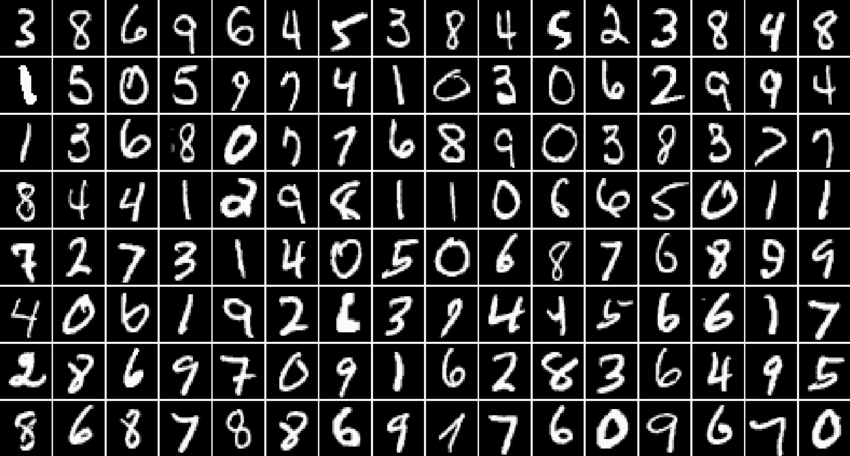
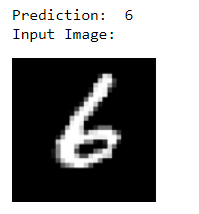
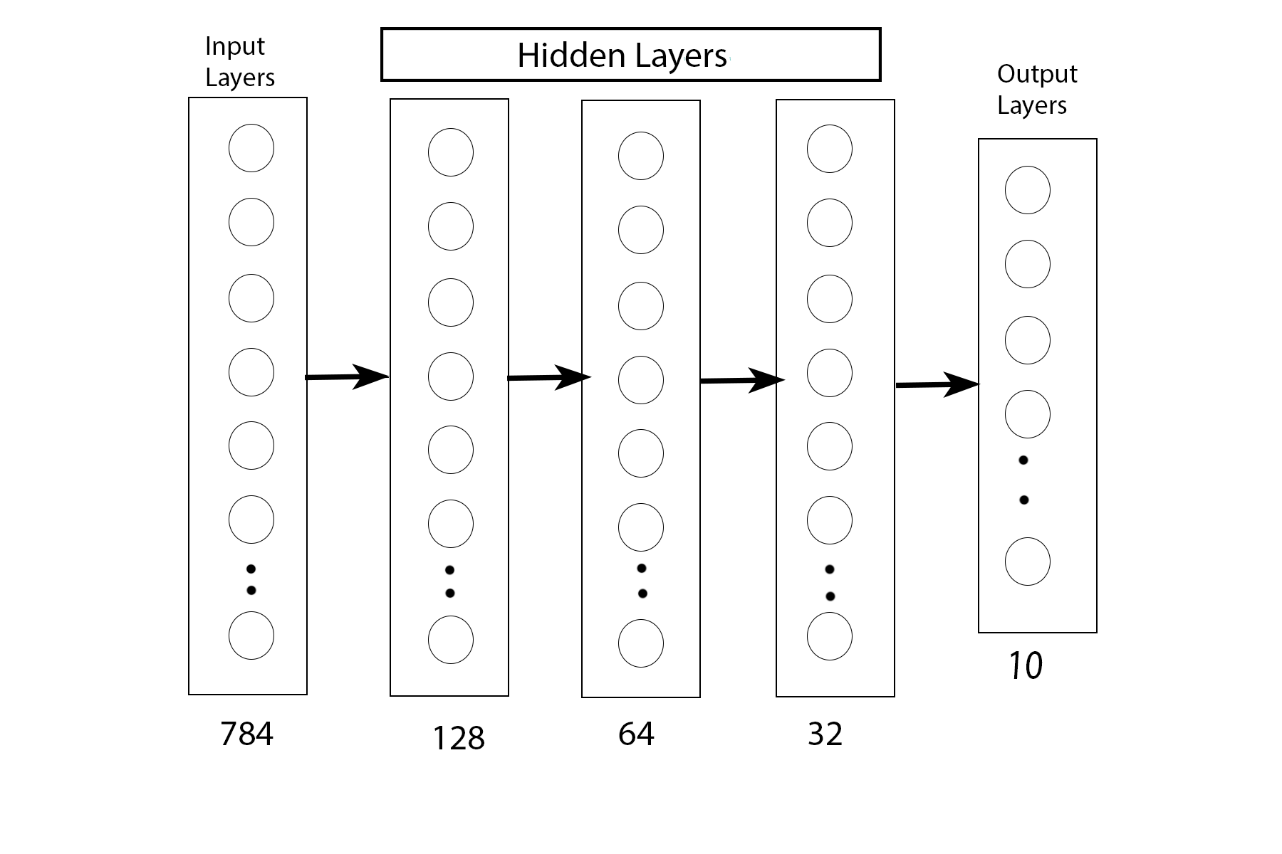
In this article, we are going to demonstrate how to implement a neural network from scratch by building a digit recognizer using MNIST dataset. MNIST dataset is a large dataset of handwritten digits of dimensions 28x28. Below is a sample taken form MNIST dataset.



