Machine Intelligence and Learning

ELL409

Assignment 1: Handwritten Digit Recognition using Neural Network

Vivek Muskan

2017MT10755

Overview

Given 7000 MNIST dataset of handwritten digits, we need to design a neural network model using Backpropagation Algorithm. Model should be trained with these 7000 dataset and final testing should be on another 3000 datasets without label. During training and tuning of model, we need to analyze the behavior of the model corresponding to changes in each parameter listed below:

1. Learning Rate
2. Batch Size
3. No. of Epoch
4. Number of neurons
5. Number of Layers
6. Activation Function

Algorithm Implementation

At first, I extracted training and testing data using NumPy library.

In [1]:
```
import random
import matplotlib.pyplot as plt
import numpy as np
train_data = np.genfromtxt(r"C:\Users\Vivek Muskan\Desktop\ML\mnist_train.csv",delimiter = ",")
test_data = np.genfromtxt(r"C:\Users\Vivek Muskan\Desktop\ML\mnist_test.csv",delimiter = ",")
answerfile = "answer.csv"
```

Then I started implementing backpropagation algorithm on single input as a new function. For algorithm I took reference from suggested handout of Satish Kumar and for functions of NumPy I used web.

In [2]:
```
def Back_Prop(X, y, w, b, eta, Num_Layers):
    # Every variable has been stored layer wise
    # Random Initialisation
    del_w = [np.zeros(i.shape) for i in w]
    del_b = [np.zeros(i.shape) for i in b]

    # Step1 -> V[0] = X
    V = [X]
    H = []
    # Step 2 -> Feed Forward
    for i in range(Num_Layers-1):
        h = ( np.add( np.dot(w[i], V[i]), b[i]))
        H.append(h)
        v = sigmoid(h)
        V.append(v)

    # Step 3 -> Calculating Error at Output Layer
    delta = np.multiply(sigmoid_Der(H[-1]), np.subtract(y, V[-1]))
    Err = [delta]

    # Step 4 -> Backward Error propagation
    for i in range(2,Num_Layers):
        temp = np.multiply(sigmoid_Der(H[-i]), np.dot(w[-i+1].transpose(), Err[-i+1]))
        Err.insert(0,temp)

    # Step 5 -> Calculation of changes in weights
    for i in range(Num_Layers-1):
        delta = np.array(Err[i]).reshape(len(Err[i]),1)
        v = np.array(V[i]).reshape(1,len(V[i]))
        del_w[i] = eta*np.matmul(delta, v)

    del_b = Err
    return (del_w, del_b)
```

I made two activation function: sigmoid and tan hyperbolic (although name of function is same). Predict function basically does the feed forwarding of inputs.

In [3]:
```
def sigmoid_scalar(x):
    return (1.0/(1+np.exp(-x)))
sigmoid = np.vectorize(sigmoid_scalar)
```

In [4]:
```
def sigmoid_Der_scalar(x):
    p = sigmoid_scalar(x)
    return p*(1-p)
sigmoid_Der = np.vectorize(sigmoid_Der_scalar)
```

In [5]:
```
def predict(X,w,b):
    for i in range(len(w)):
        h = ( np.add( np.dot(w[i], X), b[i])).transpose()
        X = sigmoid(h)
    return X
```

In [3]:
```
def sigmoid_scalar(x):
    return (np.tanh(x))
sigmoid = np.vectorize(sigmoid_scalar)
```

In [4]:
```
def sigmoid_Der_scalar(x):
    p = sigmoid_scalar(x)
    return (1-p*p)
sigmoid_Der = np.vectorize(sigmoid_Der_scalar)
```

After these, I used Back Propagation function of one input to apply Gradient Descent method on a batch of training data to update the weights and biases of the network. Finally, the function Neural Network initialize all weights and biases, then randomly forms a batch from training data and keep on updating the model for certain numbers of epochs. It gives output by applying trained model on test data.

