which is to say the NN can handle more complex problems better, because nonlinear functions often represent real world problems better than a linear function. Common activation functions are the sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU) functions. What the sigmoid function does is restrict the output of the layer between 0 and 1 according to the equation $\sigma(x) = \frac{1}{1+e^{-x}}$.

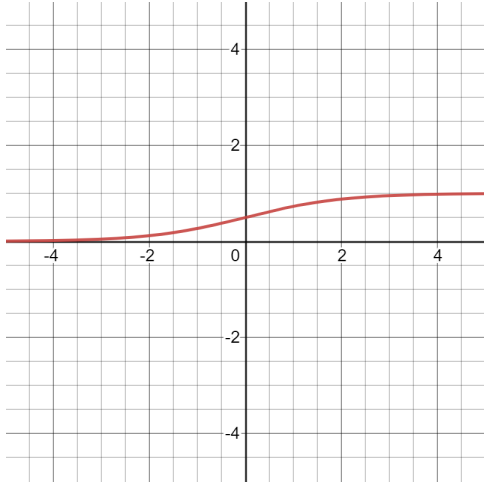


Fig 5: Graph of Sigmoid Function

The hyperbolic tangent activation function restricts the output of the layer between -1 and 1

which is defined by the equation $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$.

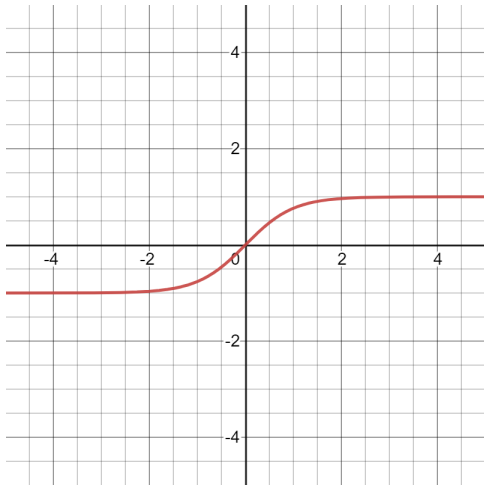


Fig 6: Graph of Hyperbolic Tangent Function

The ReLU activation function follows the equation $\text{ReLU}(x) = \max(x, 0)$.

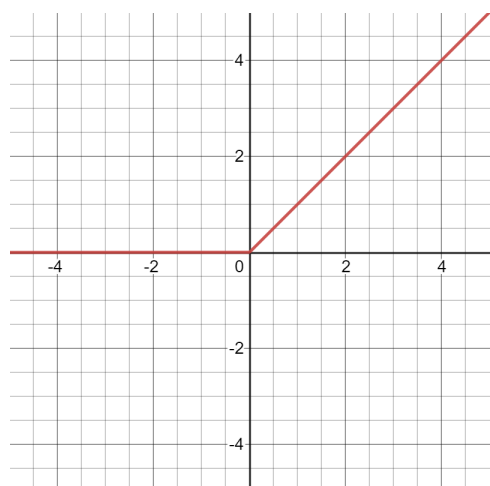


Fig 7: Graph of ReLU Function

The reason why one would choose one activation function over another is where the activation functions are located in the neural network and what format their data is in. ReLU functions are often used in hidden layers because they minimize what's known as gradient explosion and gradient vanishing, more on those later. Sigmoid and the hyperbolic tangent function are often used on the output layer to fit specific types of data that can either be scaled from 0 to 1 or -1 to 1 respectively.