****After the model is developed and trained, the model will be able to predict handwritten digits from 0-9 from image provided by us. The output will be as below:

**Model Architecture**

****

As one input in MNIST dataset is of shape 28x28(784) pixels, we take 784 inputs in the input layers. As we have 10 digits (0-9), this is a multiclass classification task, so we have 10 outputs in the output layers. The overall architecture is [784,128,64,32,10]. Relu Activation fuction is used in the hidden layers, and softmax activation function is used in output layer as it is a classification task.

**Libraries**

Let us implement all the required components of the neural network. The libraries used are numpy to perform mathematical operations, matplotlib will be used to plot the graph of the costs and pillow will be used to import the image from user provided path.

import numpy as np

import PIL

import matplotlib.pyplot as plt

**Initialize Weights and Biases**

Let us define a function to initialize the all the parameters i.e. weights and biases. The weights are initialized using He approximation(i.e W[l] = np.random.randn(shape)\*np.sqrt(2/n[l-1] , l is the layer in NN, and n is number of nodes in the layer) and biases are initialized to 0.

def initialize\_parameters\_deep(layer\_dims):

np.random.seed(1)

parameters = {}

L = len(layer\_dims)

for l in range(1,L):

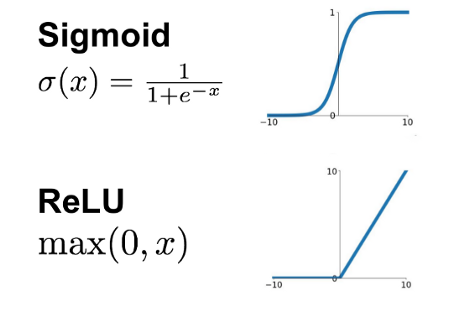
parameters['W' + str(l)] = np.random.randn(layer\_dims[l], layer\_dims[l-1]) \* np.sqrt(2/layer\_dims[l-1])

parameters["b"+str(l)]=np.zeros((layer\_dims[l],1))

return parameters

**Activation Functions**

The activation functions we will be using are softmax and relu.



Let us the define relu and softmax along with their derivatives.

**Relu**

def relu(x):

x = np.maximum(0,x)

return x

**Relu\_Prime**

def relu\_prime(x):

x[x<=0] = 0

x[x>0] = 1

return x

**Softmax**

def softmax(x):

ans = np.exp(x)/np.sum(np.exp(x),axis=0)

return ans

**Softmax\_Prime**

def softmax\_prime(x):

ans = softmax(x)\*(1-softmax(x))

return ans

**Forward Propagation**

Let us define the helper functions required for the forward propagation task

Let us define a linear\_forward layer function with calculates z = w\*a+b in a layer of neural network.

def linear\_forward(A,W,b):

Z=W.dot(A)+b

cache = (A, W ,b)

return Z,cache

Here A is the activations from previous layer, W is weight numpy array of size (size of current layer,size of previous layer), b is bias numpy array of size (size of current layer, 1) and cache is tuple of (A,W,b) which will be used during backpropagation

Now we will implement deep\_layer which will calculate the activations according to the defined activation function as:

def deep\_layer(A,W,b,activation):

Z,linear\_cache = linear\_forward(A,W,b)

if activation == 'softmax':

A = softmax(Z)

activation\_cache = Z

elif activation == 'relu':

A = relu(Z)

activation\_cache = Z

cache = (linear\_cache,activation\_cache)

return A,cache

Here cache contains linear\_cache(A,W,b) from linear\_forward and activation\_cache(Z) , which are going to be used during the back propagation.