In [6]:
```
def batch_update(batch, weights, biases, eta, Num_Layers):
    del_w = [np.zeros(i.shape) for i in weights]
    del_b = [np.zeros(i.shape) for i in biases]
    for i in range(len(batch)):
        # Vectorisation of label
        y = np.zeros(10)
        y[int(batch[i][0])] = 1
        dw, db = Back_Prop(batch[i][1:]/255, y, weights, biases, eta, Num_Layers)
        # Update Summation over batch
        del_w = [a+b for a, b in zip(del_w, dw)]
        del_b = [a+b for a, b in zip(del_b, db)]
    temp_w = [w+(1.0/len(batch))*dw for w, dw in zip(weights, del_w)]
    temp_b = [b+(1.0/len(batch))*db for b, db in zip(biases, del_b)]
    return (temp_w, temp_b)
```

In [7]:
```
def Neural_Network(train_data, test_data, Layer_details, epoch_no, batch_size, learning_rate):
    # Weights intialisation
    M = len(Layer_details)
    weights = [np.random.randn(Layer_details[i+1], Layer_details[i])*(0.1) for i in range(M-1)]
    biases = [np.random.randn(1, Layer_details[i+1]).ravel()*(0.1) for i in range(M-1)]

    Accuracy = []
    x_cord = []

    print("Learning Rate = {0},    Epoch no = {1},    Batch Size = {2} ".format(learning_rate, epoch_no, batch_size))
    print("")

    for i in range(epoch_no):
        random.shuffle(train_data)
        Batch_set = [train_data[k:k+batch_size] for k in range(0, len(train_data), batch_size)]
        for batch in Batch_set:
            temp_w, temp_b = batch_update(batch, weights, biases, learning_rate, M)
            weights = temp_w
            biases = temp_b

        count = 0
        for t in test_data:
            temp = predict(t[1:], weights, biases)
            if t[0] == np.where(temp == np.amax(temp))[0][0]:
                count += 1
        acc_percent = 100*count/(len(test_data))
        #print("Epoch {0} / 10 : {1} / 1000 ".format(i,count))
        Accuracy.append(acc_percent)
        x_cord.append(i)

    plt.plot(x_cord, Accuracy, label = 'Layer Details = {0}'.format(Layer_details))
#     result = []
#     k =0
#     for t in test_data:
#         y = predict(t,weights,biases)
```

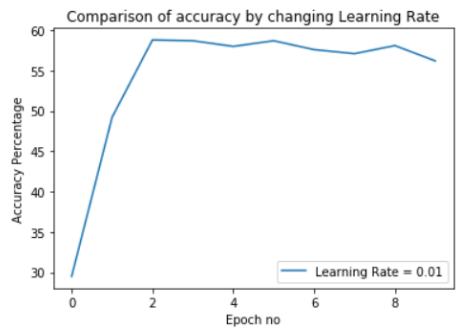
Effect of Changes in Parameter

I have divided the training data in to two parts. I used 6000 data to train my model and 1000 data for cross validation of my model as test data didn't contain any label. Based on its performance on this 1000 data, I have plotted accuracy percentage versus epoch cycle graph for different parameters to compare the result.

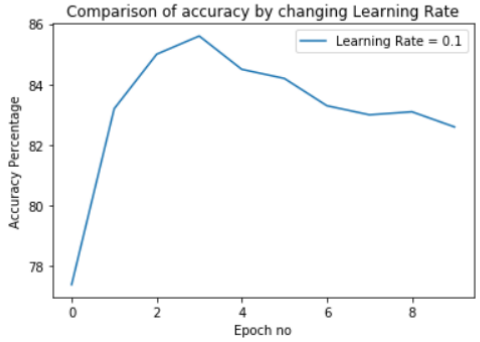
1. Learning Rate

   So, by keeping rest parameters same i.e. Neurons as [784, 22, 10] in each layer, batch size as 1 and for 10 epochs, the graphs show accuracy decreases with either very low learning rate (~0.01) or with very high learning rate (~2). It has a maximum of 89% around 0.5.

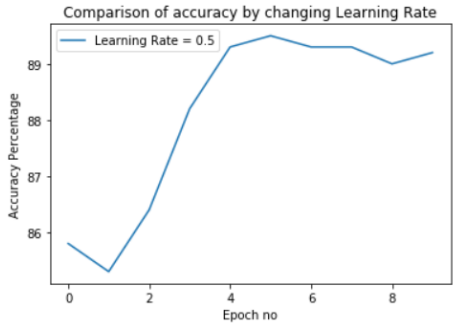
```
Learning Rate = 0.01,    Epoch no = 10,    Batch Size = 1
```



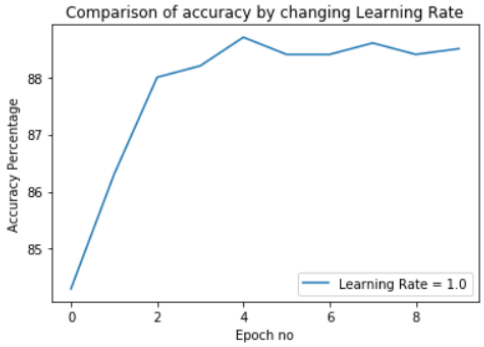

```
Learning Rate = 0.1,    Epoch no = 10,    Batch Size = 1
```