Training a Neural Network

Before training a NN, the weights and biases of the NN have to be initialized. The bias vector is the simplest to initialize since standard practice is to initialize all bias values to zero. For weight initialization, two popular methods are the Xavier and Kaiming initialization schemes. Xavier initialization aims to prevent the vanishing gradients problem in machine learning. To put it simply, the vanishing gradient problem occurs when calculating the gradient of the NN, some of the gradients of the network go to zero. When a gradient goes to zero no training occurs to the weight or bias associated with the gradient which can cause a cascading

effect on the entire network. The two steps that are involved in Xavier initialization are initializing the weights according to a Gaussian distribution and then scale those weights in proportion to the number of inputs to the layer which is represented by the equation

$$W^{(L)} := W^{(L)} \cdot \sqrt{\frac{1}{m^{(L-1)}}}; W_{ij}^{(L)} \sim N(\mu = 0, \sigma^2 = 0.01), \text{ where } W \text{ is the weight matrix, } L \text{ is the}$$

layer index, m is the output dimension of the current layer, N is the Gaussian distribution function, μ is the mean of the distribution, and σ^2 is the variance of the distribution. Kaiming initialization works almost exactly like Xavier initialization, but the scaling is a little different.

The equation is instead $W^{(L)} := W^{(L)} \cdot \sqrt{\frac{2}{m^{(L-1)}}}; W_{ij}^{(L)} \sim N(\mu = 0, \sigma^2 = 0.01)$ due to the activations no longer being centered around zero. After we initialize our weights then the NN is ready to be trained.

The goal of training is to reach the minimum point of a function. In real life, we don't know what the graph looks like and where the minimum point is. What we do know is the cost of the given parameters and the direction where the graph is steepest. Back to Fig. 8, imagine a ball on the graph. The ball will follow the path where the graph is steepest until it reaches the lowest point it can find. We can imagine the ball's path as tiny steps in a given direction. The ball knows where the graph is steepest and takes small steps in the steepest direction. The ball continues to take tiny steps until it reaches a point on the graph where nothing curves up or down. This is the essence of how gradient descent works. We find the gradient, the steepness of the graph, of the NN and take small steps to minimize the cost function of the network.

To find the gradient of the network, researchers have developed the backpropagation algorithm. Every weight and bias has an effect on the NN, but not every weight and bias has the same effect on the NN. Some weights and biases will have a stronger effect than other weights

and biases. The backpropagation algorithm figures out which weights and biases will have the most influence on the network. How the backpropagation algorithm works is by taking the gradient of the network given each data point in the dataset.

