
RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

INTRODUCTION TO DEEP LEARNING - PROJECT REPORT

---

# FULLY CONNECTED NEURAL NETWORK USING NUMPY FOR HANDWRITTEN DIGITS RECOGNITION

---

Vikhyat Dhamija
RUID: 194003013

# Table of Contents

## Contents

# 1. Abstract

Neural Network is an important constituent of the world of Artificial Intelligence which uses the structure similar to brain to develop algorithms that can be used to model complex patterns and prediction problems. In this project , one such problem of Handwritten Digit recognition has been dealt with , for which we have been able to come out with Fully Connected Neural Network model which has been able to achieve more than 97 percent accuracy in very quick time. In this project both strategies of the Batch Gradient Descent and Mini Batch Gradient Descent have been tried and further their performances in the project have also been compared along with their thorough discussions. In this project , the concept of moving averages has also been explored which has proven to very useful in reducing the training time of the Neural Network. Further , the regularization loss addition technique has also been used in order to prevent the overfitting of the model . The main thing about this project is that the whole implementation has been undertaken using NumPy Libraries in Python which support large arrays and matrices mathematical functions, hence this project provide with an in-depth learning about the underlying mathematics and logic behind the Neural Networks.

# 2. Introduction

The human visual system is one of the wonders of the world. Many People effortlessly recognize the Hand-written digits. In each hemisphere of our brain, humans have a primary visual cortex, also known as V1, containing 140 million neurons, with tens of billions of connections between them. And yet human vision involves not just V1, but an entire series of visual cortices - V2, V3, V4, and V5 - doing progressively more complex image processing.The difficulty of visual pattern recognition becomes apparent if we attempt to write a computer program to recognize digits like those above. Simple intuitions about how we recognize shapes - "a 9 has a loop at the top, and a vertical stroke in the bottom right" - turn out to be not so simple to express algorithmically. If we try to make such rules precise, we quickly get lost in a morass of exceptions and caveats and special cases.

Here, the Neural Networks come to our rescue. Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples, and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. Hence , produces a Function approximation for a given task.

Neural Network can be easily implemented using the various Libraries like Pytorch , Tensor Flow but in this project , the whole implementation of the Fully Connected Neural Network has been done using the NumPy Library in Python ( NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.) which has helped us create the whole Neural Network in terms of

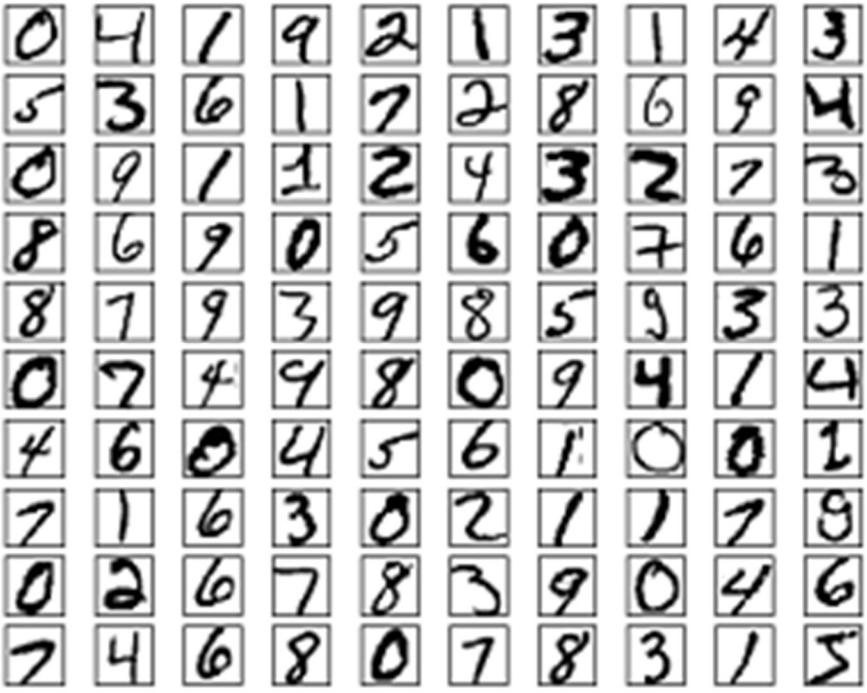
Matrices/Vectors Calculations and Manipulation and hence provided with the in-depth learning about the underlying mathematics and logic behind the Neural Networks.