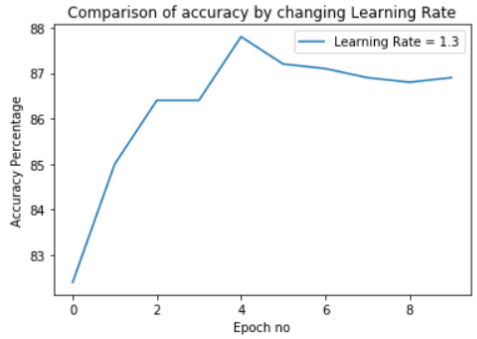
Learning Rate = 0.5, Epoch no = 10, Batch Size = 1



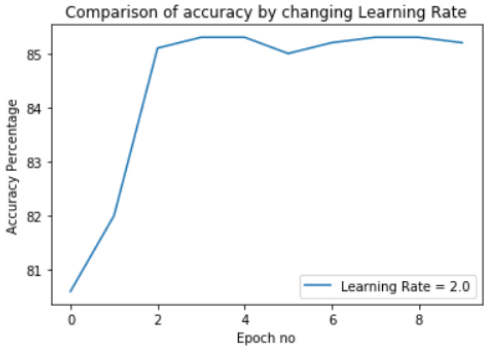
Learning Rate = 1.0, Epoch no = 10, Batch Size = 1



Learning Rate = 1.3, Epoch no = 10, Batch Size = 1



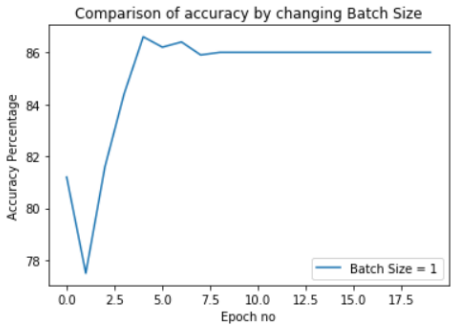
Learning Rate = 2.0, Epoch no = 10, Batch Size = 1



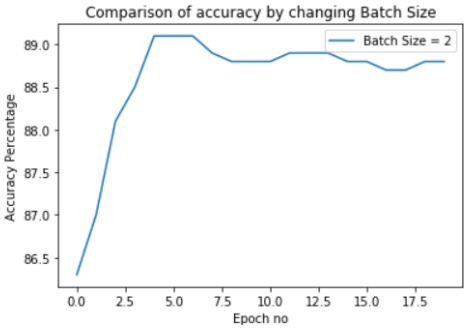
2. Batch Size

Parameters are Neurons as [784, 22, 10], learning rate as 2 and batch size of 1 for 20 epoch cycles. Graph shows that having too less batch size (~1) doesn't give a good estimate of gradient of whole data and hence accuracy is low but having a large batch size (~100) also reduces accuracy because of less no. of updates in the weights and biases. Accuracy is getting maximized (~89%) from 2~5.

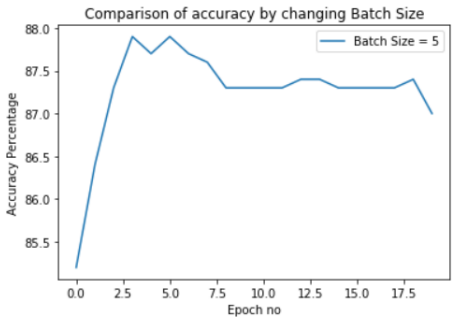
Learning Rate = 2, Epoch no = 20, Batch Size = 1



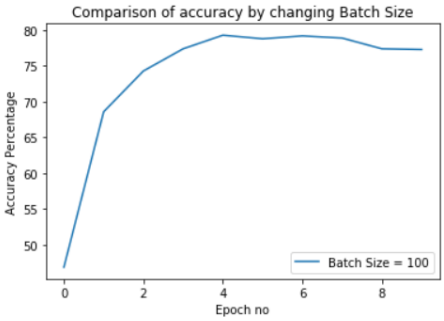
Learning Rate = 2, Epoch no = 20, Batch Size = 2