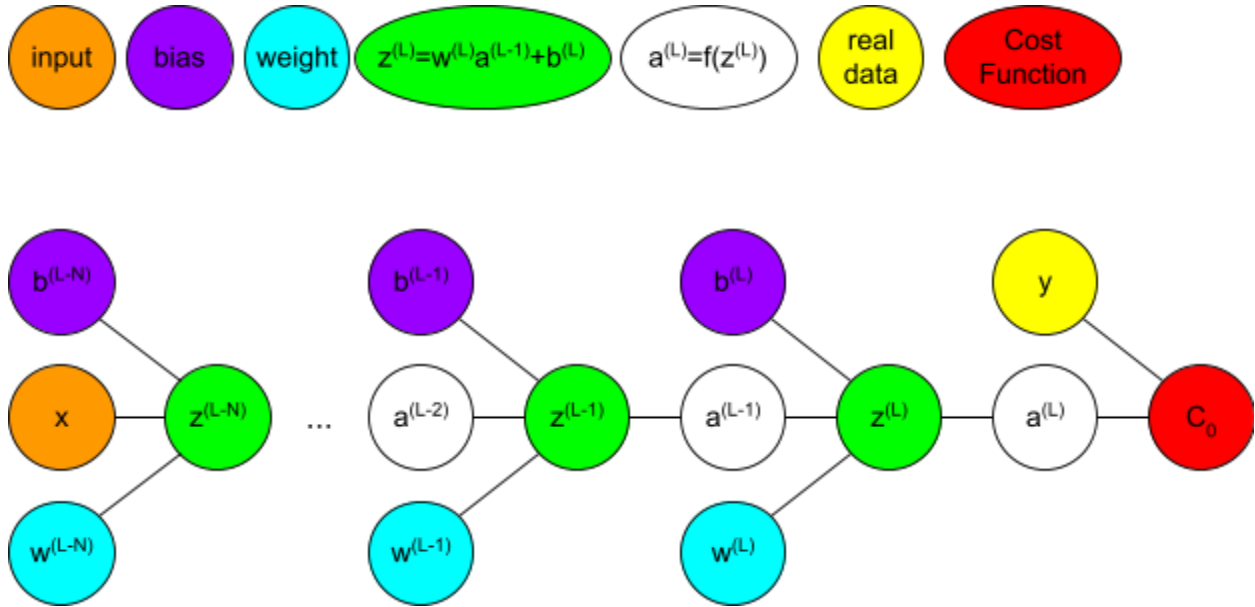


Fig 9: Visualization of a Computational Graph

The variable z is the linear output of the weights, biases, and previous inputs of the network. The variable a is the output after the variable z has been passed through an activation function. Using the equation found using the computation graph and the chain rule, we can construct the exact gradient of each respective term. The gradient descent algorithm is

represented by the equation $P_{new} = P - \alpha \frac{\partial C_0}{\partial P}$, where P is our parameter we can change e.g.

weight matrix or bias vector, α is our learning rate, and C_0 is our cost function. We then repeat the backpropagation and gradient descent algorithm until the cost function is close to or exactly equal to zero.

Implementing Neural Networks Using PEMDAS

I chose to use python to implement a NN from scratch using PEMDAS. The first step for NN training is loading and preparing the data. For this network, I will use the MNIST handwritten digits dataset. The MNIST dataset has 60000 handwritten digits for training and 10000 handwritten digits for testing. The digits are 28 pixels wide by 28 pixels tall which means we need to process 784 pixels. Each pixel has a value ranging from 0 to 255.

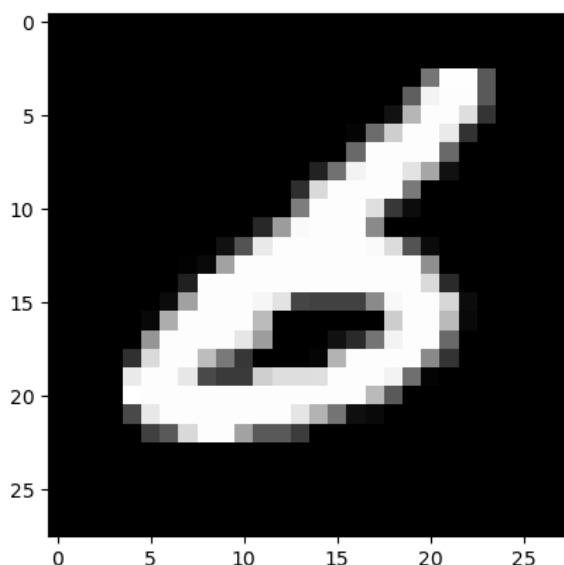


Fig 10: A 28 pixel by 28 pixel image of the number 6

A value of 0 is the darkest value in the image and 255 has the brightest value in the image. But in practice, we need to rescale the pixels in order to avoid the gradients of the network exploding. Exploding gradients are when the gradient values are so high the computer can't process it because it will run out of memory. We divide each pixel value by 255 so that each pixel is rescaled between 0 and 1. For our network to process the image, we need to change the shape of the image from 28x28 to 784x1. We can do both the rescaling and resizing in a single line of code as seen in Fig. 11.


```
x = train_X[idx].reshape((-1, 1))/255
y = train_y[idx]
```

Fig 11: Code for Reshaping and Rescaling the Training Data

The variable `y` is to track which label associates with each image. The variable `idx` is to track the index of the image in the dataset i.e. images 0, 1, 2, 3, etc. The next step is to define the functions we are going to use for the neural network.

```
def linear(x, w, b):
    return w*x+b

def ReLU(Z):
    return np.where(Z > 0, Z, 0)

def sigmoid(Z):
    return 1/((1+np.exp(-Z)))

def ReLUPrime(Z):
    return np.where(Z > 0, 1, 0)

def sigmoidPrime(Z):
    return np.exp(-Z)/((1+np.exp(-Z)**2))
```