Now let us use the above functions and implement forward propagation as:

def forward\_pass(input\_X,parameters):

caches=[]

depth = int(len(parameters)/2) # number of layers in the neural network

A = input\_X

for l in range(1,depth):

A\_prev = A

A,cache = deep\_layer(A\_prev,parameters['W'+str(l)],parameters['b'+str(l)],'relu')

caches.append(cache)

A\_last,cache = deep\_layer(A,parameters['W'+str(depth)],parameters['b'+str(depth)],'softmax')

caches.append(cache)

return A\_last,caches

Here A\_last is the output of the neural network after passing through all layers and caches is the list of all the cache(A,W,b,Z) of each layer of neural network.

Thus, forward propagation part is complete

**Cost Computation**

Since this is a multiclass classification task, the loss function we use is categorical cross-entropy

L

So, the cost is calculated as

- )

Now, lets write a function to compute cost as:

def compute\_cost(AL,Y):

Y= np.reshape(Y,(Y.shape[0],Y.shape[1]))

m = Y.shape[1]

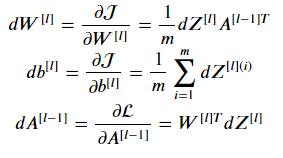
cost =(-1/m)\*(np.multiply(Y,np.log(AL))) #categorical\_cross\_entropy

cost = np.squeeze(cost)

return cost

**Backpropagation**

In backpropagation we have to calculate the gradients so as the update the weight and bias parameters across the network using loss generated by the loss function at output layer. After calculating 𝑑𝑍[𝑙]=∂L/∂𝑍[𝑙], we have to calculate 𝑑𝑊[𝑙],𝑑𝑏[𝑙],𝑑𝐴[𝑙−1] as



Let us write helper function to calculate the gradients as below

def linear\_backward(dZ,cache):

A\_prev , W, b = cache

m = A\_prev.shape[1]

dW = (1/m)\*np.dot(dZ,A\_prev.T)

db = (1/m)\*np.sum(dZ,axis=1,keepdims=True)

dA\_prev = np.dot(W.T,dZ)

return dA\_prev, dW, db

def backward\_activation(dA, cache, activation):

linear\_cache,activation\_cache = cache

if activation == "relu":

dZ = dA\*relu\_prime(activation\_cache)

dA\_prev, dW,db = linear\_backward(dZ,linear\_cache)

elif activation == "softmax":

dZ = dA

dA\_prev, dW,db = linear\_backward(dZ,linear\_cache)

return dA\_prev, dW, db

In case of softmax at the output layer, to avoid the division by zero during gradient calculation in backward pass, we compute dz = A\_last-Y before hand send it to the backward\_activation function as dA\_last\_Z, whereas the dZ for relu is calculated in backward\_activation function itself.

Now, let us implement backward pass using the above helper functions.

def backward\_pass(A\_last,Y,caches):

grads={}

L = len(caches)

m = A\_last.shape[1]

Y = Y.reshape(A\_last.shape)

dA\_last\_Z = A\_last-Y

current\_cache = caches[L-1]

grads["dA"+str(L-1)],grads["dW"+str(L)],grads["db"+str(L)] = backward\_activation(dA\_last\_Z,current\_cache,activation='softmax')

for l in reversed(range(L-1)):

current\_cache = caches[l]

dA\_prev\_temp, dW\_temp, db\_temp = backward\_activation(grads['dA'+str(l+1)],current\_cache,activation="relu")

grads["dA"+str(l)] = dA\_prev\_temp

grads["dW"+str(l+1)] = dW\_temp

grads["db"+str(l+1)]=db\_temp

return grads

Now, after all the gradients have been calculated, we have to updates all the weights and biases across the network as below:

def update\_parameters(parameters, grads, learning\_rate):

depth = len(parameters) // 2

for l in range(depth):

parameters["W"+str(l+1)] = parameters["W"+ str(l+1)]-learning\_rate\*grads['dW'+str(l+1)]

parameters["b"+str(l+1)] = parameters["b" + str(l+1)]-learning\_rate\*grads['db'+str(l+1)]

return parameters

**Compiling the model**

We will be implementing mini batch gradient descent for our model as the dataset is very large, to update the parameters after the training is complete on the particular batch instead of waiting to update the parameters after an epoch like in batch gradient descent.