Learning Rate = 2, Epoch no = 20, Batch Size = 5

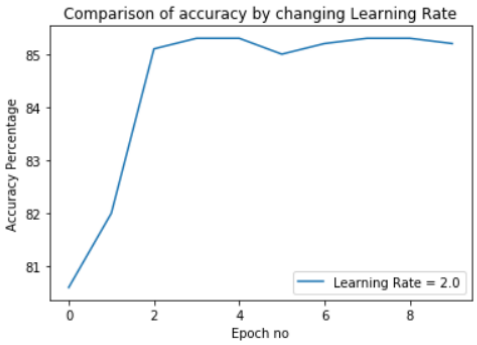


Learning Rate = 2, Epoch no = 10, Batch Size = 100

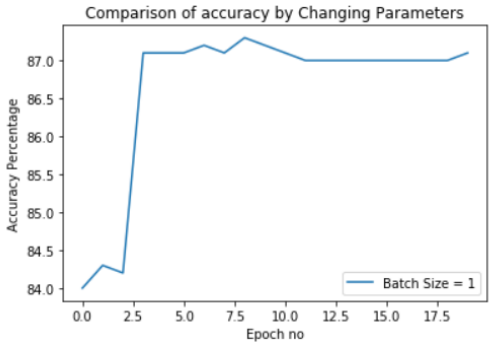


3. Epoch

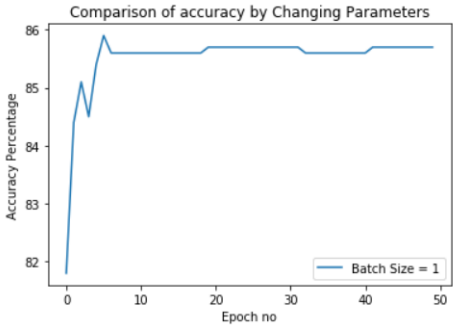
Learning Rate = 2.0, Epoch no = 10, Batch Size = 1



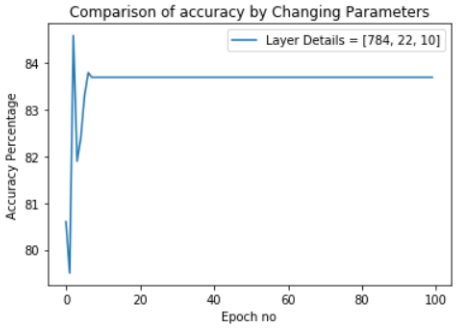
Learning Rate = 2, Epoch no = 20, Batch Size = 1



Learning Rate = 2, Epoch no = 50, Batch Size = 1



Learning Rate = 2, Epoch no = 100, Batch Size = 1

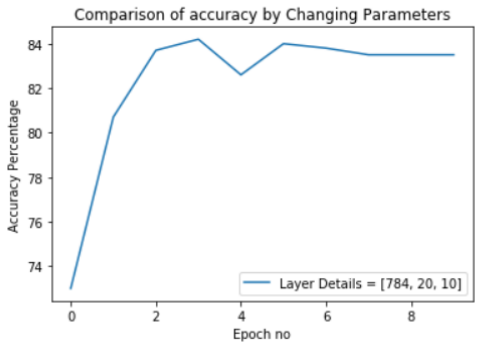


Parameters are Neurons as [784, 22, 10] with a learning rate of 2 on a batch size of 1 for 10 to 100 epochs. Changing epoch cycle doesn't change much

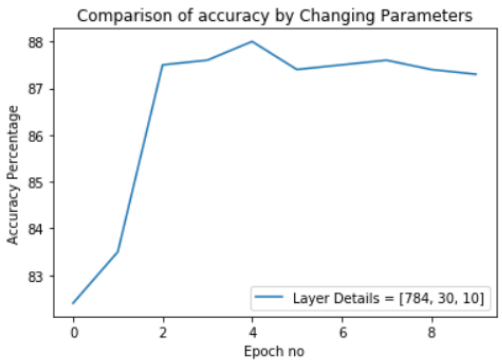
accuracy as our model's accuracy starts to enter in a saturation region after certain number of epoch cycles. These saturating epoch number might vary with different number of neurons and with learning rate but remains almost constant with respect to accuracy.