Fig 12: Sigmoid and ReLU Functions Along With Their Derivatives

`ReLUPrime` and `sigmoidPrime` are the derivatives of the `ReLU` and `sigmoid` functions respectively. The next step is to create the weights and biases of our network using the Xavier and Kaiming initialization schemes. Our network's structure has an input layer, two hidden layers, and an output layer. We need weights and biases for the input layer and the two hidden layers. The input layer and the first hidden layer will have the `ReLU` activation function and the second hidden layer will have the `sigmoid` activation function. This is implemented by Fig 13.

```

# Kaiming Initialization
W1 = np.random.randn(16, 28*28) * np.sqrt(2/(28*28))
b1 = np.zeros((16, 1))

W2 = np.random.randn(16, 16) * np.sqrt(2/16)
b2 = np.zeros((16, 1))

# Xavier Initialization
W3 = np.random.randn(10, 16) * np.sqrt(1/16)
b3 = np.zeros((10, 1))

```

Fig 13: Weight and Bias Initialization in Python

The W variables represent the weights of the corresponding layers and the b variables are the biases of each layer. All the biases are set to zero using the `np.zeros()` function. The function `np.random.randn()` creates a matrix with random variables that have been determined by the gaussian distribution function. The variables W1 and W2 are scaled by the value $\sqrt{\frac{2}{784}}$ in accordance with the Kaiming Initialization scheme. W3 is scaled by $\sqrt{\frac{1}{16}}$ in accordance with the Xavier Initialization scheme. Now that we have the weights and biases we can start with the forward calculation of the NN.

```
# Forward Calculation
z1 = linear(X, w1, b1)
a1 = ReLU(z1)
z2 = linear(a1, w2, b2)
a2 = ReLU(z2)
z3 = linear(a2, w3, b3)
a3 = sigmoid(z3)
```

Fig 14: Forward Calculation in Python

The variable `a3` is the final output of our system and can be considered as the system's answer to the given input image. The output is a 10x1 vector where the largest element value should correspond to the digit that was inputted into the network. Since the network has not been trained, it looks more like random noise rather than a clear answer as shown in Fig 15.

Network Output	Desired Output
0.502	0
0.414	0
0.401	0
0.551	0
0.485	1
0.430	0
0.421	0
0.439	0
0.468	0
0.590	0

Fig 15: Representation of an Untrained Network's Output to Desired Output

The desired output is in a format called one-hot encoding in which the desired value is represented by 1 and all the other possible values are zeros. Our goal is to transform the network's output to match the desired output. We will use the squared error cost function to

represent the distance between our network output and the desired output. The squared error cost function is the difference between the network output and the desired output squared which is represented by $C = (a_3 - y)^2$ where y is our desired output and C is the cost. The derivative of the squared error function is represented by the equation $\frac{\partial C}{\partial a_3} = 2(a_3 - y)$ where $\frac{\partial C}{\partial a_3}$ is the change in the cost function given a change in a_3 .

```
# Change in cost given a3
dC_dA3 = 2*(a3-one_hot)

# Jacobian dZ3_dW3 - Change in z3 given W3 is a2
dZ3_dW3 = np.tile(a2, (1, 10)).T
dC_dW3 = dC_dA3*sigmoidPrime(z3)*dZ3_dW3
dC_dB3 = dC_dA3*sigmoidPrime(z3)

# Jacobian dZ2_dW2 - Change in z2 given W2 is a1
dZ2_dW2 = np.tile(a1, (1, 16)).T
dC_dW2 = np.dot(W3.T, dC_dB3)*ReLUPrime(z2)*dZ2_dW2
dC_dB2 = np.dot(W3.T, dC_dB3)*ReLUPrime(z2)

# Jacobian dZ1_dW1 - Change in z1 given W1 is X
dZ1_dW1 = np.tile(X, (1, 16)).T
dC_dW1 = np.dot(W2.T, dC_dB2)*ReLUPrime(z1)*dZ1_dW1
dC_dB1 = np.dot(W2.T, dC_dB2)*ReLUPrime(z1)
```

Fig 16: Finding the Jacobians of the Weights and Biases According to the Cost Function

Let's first analyze the change in the weights and biases in the third layer $\frac{\partial C}{\partial W_3}$ and $\frac{\partial C}{\partial b_3}$.

Referencing the compute graph in Fig 9, we can see that the equation for the cost function given the inputs W_3 and b_3 is $C = (a_3 - y)^2 = (\sigma(z_3) - y)^2 = (\sigma(W_3 a_2 + b_3) - y)^2$ where σ is the sigmoid function. The partial derivative for W_3 and b_3 respectively are therefore

$$\frac{\partial C}{\partial W_3} = \frac{\partial C}{\partial a_3} \frac{\partial a_3}{\partial z_3} \frac{\partial z_3}{\partial W_3} = 2(a_3 - y) \sigma'(z_3) a_2 \text{ and } \frac{\partial C}{\partial b_3} = \frac{\partial C}{\partial a_3} \frac{\partial a_3}{\partial z_3} \frac{\partial z_3}{\partial b_3} = 2(a_3 - y) \sigma'(z_3). \text{ When}$$

we try and implement the equation for $\frac{\partial C}{\partial W_3}$ however, we find that the shape of a_2 can't be

computed with the rest of the equation normally. Instead we have to modify it by taking the

transpose of the `np.tile()` function. The reason why this works is expressed in Fig 17.

$$\begin{aligned}
 W &= \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} & w_{1,5} & w_{1,6} & w_{1,7} & w_{1,8} & w_{1,9} & w_{1,10} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{2,5} & w_{2,6} & w_{2,7} & w_{2,8} & w_{2,9} & w_{2,10} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & w_{3,5} & w_{3,6} & w_{3,7} & w_{3,8} & w_{3,9} & w_{3,10} \\ w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} & w_{4,5} & w_{4,6} & w_{4,7} & w_{4,8} & w_{4,9} & w_{4,10} \\ w_{5,1} & w_{5,2} & w_{5,3} & w_{5,4} & w_{5,5} & w_{5,6} & w_{5,7} & w_{5,8} & w_{5,9} & w_{5,10} \\ w_{6,1} & w_{6,2} & w_{6,3} & w_{6,4} & w_{6,5} & w_{6,6} & w_{6,7} & w_{6,8} & w_{6,9} & w_{6,10} \\ w_{7,1} & w_{7,2} & w_{7,3} & w_{7,4} & w_{7,5} & w_{7,6} & w_{7,7} & w_{7,8} & w_{7,9} & w_{7,10} \\ w_{8,1} & w_{8,2} & w_{8,3} & w_{8,4} & w_{8,5} & w_{8,6} & w_{8,7} & w_{8,8} & w_{8,9} & w_{8,10} \\ w_{9,1} & w_{9,2} & w_{9,3} & w_{9,4} & w_{9,5} & w_{9,6} & w_{9,7} & w_{9,8} & w_{9,9} & w_{9,10} \\ w_{10,1} & w_{10,2} & w_{10,3} & w_{10,4} & w_{10,5} & w_{10,6} & w_{10,7} & w_{10,8} & w_{10,9} & w_{10,10} \end{bmatrix} \quad a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \\ a_9 \\ a_{10} \end{bmatrix} \\
 z = W \cdot a &= \begin{bmatrix} a_1 w_{1,1} + a_2 w_{1,2} + a_3 w_{1,3} + a_4 w_{1,4} + a_5 w_{1,5} + a_6 w_{1,6} + a_7 w_{1,7} + a_8 w_{1,8} + a_9 w_{1,9} + a_{10} w_{1,10} \\ a_1 w_{2,1} + a_2 w_{2,2} + a_3 w_{2,3} + a_4 w_{2,4} + a_5 w_{2,5} + a_6 w_{2,6} + a_7 w_{2,7} + a_8 w_{2,8} + a_9 w_{2,9} + a_{10} w_{2,10} \\ a_1 w_{3,1} + a_2 w_{3,2} + a_3 w_{3,3} + a_4 w_{3,4} + a_5 w_{3,5} + a_6 w_{3,6} + a_7 w_{3,7} + a_8 w_{3,8} + a_9 w_{3,9} + a_{10} w_{3,10} \\ a_1 w_{4,1} + a_2 w_{4,2} + a_3 w_{4,3} + a_4 w_{4,4} + a_5 w_{4,5} + a_6 w_{4,6} + a_7 w_{4,7} + a_8 w_{4,8} + a_9 w_{4,9} + a_{10} w_{4,10} \\ a_1 w_{5,1} + a_2 w_{5,2} + a_3 w_{5,3} + a_4 w_{5,4} + a_5 w_{5,5} + a_6 w_{5,6} + a_7 w_{5,7} + a_8 w_{5,8} + a_9 w_{5,9} + a_{10} w_{5,10} \\ a_1 w_{6,1} + a_2 w_{6,2} + a_3 w_{6,3} + a_4 w_{6,4} + a_5 w_{6,5} + a_6 w_{6,6} + a_7 w_{6,7} + a_8 w_{6,8} + a_9 w_{6,9} + a_{10} w_{6,10} \\ a_1 w_{7,1} + a_2 w_{7,2} + a_3 w_{7,3} + a_4 w_{7,4} + a_5 w_{7,5} + a_6 w_{7,6} + a_7 w_{7,7} + a_8 w_{7,8} + a_9 w_{7,9} + a_{10} w_{7,10} \\ a_1 w_{8,1} + a_2 w_{8,2} + a_3 w_{8,3} + a_4 w_{8,4} + a_5 w_{8,5} + a_6 w_{8,6} + a_7 w_{8,7} + a_8 w_{8,8} + a_9 w_{8,9} + a_{10} w_{8,10} \\ a_1 w_{9,1} + a_2 w_{9,2} + a_3 w_{9,3} + a_4 w_{9,4} + a_5 w_{9,5} + a_6 w_{9,6} + a_7 w_{9,7} + a_8 w_{9,8} + a_9 w_{9,9} + a_{10} w_{9,10} \\ a_1 w_{10,1} + a_2 w_{10,2} + a_3 w_{10,3} + a_4 w_{10,4} + a_5 w_{10,5} + a_6 w_{10,6} + a_7 w_{10,7} + a_8 w_{10,8} + a_9 w_{10,9} + a_{10} w_{10,10} \end{bmatrix}
 \end{aligned}$$

$$\begin{aligned}
\frac{\partial z}{\partial W} &= \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \end{bmatrix} \\
\text{np.tile(a,(1,10))} &\Rightarrow \begin{bmatrix} a_1 & a_1 & a_1 & a_1 & a_1 & a_1 & a_1 & a_1 & a_1 & a_1 \\ a_2 & a_2 & a_2 & a_2 & a_2 & a_2 & a_2 & a_2 & a_2 & a_2 \\ a_3 & a_3 & a_3 & a_3 & a_3 & a_3 & a_3 & a_3 & a_3 & a_3 \\ a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & a_4 \\ a_5 & a_5 & a_5 & a_5 & a_5 & a_5 & a_5 & a_5 & a_5 & a_5 \\ a_6 & a_6 & a_6 & a_6 & a_6 & a_6 & a_6 & a_6 & a_6 & a_6 \\ a_7 & a_7 & a_7 & a_7 & a_7 & a_7 & a_7 & a_7 & a_7 & a_7 \\ a_8 & a_8 & a_8 & a_8 & a_8 & a_8 & a_8 & a_8 & a_8 & a_8 \\ a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 \\ a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} \end{bmatrix} \\
&\quad \therefore \\
\frac{\partial z}{\partial W} &= \begin{bmatrix} a_1 & a_1 & a_1 & a_1 & a_1 & a_1 & a_1 & a_1 & a_1 & a_1 \\ a_2 & a_2 & a_2 & a_2 & a_2 & a_2 & a_2 & a_2 & a_2 & a_2 \\ a_3 & a_3 & a_3 & a_3 & a_3 & a_3 & a_3 & a_3 & a_3 & a_3 \\ a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & a_4 & a_4 \\ a_5 & a_5 & a_5 & a_5 & a_5 & a_5 & a_5 & a_5 & a_5 & a_5 \\ a_6 & a_6 & a_6 & a_6 & a_6 & a_6 & a_6 & a_6 & a_6 & a_6 \\ a_7 & a_7 & a_7 & a_7 & a_7 & a_7 & a_7 & a_7 & a_7 & a_7 \\ a_8 & a_8 & a_8 & a_8 & a_8 & a_8 & a_8 & a_8 & a_8 & a_8 \\ a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 & a_9 \\ a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} & a_{10} \end{bmatrix}^T
\end{aligned}$$

Fig 17: Proof of the np.tile() Function's Transpose is the Same as the Jacobian

For $\frac{\partial C}{\partial w_2}$, $\frac{\partial C}{\partial w_1}$, $\frac{\partial C}{\partial b_2}$, and $\frac{\partial C}{\partial b_1}$ we have to deal with the shape of the previous weights in

our calculation. For example the change in the cost given a change in b_2 would be,

$$\frac{\partial C}{\partial b_2} = \frac{\partial C}{\partial a_3} \frac{\partial a_3}{\partial z_3} \frac{\partial z_3}{\partial a_2} \frac{\partial z_2}{\partial b_2} = 2(a_3 - y)\sigma'(z_3)W_3 \text{ReLU}'(z_2).$$

The problem here is that we can't multiply w_3 by the rest of the equation. What we have to do instead is take the transpose of w_3 ,

move it to the front, and apply the dot product which gives us the equation

$$\frac{\partial C}{\partial b_2} = W_3^T \cdot 2(a_3 - y)\sigma'(z_3)\text{ReLU}'(z_2).$$

We can extend this to $\frac{\partial C}{\partial w_1}$ and $\frac{\partial C}{\partial b_1}$ in a similar fashion.

```

# Update
# Layer 3/Output layer update
W3 = W3 - lr*dC_dW3
b3 = b3 - lr*dC_dB3

# Layer 2/Hidden layer 2 update
W2 = W2 - lr*dC_dW2
b2 = b2 - lr*dC_dB2

# Layer 1/Hidden layer 1 update
W1 = W1 - lr*dC_dW1
b1 = b1 - lr*dC_dB1

```

Fig 18: Weight and Bias Updates in Python

The variable `lr` is the learning rate, which in this case is equal to 0.001. Now that we can train our model we need to test how effective it is. How I tested this model is by counting how many examples in the testing dataset it got right and dividing it by the total number of test images in the dataset. The first time I ran the program, I trained the network on each of the training images one time. This is known as running the network for one epoch. The starting accuracy is 9.79% and after one epoch it improved to 89.99% as shown in Fig 19.

```

100%|██████████|
Accuracy: 9.79%
100%|██████████|
100%|██████████|
Accuracy: 89.99%

```

Fig 19: Results of Neural Network After One Epoch

I then tested 100 epochs in which the starting accuracy is 9.53% and it improved to 95.14%. 100 epochs means that it has trained on each image in the dataset 100 times.

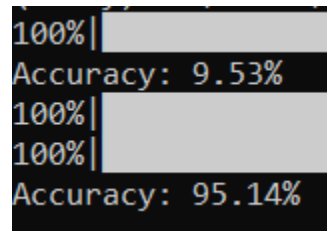


Fig 20: Results of Neural Network After 100 Epochs

Bibliography

- Babs, Temi. "Multi-Layer Perceptron for Beginners." *Medium*, Medium, 27 June 2018, medium.com/@temi.ayo.babs/multi-layer-perceptron-for-beginners-6aee246c6a03.
- . "The Mathematics of Neural Networks." *Coinmonks*, Medium, 14 July 2018, medium.com/coinmonks/the-mathematics-of-neural-network-60a112dd3e05.
- "Backpropagation calculus | Chapter 4, Deep learning", *Youtube*, uploaded by 3Blue1Brown, 3 November 2017, <https://www.youtube.com/watch?v=tIeHLnjs5U8>.
- "Backpropagation Details Pt. 1: Optimizing 3 parameters simultaneously.", *YouTube*, uploaded by StatQuest with Josh Starmer, 1 Nov. 2020, <https://www.youtube.com/watch?v=iyn2zdALii8>
- "Backpropagation Details Pt. 2: Going bonkers with The Chain Rule.", *YouTube*, uploaded by StatQuest with Josh Starmer, 1 Nov. 2020, <https://www.youtube.com/watch?v=GKZoOHXGcLo>
- "But what is a neural network? | Chapter 1, Deep learning", *Youtube*, uploaded by 3Blue1Brown, 5 October 2017, <https://www.youtube.com/watch?v=aircAruvnKk>.
- "Deep Learning Frameworks: Computation Graphs", *YouTube*, uploaded by Jordan Boyd-Graber, 26 Aug. 2018, <https://www.youtube.com/watch?v=FJQl0ujTAGw>
- Deisenroth, Marc Peter, et al. *Mathematics for Machine Learning*. Cambridge University Press, 2020, mml-book.github.io/book/mml-book.pdf.
- "Gradient descent, how neural networks learn | Chapter 2, Deep learning", *Youtube*, uploaded by 3Blue1Brown, 16 October 2017, https://www.youtube.com/watch?v=IZwE_FHWa-w.
- Katanforoosh & Kunin, "Initializing neural networks", *deeplearning.ai*, 2019, <https://www.deeplearning.ai/ai-notes/initialization/>.

- “L11.6 Xavier Glorot and Kaiming He Initialization”, *YouTube*, uploaded by Sebastian Raschka, March 2021, <https://www.youtube.com/watch?v=ScWTYHQra5E>
- LeCun, Yann, and Cortes, Corinna, and Burges, CJ “MNIST Handwritten Digit Database.” *ATT Labs*, vol. 2, 2010, <http://yann.lecun.com/exdb/mnist>.
- Metz, Cade. *Genius Makers: The Mavericks Who Brought AI to Google, Facebook, and the World*. Penguin Random House LLC, 2021.
- “Neural Networks Demystified [Part 4: Backpropagation]”. *YouTube*, uploaded by Welch Labs, 5 Dec 2014, <https://www.youtube.com/watch?v=GlcnxUlrtk>.
- Ng, Andrew. “Computation Graph (C1W2L07)”, *YouTube*, uploaded by DeepLearningAI, 25 Aug. 2017, <https://www.youtube.com/watch?v=hCP1vGoCdYU>.
- . “Derivatives With Computation Graphs (C1W2L08)”, *YouTube*, uploaded by DeepLearningAI, 25 Aug. 2017, <https://www.youtube.com/watch?v=nJyUyKN-XBQ>.
- . “Lecture 0603 Model selection and training/validation/test sets”, *YouTube*, uploaded by Blitz Kim, 26 Feb. 2017, <https://www.youtube.com/watch?v=MyBSkmUeIEs>.
- Tateisi, Jukan. "Understand Kaiming Initialization and Implementation Detail in PyTorch." *Towards Data Science*, Medium, towardsdatascience.com/understand-kaiming-initialization-and-implementation-detail-in-pytorch-f7aa967e9138.
- Thomas, Garrett. *Mathematics for Machine Learning*. Department of Electrical Engineering and Computer Sciences University of California, Berkeley, 2018, [gwtthomas.github.io/docs/math4ml.pdf](https://github.com/gwtthomas/math4ml).

Wang, Chi-Feng. "The Vanishing Gradient Problem: The Problem, Its Causes, Its Significance, and Its Solutions." *Towards Data Science*, Medium, 8 Jan. 2019,

towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484.

"What is Automatic Differentiation?", *YouTube*, uploaded by Ari Seff, 31 Jul. 2020,

https://www.youtube.com/watch?v=wG_nF1awSSY.

"What is backpropagation really doing? | Chapter 3, Deep learning", *Youtube*, uploaded by

3Blue1Brown, 3 November 2017, <https://www.youtube.com/watch?v=Ilg3gGewQ5U>.