# 3. Data Set

## 3.1. Description

In order to train and test our Neural Network , we have used the MNIST handwritten digit dataset which is a standard dataset widely used in computer vision and deep learning studies.

The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 60,000 small square 28×28-pixel grayscale images of handwritten single digits between 0 and 9 for training and 10,000 small square 28×28-pixel grayscale images of handwritten single digits between 0 and 9 for testing.



The task is to classify a given image of a handwritten digit into one of 10 classes representing integer values from 0 to 9, inclusively.

# 4. Model

## 4.1. Description

The structure of our Layered FC Neural Network is **784-200-50-10** .

**Input Layer :** 784 neurons - because each image in MNIST dataset is 28x28 which needs to be stretched to a length-784 vector.

**Hidden Layer 2 :** 200 neurons

**Hidden Layer 3:** 50 neurons

**Output Layer :** 10 neurons

The two hidden layers are followed by ReLU activation layers. The output layer has a Soft max activation layer in order to give classified outputs. 10 number of neurons in the output layer signify the 10 types of digits which are to be recognized.

# 5. Method

## 5.1. Description

In this section the detailed mathematics behind the neural network will be explained with snippets of the code used in the project code .

### a. Loading of the data and reshaping the Data

```
#x_train = np.load('../x_train.npy')
#y_train = np.load('../y_train.npy')
#x_test = np.load('../x_test.npy')
#y_test = np.load('../y_test.npy')
x_train, y_train, x_test, y_test = load()
x_train = x_train.reshape(60000,784)
x_test  = x_test.reshape(10000,784)
y_train=y_train.reshape(1,60000)
y_test=y_test.reshape(1,10000)
x_train = x_train.astype(float)
x_test = x_test.astype(float)
y_train=y_train.astype(int)
```

In this part of the code the loading of the Mnist Data and the reshaping has been performed. As conveyed , the dataset we are having is the 60000 28 * 28 matrices i.e. the gray scale images with 0-255 values for training and 10000 28 * 28 matrices for testing.

Here those image matrices are converted into 784 value vectors and hence after that we have 60000 rows of 784 values that corresponds to each image for training data and 10000 rows of 784 values for testing Data.

### b. Transposing the Test Set

In mathematics performed in this Neural Network implementation , we have transposed the 60000 rows of 784 values for training dataset and 10000 rows of 784 values for testing dataset . So after this step we will have 60000 columns of 784 values for training dataset and 10000 columns of 784 values for testing dataset.

```
#Transposing the test
x_train = x_train.transpose()
x_test=x_test.transpose()

x_train1 =x_train[:,range(num)]
y_train1=y_train[:,range(num)]
```

### c. Initialization of the Weight Matrices

This is an important step in our Neural Network implementation. For example, we have 784 neurons as an input for the first hidden layer which has 200 neurons . So if we mathematically visualize the operation is like this :

Each Neuron of the 200 neurons is performing the weighted sums of 784 values so it has 784 weights for it . So that means we have 200 * 784 matrix of weights for one hidden layer. That means that we have 200 rows of 784 values, so it is a 200 * 784 matrix of weights.

Using the same concept , we can visualize the Hidden layer 2 has 50 * 200 dimensions weight matrix and Output Layer has 10 * 50-dimension matrix.

Note that bias is also added by each neuron, so we also have Number neurons with one value of bias.(200*1,50*1,10*1 size matrices)

```
#Random Initialization of W arrays for  784-200-50-10

w_1=np.random.randn(200,784)*0.01
```

```
b_1=np.zeros((200,1))

w_2=np.random.randn(50,200)*0.01
b_2=np.zeros((50,1))


w_3=np.random.randn(10,50)*0.01
b_3=np.zeros((10,1))
```

So as described the Weights are initialized with the random values and the Bias are initialized with zero values.

## d. Forward Calculations

```
 #Layer 1 linear regression

z_1=np.dot(w_1,x_train)+b_1 #here b_1 is broadcasted for the 60000 training images
a_1=np.maximum(0,z_1)       # Relu activation function

 #Layer 2 linear regression

 z_2=np.dot(w_2,a_1)+b_2  #here b_2 is broadcasted for the 60000 training images
 a_2=np.maximum(0,z_2)

 #Layer 3 Logistic regression for output
 z_3=np.dot(w_3,a_2)+b_3  #here b_3 is broadcasted for the 60000 training images

 exp_scores=np.exp(z_3)
 a_3=exp_scores/np.sum(exp_scores, axis=0,keepdims=True)
```

Here:

Dot operation has been performed between the input and the weight matrix, which is nothing, but the weighted sum calculated of input column using the weight matrix at the particular layer. For example, for the First hidden layer 784 inputs and 200*784 matrix – dot product leading to the 200*1 matrix which then passed through the activation layer , in our case RELU Layer i.e. which passes every output greater than 0 as it is and the other less than 0 turned to 0.

So similar thing was performed for the second hidden layer and finally for the output layer with soft max layer.

### e. Loss calculation after forward pass

As after the forward pass we have actually calculated the output with respect to the function which has been set up based on the Neural Network.

Then we calculated the loss that how our result is far/close from the actual/correct result and based on that we will back propagate to modify our Weight , Bias parameters which are the inner parameters of the function set up by neural network.

Loss calculation type is the Cross-Entropy Loss:

```
#Loss Function
loss=-np.log(a_3[y_train,range(0,y)])
f_loss=np.sum(loss)/y
```

Cross entropy Loss is the – (summation of (actual output * Log of function output))

As we have only 1 as actual output for one of the 10 values and based on that our output is the -1* Log (output function value corresponding to 1 in the 10-value output vector).

Note that divide by y is because we have averaged it out for all inputs losses in a batch of y.

### f. Vectorization explained

In the above , the explanation is being done from the perspective of the fact that we have 784 neurons input from one 28*28 matrix, but we have 60000 images of training set. So we use vectorization to use all 60K images or a part of it in case of mini-batch gradient descent to set the parameters and checking loss.

So it can be visualized that there are 60K or y columns of 784 values (0-255) and they are being passed together through each layer and hence we have 60K or y columns output of Layer wise neurons like 200 , 50 values and 10 values produced through each hidden layer and the output layer respectively.

This vectorization is a very important concept in our training and testing through various images together.

## g. Regularization

```
#need to add regularization loss
reg_loss=0.5 * reg
(np.sum(np.square(w_3))+np.sum(np.square(w_2))+np.sum(np.square(w_1)))
total_loss+= (f_loss + reg_loss)
```

As it can be seen that regularization loss has been added to the loss calculated in order to prevent over fitting of our parameters to the input training data so that in case of testing they may not diverge from the actual output.

Note that reg is a hyper parameter to calculate the regularization loss.
Further calculation is simple that we are taking the sum of the squares of all elements of W matrices and then multiplying by reg and 0.5 .

Then adding the regularization loss to come out with the total loss.

## h. Back Propagation

```
#Backward Propagation
dz_3=a_3
dz_3[y_train,range(0,y)]-=1


dw_3=np.dot(dz_3,np.transpose(a_2))
dw_3/=y
db_3=np.sum(dz_3,axis=1,keepdims=True)
db_3/=y

# back propagation into layer 2
dz_2=np.dot(np.transpose(w_3),dz_3)
dz_2[z_2 <= 0]=0 # Multiplied by diffrentiation of activation function
dw_2=np.dot(dz_2,np.transpose(a_1))
dw_2/=y
db_2=np.sum(dz_2,axis=1,keepdims=True)
db_2/=y

# back propagation into layer 1
dz_1=np.dot(np.transpose(w_2),dz_2)
dz_1[z_1 <= 0]=0 # Multiplied by diffrentiation of activation function
dw_1=np.dot(dz_1,np.transpose(x_train))
dw_1/=y
db_1=np.sum(dz_1,axis=1,keepdims=True)
db_1/=y
```

Let us discuss the back propagation :

Note in above code snippet , we are starting with the a_3 which is the soft max output i.e. the final output.

X F Y

Dy/Dx=F'( F prime)

M G X F Y

So In this case :

DY/DM= G' * F'

Note in the above the G' and F' are DG/DM and DF/DX respectively i.e. their differentiation with respect to their individual inputs.

So the above is the chain rule : so when we have sequence of functions like the above then we can reach to the diffrentiation of Output y w.r.t to that input value like X and M above by just calculating the differentiation of the individual functions in chain of functions and then multiplying them.( moving backwards)

Like for DX we calculated F' and For DM we have F' * G' . This is the chain rule which is the core foundation of the Neural Network. By this method we go block by block backwards and then calculate the change of the output with respect to that variable of the function and change that variable of the function in the direction to decrease the output as in our case the output is the Loss which we need to minimize.

So, our purpose is to get the (DY/D param) of all parameters W and B in our layers so as to change them opposite to the sign of the variation and magnitude of change decided by the step size.
Let us go step by step :-

Consider last layer 3:

A_2 → [W3 . A_2 +B_3] → Soft-max → Loss Function(Y,Y-hat)
Linear Function Z_3 Y-Hat

Dloss/Dz_3= (Z_3) i - Y i [There is a mathematical proof for it. Here i varies from 0 to 9]

So in conclusion we have to minus those values of Z_3 by 1 for which Yi is 1

So that is being done in the first step y_train provide the index and value for Y index is obviously 1 as that final true number is the 1 output out of 10 possible outputs.

```
dz_3=a_3
dz_3[y_train,range(0,y)]-=1
```

Then we calculate DW3
(D Linear Function/DW3) * Dy/Dz_3 = DY/Dw3

Which is :

dw_3=np.dot(dz_3,np.transpose(a_2))

Z_3 dimension here is 10 rows and y columns
A_2 Dimension is 50 rows and y columns
Transpose of A_2 is y*50

So dw_3 is 10*y . y*50 =10*50 matrix ( * is equivalent to , showing dimension of matrices)Then we divide each value of the matrix by y as each value in the matrix is the summation for all y and hence by dividing y we need to take average.

Similar to `Dy/DW_3` we calculate for DY/DA_2 or DZ_2

```
1. dz_2=np.dot(np.transpose(w_3),dz_3)
2. dz_2[z_2 <= 0]=0 # Multiplied by diffrentiation of activation function
```

Here 1 is basically DY/Da_2 as above and Then in the 2 , we are basically calculating the Relu diffrentiation which is 0 for all input < 0 and 1 other wise and hence we get DZ_2 where

Z_2 Relu Function A_2

Now as other layers are just Relu activation and Linear regression same things are replicated for other layers to calculate dw_2,dw_1 and db_1 , db_2

**i. Adding the regularization to dWs**

```
#adding regularization loss
 dw_3+=reg*w_3
 dw_2+=reg*w_2
 dw_1+=reg*w_1
```

Adding the differentiation of the Regularization loss( sum of squares of all w elements) we added to the final loss to each of w

**j. Changing the parameters in the opposite direction of the direction of change of Loss with respect to them**

```
#Calculate the moving average
 v_w1= (1-b)*dw_1 + b*v_w1
 v_w2= (1-b)*dw_2 + b*v_w2
 v_w3= (1-b)*dw_3 + b*v_w3

 v_b1= (1-b)*db_1 + b*v_b1
 v_b2= (1-b)*db_2 + b*v_b2
 v_b3= (1-b)*db_3 + b*v_b3

 #Learning based on the derivatives
 '''w_1+= (-step_size * dw_1)
 w_2+= (-step_size * dw_2)
 w_3+= (-step_size * dw_3)
 b_1+= (-step_size * db_1)
 b_2+= (-step_size * db_2)
 b_3+= (-step_size * db_3)'''

 #Learning based on the moving averages
 w_1+= (-step_size * v_w1)
 w_2+= (-step_size * v_w2)
 w_3+= (-step_size * v_w3)
 b_1+= (-step_size * v_b1)
 b_2+= (-step_size * v_b2)
 b_3+= (-step_size * v_b3)
```

Note that I have used two methods in my training ( not at the same time that's why one is commented) one is normal Learning based on derivatives and other is the concept of moving averages.

Note by the concept of moving averages , the derivative of loss with respect to any parameter do not take parameter here and there too much and just make the path towards the minima point in the loss function fast and smooth

**So, the above are the discreet parts in one cycle of forward and backward pass.**

**k. Hyper parameter selection**

```
#hyperparameters
step_size=0.01
reg=1e-3
b=0.8 #momentum
```

Step_size , reg constant and momentum are the Hyper parameters selected for the better performance of our Neural Network.

# 6. Procedure and Results

## 6.1. Description

1. Mini Batch Gradient Descent with Batch Size of 128 and 10 epochs with Momentum=0.9 and Regularization loss to prevent overfitting

```
C:\Users\vikhyat\data_structures_algo\deep_learning_homework\Final_project>python Final_FC_NN.py
The Total average loss after the epoch number  0 is :  1.4276214578002793
The Total average loss after the epoch number  1 is :  0.5259172895165155
The Total average loss after the epoch number  2 is :  0.36351542857716335
The Total average loss after the epoch number  3 is :  0.27740743314573074
The Total average loss after the epoch number  4 is :  0.22172597001242328
The Total average loss after the epoch number  5 is :  0.17867987020599982
The Total average loss after the epoch number  6 is :  0.14577622093647472
The Total average loss after the epoch number  7 is :  0.12267011692052825
The Total average loss after the epoch number  8 is :  0.10811827737125904
The Total average loss after the epoch number  9 is :  0.09916220425488319
Training accuracy :  0.9738
Training Time taken:  30.23827624320984
```

97 % accuracy in just 30 seconds

2. Mini Batch Gradient Descent with Batch Size of 128 and 10 epochs without Momentum and with Regularization loss to prevent overfitting

```
C:\Users\vikhyat\data_structures_algo\deep_learning_homework\Final_project>python Final_FC_NN.py
The Total average loss after the epoch number  0 is :  1.3497112182312538
The Total average loss after the epoch number  1 is :  0.4955767061533177
The Total average loss after the epoch number  2 is :  0.3449577985432909
The Total average loss after the epoch number  3 is :  0.25958782518252926
The Total average loss after the epoch number  4 is :  0.20136653862171391
The Total average loss after the epoch number  5 is :  0.16282271689218736
The Total average loss after the epoch number  6 is :  0.1315512534422
The Total average loss after the epoch number  7 is :  0.11072552968100371
The Total average loss after the epoch number  8 is :  0.1011973885741633
The Total average loss after the epoch number  9 is :  0.08901558366546412
Training accuracy :  0.9773
Training Time taken:  23.819546937942505
```

Almost Same performance as above (1) case

3. Full 60000 batch Gradient Descent and 500 epochs without Momentum but with Regularization loss to prevent overfitting

```
C:\Users\vikhyat\data_structures_algo\deep_learning_homework\Final_project>python Final_FC_NN_FBatch.py
The Total loss in these  0  iterations is :  2.3041838230971683
The Total loss in these  100  iterations is :  0.31754088587910123
The Total loss in these  200  iterations is :  0.20589015392018165
The Total loss in these  300  iterations is :  0.16158948243621435
The Total loss in these  400  iterations is :  0.13397630739828248
Training accuracy :  0.9641
Training Time taken:  505.10824942588806
Testing Time taken:  0.09059810638427734
```

96.41 % accuracy in in almost 9 Minutes which shows Mini batch SGD is fast in our case.

# 7. Conclusion

## 7.1. Target achieved

The target of 90% accuracy in setting up our FC Neural Network on MNIST Dataset has been achieved using the Batch Gradient Descent as well as the Mini Batch Gradient Descent , totally using the NUMPY Library functions.

## 7.2. Performance comparison of Batch GD and Mini Batch GD

Discussion

Batch gradient descent computes the gradient using the whole dataset. In this case, we move somewhat directly towards an optimum solution, either local or global. Additionally, batch gradient descent, given an annealed learning rate, will eventually find the minimum located in its basin of attraction.

Stochastic gradient descent (SGD) computes the gradient using a single sample. But in most applications of SGD actually use a minibatch of several samples.( Mini Batch Gradient descent). This Mini Batch SGD works well for error manifolds that have lots of local maxima/minima. In this case of SGD , we feel that the somewhat noisier gradient calculated using the reduced number of samples tends to jerk the model out of local minima into a region that hopefully is more optimal.

If we use the single samples the losses, we get are really noisy that means they not always travel down to local minima , while minibatches tend to average a little of the noise out. Thus, the amount of jerk is reduced when using minibatches.

We have to try out and find the suitable minibatch size which is small enough to avoid some of the poor local minima, but large enough that it doesn't avoid the global minima or better-performing local minima. Generally, they are chosen in the powers of 2 , which makes them more convenient for performance in computer architecture( we have chosen 128).**So , in nutshell from above discussion , it depends , that you may stuck in an optimal solution / local minimum and come out with solution, but it is also possible that you are away from the best optimal minima. In our case this has not been issue I think that for the dataset used , we are able to reach an optimal solution using our Mini Batch SGD.**

But one major benefit of Mini batch SGD is that it is computationally a whole lot faster. Large datasets often cannot be held in RAM, which makes vectorization much less efficient. Rather, batch of samples must be loaded, worked with, the results stored, and so on. Usually, this computational advantage is leveraged by performing many more iterations of SGD, making many more steps than conventional batch gradient descent. ( in our case ((60,000/128 )*10) steps in Mini Batch SGD )This usually results in a model that is very close to that which would be found via batch gradient descent, or better. **This can be seen in our results which has been trained very faster in case of Mini Batch GD.**

## 7.3. Performance of Momentum in Mini Batch SGD

Discussion

In case of Mini Batch SGD , we do not compute the exact derivate of our loss function. Instead, we are estimating it on a small batch. Which means we are not always going in the optimal direction, because our derivatives are ‘noisy’. So, in that case , the use of Momentum i.e. the moving averages or an exponentially weighed averages can provide us a better estimate which is closer to the actual derivate than our noisy calculations. This is one of an important reason that why momentum work better in case of Mini Batch SGD.

Inference in our Project

In case of our project , it seems that we are going in the optimal direction hence our moving averages are not able to enhance the performance that much. This is an inference based on the experimental performance on the dataset used.

# 8. References


1. neuralnetworksanddeeplearning.com
2. https://stats.stackexchange.com/questions/49528/batch-gradient-descent-versus-stochastic-gradient-descent
3. https://math.stackexchange.com/questions/945871/derivative-of-softmax-loss-function
4. DeepLearning.AI Stanford Professor Andrew Ng. Lectures Online
5. Course Material , Introduction to Deep Learning , by Prof. Bo Yuan
6. https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d