4. Number of Neurons

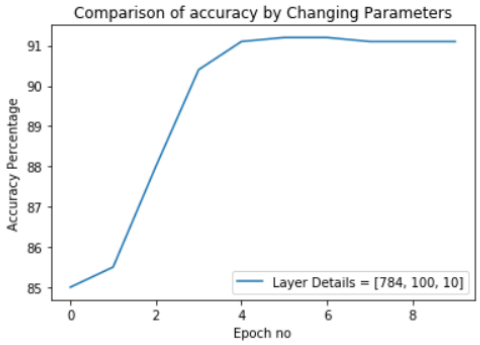
Learning Rate = 2, Epoch no = 10, Batch Size = 1



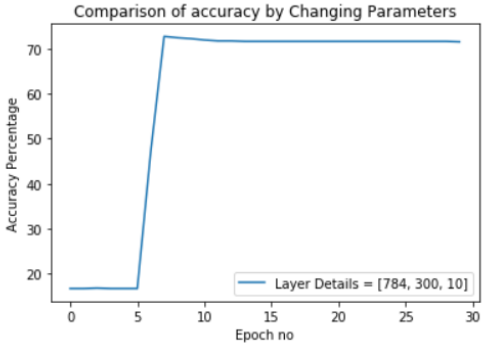
Learning Rate = 2, Epoch no = 10, Batch Size = 1



Learning Rate = 2, Epoch no = 10, Batch Size = 1



Learning Rate = 2, Epoch no = 30, Batch Size = 1

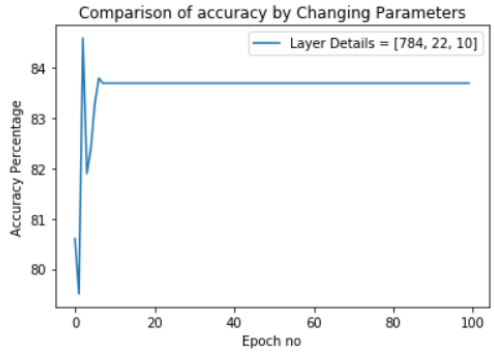


Since our input is 28*28-pixel image so input layer has 784 neurons and output layer has 10 neurons because of 10 possible digits (outcomes), so I changed number of neurons in hidden layer from 20 to 300 to observe changes. Clearly increasing number of neurons increased the efficiency up to 91% (~100 neurons) but then drastically reduced to 70% when we used too much neurons (~300). Reason can be over sensitivity to noises since each neuron is sensitive to some features of data, hence increasing to higher number may be sensitive to noise also.

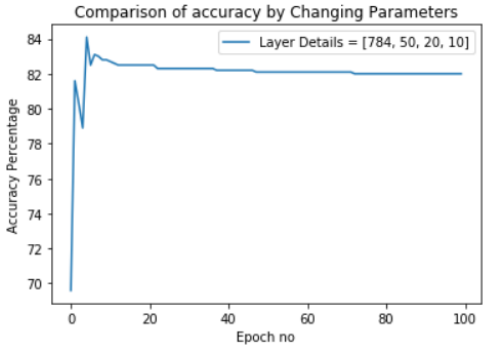
5. Number of Layers

Keeping learning rate, batch size and epoch same, we see that increasing number of layers is reducing the accuracy slowly. Reason can be overfitting of model due to multiple composition of function since we know that each layer is nothing but composition of activations function of previous layer.

Learning Rate = 2, Epoch no = 100, Batch Size = 1

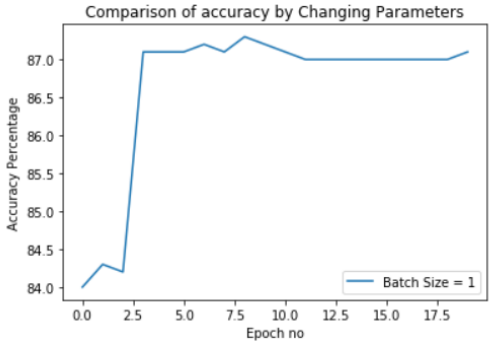


Learning Rate = 2, Epoch no = 100, Batch Size = 1

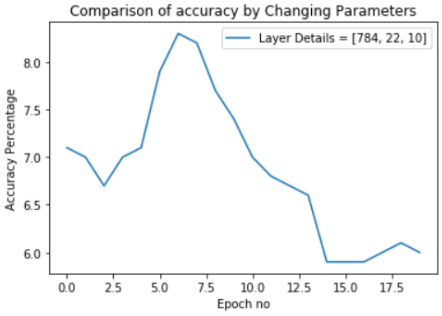


6. Activation Functions

Learning Rate = 2, Epoch no = 20, Batch Size = 1



Learning Rate = 2, Epoch no = 20, Batch Size = 1



I used two types of activation function, first is sigmoid and another image is tan hyperbolic. Clearly tan hyperbolic didn't give a remarkable accuracy due to fact that weights and biases initialization has been done between -0.1 to 0.1 which is good range for sigmoid but too small range for tan hyperbolic.