def mini\_batch\_gradient\_descent(X,Y,layer\_dims=[3,2,1],mini\_batch\_size=8,epochs=100, learning\_rate=0.0075):

np.random.seed(1)

m = X.shape[1]

mini\_batches = []

#shuffling the data

permutation = list(np.random.permutation(X.shape[1]))

X\_shuffled = X[:, permutation]

Y\_shuffled = Y[:, permutation]

num\_of\_complete\_batches = m // mini\_batch\_size

for i in range(num\_of\_complete\_batches):

mini\_batch\_X = X\_shuffled[:,i\*mini\_batch\_size:(i+1)\*mini\_batch\_size]

mini\_batch\_Y = Y\_shuffled[:,i\*mini\_batch\_size:(i+1)\*mini\_batch\_size]

mini\_batch = (mini\_batch\_X,mini\_batch\_Y)

mini\_batches.append(mini\_batch)

#if there is incomplete batch

if m % mini\_batch\_size != 0:

mini\_batch\_X=X\_shuffled[:,num\_of\_complete\_batches\*mini\_batch\_size:num\_of\_complete\_batches\*mini\_batch\_size + (m - mini\_batch\_size\*num\_of\_complete\_batches)]

mini\_batch\_Y=Y\_shuffled[:,num\_of\_complete\_batches\*mini\_batch\_size:num\_of\_complete\_batches\*mini\_batch\_size + (m - mini\_batch\_size\*num\_of\_complete\_batches)]

mini\_batch = (mini\_batch\_X,mini\_batch\_Y)

mini\_batches.append(mini\_batch)

#parameters\_initialize

costs=[]

parameters = initialize\_parameters\_deep(layer\_dims)

for j in range(epochs):

for mini\_batch in mini\_batches:

x\_batch,y\_batch=mini\_batch

# parameters = parameters

A\_Last,caches = forward\_pass(x\_batch,parameters)

cost = np.sum(compute\_cost(A\_Last, y\_batch))/A\_Last.shape[0]

grads = backward\_pass(A\_Last, y\_batch , caches)

parameters = update\_parameters(parameters, grads, learning\_rate)

if j%2 == 0:

print(f'Iteration {j} : {cost}')

costs.append(cost)

return parameters,costs

**Loading The DataSet and Preprocessing**

Now, lets download the MNIST dataset to train our model.

from sklearn.datasets import fetch\_openml

X, y = fetch\_openml('mnist\_784', version=1, return\_X\_y=True)

Let us define one hot encoding function to convert the output into one hot vector.

def one\_hot(y, depth):

one\_hot\_list = np.array([],dtype=int)

for i in range(depth):

temp =np.zeros((1,depth),dtype=int)

temp[0][i]=1

one\_hot\_list =np.append(one\_hot\_list,temp)

one\_hot=np.reshape(one\_hot\_list,(depth,depth))

y = one\_hot[y].T

return y

We will separate the dataset into train set and test set.

train\_x = X[0:60000]/255.0 #normalizing the data

train\_x=train\_x.T

train\_y = y[0:60000]

train\_y = train\_y.astype(int)

train\_y= np.reshape(train\_y,(1,60000))

test\_x = X[60000:X.shape[0]]/255.0 #normalizing the data

test\_x=test\_x.T

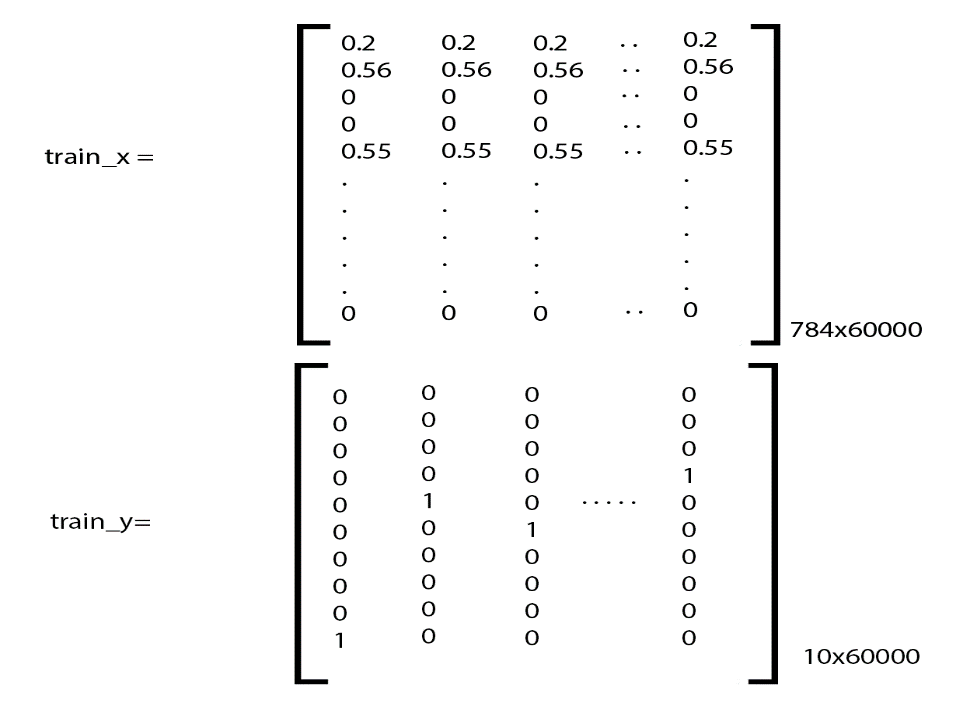
test\_y = y[60000:y.shape[0]]

test\_y = test\_y.astype('int')

Converting the train output into one hot encoding using above function

train\_y = one\_hot(train\_y,10)

The data is now in shape:



Each column denotes one training example in the dataset.

**Running the model:**

The below command runs the trains the model.

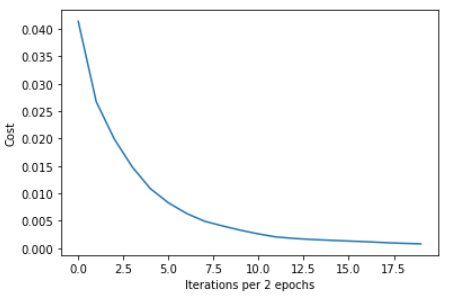
parameters,costs=mini\_batch\_gradient\_descent(train\_x, train\_y,layer\_dims=[784,128,64,32,10],mini\_batch\_size=64,epochs=40)

The graph showing the cost can be plotted as:

plt.plot(costs)

plt.ylabel('Cost')

plt.xlabel('Iterations per 2 epochs')

plt.show()

**Testing the Model**

The following code passes the test\_x through the forward\_pass and predict the resut. The test accuracy was found to be 97.29% in our case.

correct = 0

incorrect=[]

for i in range(test\_y.shape[0]):

last,\_=forward\_pass(np.reshape(test\_x[:,i],(784,1)),parameters)

if np.argmax(last) == test\_y[i]:

correct +=1

else:

incorrect.append(i)

accuracy = correct/test\_y.shape[0]

print("Test Accuracy:",accuracy)

**Trying with your own image**

You can try with your own image by changing the path of the image below. Keep in mind the images in the dataset I used had black background and white digits. If the colors are inverted, the neural network will fail to give correct prediction. You can make the model more accurate by using dataset containing both black and white background. However, in our case I have implemented a function called invert\_image() which inverts the background of image to black if the background of the image is white and give correct prediction.

# Code to convert the image having white background to black

import PIL.ImageOps

def invert\_image(test\_img):

I = np.asarray(test\_img)

zero\_pixels=0

for i in range(I.shape[0]):

for j in range(I.shape[1]):

if I[i,i] == 0:

zero\_pixels+=1

non\_zero\_pixels = 784-zero\_pixels

if non\_zero\_pixels > zero\_pixels:

inverted\_image = PIL.ImageOps.invert(test\_img)

return inverted\_image

else:

return test\_img

# Try with your own image

from PIL import Image

test\_img = Image.open('test\_img.jpg') #path of the image

test\_img\_converted = test\_img.convert('L')

test\_img\_resized = test\_img\_converted.resize((28,28))

test\_img\_array=np.asarray(test\_img\_resized)/255.0

prediction,\_=forward\_pass(np.reshape(test\_img\_array,(784,1)),parameters)

predicted\_num=np.argmax(prediction)

print("Prediction: ",predicted\_num)

print("Input Image: ")

test\_img.resize((128,128))

Some of the examples I have tried on are:

